SWARMFILE

Documentation

Version 0.3.17

Generated 2026-11-26 from https://swarmfile.com/docs

The complete consumer documentation: guides, admin and IT, reference, and the command-line tools. The always-current version lives at https://swarmfile.com/docs. This PDF was generated from the same markdown the site renders.

Contents

Guides

Getting Started
Coming from Git
Moving from Perforce
Migrating Existing Data
Deployment Topologies
Working with Files
Working Offline (Pack & Go)
Multiple Mounts
File Previews
Search
Notifications & Inbox
Sync Exclusions
Performance & Disk Usage
Version Control
Branches & Merging
Trash, History & Rollback
Line Attribution
Verifiable History
Git and Swarmfile
Cloning a Repository with Git
The Zero-Install Demo
Git-LFS
Coding Agents
VS Code Extension
Runner (Headless CI)
Hosted CI
Self-Hosted CI Runners
Automating Swarmfile
Sharing & Collaboration
Anonymous Drops & Claiming
Worksharing (no plugin)
Ableton Live Collaboration (Stem Handoff)
RFIs & Submittals
Public Repositories & Org Profiles
Publishing Quickstart (Open Source)
Publishing Releases
Troubleshooting
Open Source Overview
Self-Hosting the Community Server
Seeding a File Into a Repository
Mirroring a Repository Between Hubs
Diagnosing Connectivity
Uninstall & Clean Reinstall

Admin & IT

Organizations, Repositories & Members
Identity
LDAP Directory Sync
Permissions
Self-Hosted Seed Nodes
Security
Security Architecture (Whitepaper)
Trust & Compliance
End-to-End Encryption Setup
Dedicated Storage Isolation
Bring Your Own Storage
Data Residency
Billing & Plans

Operations
Webhooks
Deploying to Your Team
Data Portability & Offboarding
Branch Mirror to S3

Reference

Filesystem Compatibility & Conformance
Supported Platforms & System Requirements
Release Channels & Updates
Diagnostics & Error Reference
Environment Variable Index
Engine Config File (config.json)
Network Requirements
Storage Format
Telemetry & Data Collection
Known Limitations

CLI Reference

swarmfile
swarmfile-doctor
swarmfile-migrate
swarmfile-search
swarmfile-seed
swarmfile-mirror
swarmfile-brlock
swarmfile-runner
swarmfile-verify-history
swarmfile-engine lfs-transfer
git-remote-swarmfile

Getting Started

Swarmfile mounts your team's repository storage as a real local drive. It is not a sync folder — nothing downloads to your machine until you actually open a file. Browsing a folder with tens of thousands of files costs zero disk space. Opening a 200GB file streams only the bytes your app reads.

Installing the client

Download the installer for your platform from Downloads.

- **macOS** — a signed and notarized universal `.dmg` / `.pkg` that runs on Apple Silicon and Intel. The client uses FUSE-T, so there is no kernel extension: no Recovery-mode "Reduced Security" step, no rebooting to approve a kext, no security prompts beyond the normal Gatekeeper install.
- **Windows** — a Setup bundle that chains the required dependencies (WinFsp, the VC++ Redistributable, and WebView2) automatically, so you don't need to install them separately. Both ARM64 and x64 builds are supported.
- **Linux** — a `.deb` package (Debian/Ubuntu). The download is checksum-only unless a release publishes the GPG-signed apt repository; when that repository is available, install or upgrade through it so apt verifies the package signature. Need an `.rpm` (Fedora/RHEL/openSUSE)? Get in touch and we'll provide one.

The `.pkg`, MSI, and `.deb` installers also put the `swarmfile` command line on your `PATH`, along with the standalone tools (`swarmfile-lfs`, `swarmfile-doctor`, `swarmfile-migrate`, `swarmfile-search`, `swarmfile-verify-history`, `swarmfile-seed`, `swarmfile-runner`) and the `git-remote-swarmfile` helper that backs `git clone swarmfile://` — see CLI: swarmfile. There is no separate CLI download.

On macOS, the `.pkg` is what installs those `PATH` entries: the `.dmg` installs the Desktop App, whose bundle carries the same tools but doesn't link them into `/usr/local/bin`, so a `.dmg`-only install needs one `.pkg` (or **Diagnostics → Reinstall**) run to put the CLI on `PATH`.

(`swarmfile-brlock`, a reference client for plugin developers, isn't part of any install; contact us if you're building a worksharing integration.)

Windows: the SmartScreen prompt

The first time you run the downloaded installer, Windows may show a blue **"Windows protected your PC"** box from Microsoft Defender SmartScreen. This is expected: the `Setup.exe` and `.msi` installers, the executables inside the desktop payload archives, and the standalone `swarmfile-doctor` download are Authenticode-signed by the release pipeline, but SmartScreen warns until the signing identity has built up download reputation.

(The Windows tool `.zip` downloads are signed by a separate job that is fail-closed too: it blocks the tool release rather than shipping an unsigned archive, and the desktop release refuses publication instead.) Either way, it does not mean the download failed or was tampered with.

1. Click **More info** — the small link, not **Don't run**; the **Run anyway** button appears underneath.
2. Click **Run anyway**.
3. If Windows asks for permission, choose **Yes**, then finish the installer as usual.

You see the prompt when you run a freshly downloaded installer; automatic updates inside the app don't show it. A scripted mass deployment (`msiexec`, GPO/SCCM/Intune) never sees it — see Deploying to Your Team.

Signing in

Open the client and sign in with your email and password, or with your organization's SSO if your IT team has configured an identity provider. The default form is email/password against the built-in identity provider; for SSO, click "Sign in with SSO" and enter your **organization's slug** (the short name in your org's Swarmfile URL) — the client looks it up and shows a "Continue with `<your IdP>`" button if that org has SSO configured.

There's no email-domain auto-detection: you need to know your org's slug (or use the direct per-org sign-in link your admin can share) rather than just entering your email and having it figured out for you. An organization on Starter or above can also require an authenticator app, a verified email, or both before it admits members; if yours does, you're prompted to enroll before enforcement starts.

If your account has an authenticator app, sign-in asks for its current six-digit code after your password (enter a recovery code instead if you don't have the app), and a wrong code can simply be retried. See Identity.

Switching to a **different account** while the app is running is refused (`account_switch_requires_restart`): the local cache identity is reconciled only at startup, so that one user's unsent work can never be uploaded as another's. Sign in with the other account and restart when the Desktop App offers it — the requested sign-in is kept, and the restart completes the switch; any unsent work from the previous account is quarantined on your machine rather than lost.

New to Swarmfile? Starter and Pro start with a **7-day free trial** covering up to 2 seats (signing up past 2 seats starts paid), and a no-card Free plan covers public repositories and private unencrypted repositories with a flat 5 GiB of storage — see pricing.

Creating a repository

If your organization doesn't have the repository yet, create one from the Desktop App or the web app. Every seeded file is a skeleton you edit; nothing is committed without your change.

Start from a template

Templates pre-set the encryption tier, commit mode and default-branch protection:

- **Blank** starts empty.
- **Media & post** seeds a starter folder layout and an ignore file.
- **Music production (Ableton Live)** seeds a multi-song `Songs/` layout, per-role stem folders, a handoff manifest, a stem validator and a session companion (see Ableton Live Collaboration). The **DAW-agnostic** variant uses an opaque file-based project for Pro Tools, Cubase or Studio One; Logic bundles are accepted but need a Pack & Go checkout.
- **Software (git repository)** seeds just the ignore file.

- **Open source repository** creates a public, git-cloneable repository seeded with `README.md`, `CONTRIBUTING.md`, `SECURITY.md`, `CODEOWNERS`, a `.swarmfile/publicignore.yml` starter and a fully commented-out `.swarmfile/ci.yml` — plus a `LICENSE` placeholder. It never guesses a license; a TODO tells you to paste one.
- **Revit worksharing** and **Geospatial** pre-set their defaults and seed a starter folder layout plus an ignore file.

A template that seeds content cannot be used on an **end-to-end encrypted** repository — the hub holds no key, so the create is refused. Start Blank there, or create the repository on a plaintext or managed tier and keep the seed.

Two choices are fixed at creation

- **Initial branch.** This is the repository's default branch: most repositories keep `main`; choose another (for example `develop`) only if your tooling expects it. It cannot be changed later, it is the branch a fresh clone checks out, and the web app opens the repository on it — so a `git clone` of a repository whose default is `develop` lands on `develop`, and its Files, Commits and Runs views start there too.
- **Encryption tier.** Managed (the default for paid private repositories), end-to-end, or plaintext (available for public repositories and for any Free-plan repository). This is fixed at creation; upgrading later does not retroactively encrypt an existing repository. See Security Architecture.

To inspect a repository's effective settings — every option's value and where it came from — run `swarmfile repository show`; change them with `swarmfile repository set <key=value>` (see the CLI reference).

Mounting a repository

Once signed in, the Desktop App shows the repositories you have access to. Pick one and it appears as a drive on your machine — a drive letter on Windows, a mount point on macOS and Linux. The exact location is in the Desktop App's header: click the **Status pill** to open the panel showing the mount point and sync state. It's assigned at mount time and can vary between machines (for example, if a default drive letter is already taken, Swarmfile picks the next free one).

From here, the mount behaves like any other local filesystem. Open it in Finder, Explorer, or your file manager, or point a native app directly at a file inside it.

First open

The first time you open a file, only the bytes your application actually reads are streamed down. Scrubbing 30 seconds into a 200GB video timeline streams roughly 30 seconds of footage, not the whole file. Browsing folders, previewing thumbnails, and listing directories don't trigger downloads at all — that metadata is served directly.

Figure omitted in print (animation): A machine lists a 64 MB file it has never fetched and its local cache does not grow at all; it then reads one megabyte from the middle and the cache grows by a few megabytes, not sixty-four. - https://swarmfile.com/demo/sparse-streaming.cast

Presenting the Desktop App

Cmd/Ctrl+Shift+P toggles **presentation mode**, the on-camera switch for demos and screen recordings. Text scales to the largest size so a 1080p capture stays readable after compression. People, emails, organizations, repositories and drives are replaced with stable pseudonyms ("Person AB"), mount paths are hidden, and live name/secret/path fields are blurred, so a customer's details can't slip into a video. File and folder names stay real unless you turn on the extra name mask in Settings → Presentation. The switch is local to your machine — it never changes what teammates see.

Prefer larger text all the time, no recording involved? The Desktop App's **Settings → Text size** offers Compact, Default and Large steps for that computer (Cmd/Ctrl + and – step it, Cmd/Ctrl 0 returns to Default); presentation mode always uses the largest step.

For multi-machine demos, presentation mode can also show a small always-on-top clock overlay, with the time synced from the network so machines shown side by side agree.

Where to go next

- Working with Files covers browsing, saving, presence, and locking in more depth.
- New to Swarmfile's own version control? Version Control covers changesets, branches and rollback, and Branches & Merging covers working an option in isolation and landing it.
- Working in git? Clone a Repository with Git covers `git clone swarmfile://` and `git push` against a repository's git view, and Git and Swarmfile lays out the three ways the two fit together. Starting from a repository that already exists elsewhere? `swarmfile git import <url>` creates the repository from it in one command — see the import reference.
- Publishing an open-source release? The Publishing Quickstart goes from repository to public release (no-account downloads, assets, contributors) in one page.
- If you're moving data in from an existing NAS or file server, see Migrating Existing Data.
- Rolling this out to a whole team rather than installing by hand? See Deploying to Your Team.

Coming from Git

If you know Git, you already know most of Swarmfile. The vocabulary is deliberately familiar — `commit`, `branch`, `merge`, `log`, `revert`, `tag` all mean what you expect. This page maps your Git habits onto Swarmfile and, more importantly, flags the few places where the mental model genuinely differs.

Swarmfile is its own tool, not a Git front-end. But the instincts carry over, and where a Git command has no equivalent, typing it anyway prints a one-line pointer instead of an error — so you can learn by doing.

The one big shift: mount instead of clone, and no save step

This is the whole difference in two sentences:

- **Your repository is a live mounted drive, not a cloned copy.** For day-to-day work there's nothing to `clone`, `pull`, or `push`. Files stream from the cloud on demand; you open and save them like any local file, and other people's changes appear as you read. (When you genuinely want a git repository of a repository — for git tooling, a build that expects one, or an offline archive — an owner can turn on git access and you can `git clone` it, and even `git push` commits back to it. The drive stays where day-to-day work happens.)
- **Saving is automatic.** There's no `add` / `commit` / `push` ceremony to make your work durable and shared — the moment you save, it starts syncing. Git fuses *saving* and *naming* into one mandatory step; Swarmfile splits them, so **naming a commit is optional, and you can do it whenever — even after the fact.**

Everything below follows from those two facts.

Cheat sheet

In Git	In Swarmfile	Notes
`git clone <url>`	*(nothing)* — just mount · or `git clone swarmfile://<org>/<repository>`	Repositories appear as a mounted drive once the app is running; `swarmfile status` shows where it's mounted. If the repository has git access turned on, `git clone swarmfile://…` gives you a real git repository of it — see Cloning a Repository with Git.
`git pull` / `git fetch`	*(nothing)* — or `git fetch` in a `swarmfile://` clone	The drive is live — a teammate's changes appear as you open files. `swarmfile log` shows recent commits. Note `swarmfile fetch` is **not** Git's fetch: it pulls a byte range of one file (including one still uploading).
`git push`	*(nothing)* — or `git push` from a `swarmfile://` clone	Saves stream to the cloud automatically. `swarmfile status` shows what's still uploading. A git clone of a git-enabled repository can push commits back, fast-forward only.
`git add <path>`	`swarmfile add` (opens a changelist if none)	No staging for everyday work — just save. `add` is idempotent: it opens a changelist so subsequent saves stage into it (and this mount stays in staged mode until you `commit`), and it exits 1 on a path that doesn't resolve. For a deliberate multi-file batch that lands all at once, see Version Control.
`git commit -m "…"`	`swarmfile commit -m "…"`	Names a specific save. You don't *have* to — unnamed autosave commits happen for you as you work.
`git commit --amend -m "…"`	`swarmfile commit --amend -m "…"`	Renames the most recent commit. **Message-only and non-destructive** — contents and history are untouched (Swarmfile never rewrites commits).
`git status`	`swarmfile status` (mount/engine) · `swarmfile changelist status` (staged) · `swarmfile diff` (branch vs base)	`swarmfile status` is *not* Git's working-tree status — it reports the mount, peers, and settings. There's no "modified but unsaved" state to show: saves sync immediately. Use `changelist status` for staged work and `conflicts` for blocked files.
`git log`	`swarmfile log`	Compact, `--oneline`-style feed.
`git show <commit>`	`swarmfile show <N>`	Whole-file: shows which files changed (A/M/D/R), not a line diff — Swarmfile versions binaries.
`git diff main...`	`swarmfile diff [<branch>]`	Previews what a branch would merge into its base — files added/modified/deleted/renamed, plus conflicts. `--format=unified` / `side-by-side` renders a text file's content (with intraline highlighting); binaries stay whole-file.
`git diff A B`	`swarmfile diff <A> <B>`	Files changed between two commits (each a number, `HEAD~N`, tag, or branch). `--format=unified` / `side-by-side` renders text content; binaries stay whole-file.
`git blame <file>`	`swarmfile blame <path> [--lines]`	Who last changed the file + its version history with authors. The default is one row per saved version; `--lines` gives per-line attribution (`--format=gutter` for a blame-style gutter).
`git switch <branch>`	`swarmfile switch <branch>`	Hot, in-process — no restart.
`git checkout <branch>`	`swarmfile switch <branch>`	`swarmfile checkout <branch>` also works (it's a shim); `switch` is clearer.
`git checkout -b <name>` / `git switch -c <name>`	`swarmfile checkout -b <name>` / `swarmfile switch -c <name>`	Creates the branch (forked from your current one — or, like git, from `checkout -b <name> <start-point>`) and switches to it in one step.
`git restore <path>` / `git checkout <commit> -- <path>`	`swarmfile restore <N> --path <path>`	Rolls a file, folder, or the whole repository back to a past commit, landed as one undoable commit.
`git revert <commit>`	`swarmfile revert <N>`	Lands a new commit undoing an old one. Non-destructive and itself undoable.
`git branch`	`swarmfile branch list`	
`git branch --merged`	`swarmfile branch merged`	Branches with nothing left to land (branch-vs-base diff empty); `swarmfile branch prune` archives them after confirming, and never archives a branch an open mount on this machine is working on.
`git add --resolved` (staging resolved conflicts)	`conflicts resolve <path> --auto`, `--winner S`, `--tool`	There's no index to stage into, and a conflicted file can't be saved until it's resolved — so a leftover-marker commit can't happen. Resolve, then saves sync. `--auto` already refuses (exit 2) if a merge leaves a real conflict.
`git tag <name>`	`swarmfile tag create <name>`	An immutable, named pointer to a commit — handy for "branch from here later."
`git merge`	`swarmfile merge`, or a merge request: `swarmfile mr create`	Whole-file merge with conflict detection. See Branches & Merging.
`git mergetool`	`swarmfile conflicts resolve <path> --tool`	Uses your git mergetool configuration, unless a `mergetool_cmd` is set in the engine's own config — that takes precedence over git config.
(auto-merge)	`swarmfile conflicts resolve <path> --auto`	Let the engine merge it (diff3 for text) with no external tool — exits 2 if a real conflict remains, so a script can fall back. `--all --auto` sweeps every conflicted file in one pass.
`git rm <path>`	*(do it on the drive)*	Delete the file on the mounted drive; it syncs like any other save and lands in Trash.
`git mv <a> <b>`	*(do it on the drive)*	Rename/move on the mounted drive. The entry keeps its id, so history, comments, and locks move with it.
`git worktree add`	`swarmfile mounts open <path>`	One engine can hold several full mounts at once — the reason you reached for a worktree. Use `--repository <id>` for a different repository (same org).
`git grep`	`git clone` + `git grep`, or `swarmfile-search`	Swarmfile's own search is metadata: names and paths, not file contents (the scoped exceptions are Explore's README/path text, a public release's file-content search, and RFI/submittal

In Git	In Swarmfile	Notes
		bodies). To grep contents, work in a `git clone` of a git-enabled repository; `swarmfile-search` (or the dashboard's search box) finds entries by name.
`git clean -fd`	*(nothing needed)*	Ignored paths are already kept off the hub. To remove content that synced before a rule existed, use `swarmfile ignore-clean`.
`git reflog`	`swarmfile log` / `swarmfile activity list`	History isn't rewritten, so there's no reflog — `log` lists commits, `activity` shows who changed what and when.
`git init` / `git remote`	*(nothing)*	There's no repo to initialize and no remotes — the drive is the repository.
`.gitignore`	`.gitignore` / `.swarmfileignore`	Honored on sync. See Sync Exclusions.

`<N>` is a commit number — the `#N` shown by `swarmfile log`. Anywhere a commit is expected (`show`, `restore`, `revert`, `describe`, `checkout <commit>`) you can also use `HEAD`, `HEAD~N` / `HEAD^` (N commits back), a **tag name**, a **branch name** (its head), a changeset id, or a 64-hex **commit hash** — just like Git. A `seq:` / `head:` / `branch:` / `tag:` / `changeset:` / `hash:` prefix forces which kind is meant.

> **`log` is per-branch, and shows a branch's *own* commits.** `swarmfile log` (and `HEAD` / `HEAD~N`) reflect the branch this mount is on. Note one difference from Git: a branch's log shows the commits made *on that branch since it forked* — not the shared history from before the fork point. Switch to `main` to see the mainline history.

> **Your Git aliases work.** If your `~/.gitconfig` defines `[alias] co = checkout`, `st = status`, etc., `swarmfile co` / `swarmfile st` expand the same way — Swarmfile reads your git aliases for any command name it doesn't already have.

Three habits to unlearn

1. **Don't reach for `add` / `commit` / `push` to "save and share."** Just save the file. It's durable and syncing before you'd have finished typing `git add`.
2. **Don't `clone` or `pull` to get to work.** The drive *is* the repository, always current. `swarmfile status` tells you where it's mounted and what's in flight. A `git clone` of a git-enabled repository is there for git tooling, archives and git-based workflows; it can push commits back, but it isn't the live drive.
3. **You don't have to name a commit when you make it.** Work all day; name the moments that matter afterward (see below). A blank autosave isn't a mistake — it's the default.

Naming work — now, or later

Because saving and naming are separate, you have three ways to put a name on history, and you can mix them freely:

- **Name as you go:** `swarmfile commit -m "Grade pass 2"` files your recent saves under that message.
- **Name after the fact:** `swarmfile describe <N> -m "v2 delivery to client"` names *any* past commit — including an autosave that landed with no message. `swarmfile commit --amend -m "…"` does the same for the most recent one. Both change only the message.
- **Mark a milestone:** `swarmfile tag create <name>` pins an immutable, memorable name to a commit.

In the desktop app and the web dashboard, the same thing is a pencil/**Add description** on any commit row — no command needed.

The two commit modes (30-second version)

- **Mode A (default, autosave):** saves land as commits automatically; a burst of edits becomes one commit, the next burst another. You do nothing. This is what an office worker who's never heard of Git gets, and it's fine.
- **Mode B (explicit):** open a *changelist*, stage a set of saves, and submit them as one atomic unit — the analog of the Git index, for work that only makes sense landing together.

They're not a setting you choose — open a changelist and you're in Mode B for those saves; otherwise you're in Mode A. Full detail in Version Control.

What genuinely has no equivalent

- **`git rebase` / history rewriting** — Swarmfile history is an append-only server-side DAG of whole-file commits. It isn't rewritten. To put a name on the past, use `describe` or `tag`.
- **`git reset` (index-moving forms)** — `swarmfile reset --hard <rev>` does work, mapping to `restore <rev>`. But bare `reset`, `--soft` / `--mixed`, and `reset <path>` have no equivalent: saves sync immediately, so there's no unstaged state to move. To undo, land a `revert` (undo one commit) or a `restore` (roll a scope back to a point in time) — both move history *forward* rather than erasing it.
- **`git stash`** — actually maps onto Swarmfile's parked work: `swarmfile stash` runs `changelist park` (set staged work aside, kept on the hub), `stash pop` resumes it, `stash list` / `show` lists it, `stash drop` discards it. Parked work is a **set, not a stack** — `pop` restores *all* parked work for the branch, and a bare `drop` discards *all* of it, so it asks first; `stash list` numbers the entries and `stash drop stash@{1}` (or `drop 1`) discards just that one. In-progress *saves* are already auto-saved, so there's nothing lost either way.
- **File plumbing (`rm`, `mv`, `cp`, `mkdir`, `touch`) and `grep`**: these aren't CLI verbs. You do file operations on the mounted drive itself; for content search, work in a `git clone` of a git-enabled repository. `swarmfile-search` finds entries by name, not file contents (a public repository's latest release is the exception: its text files are searchable on the public page).

Type any of these (`swarmfile rebase`, `swarmfile rm`, `swarmfile worktree`, `swarmfile shortlog`, `swarmfile fsck`, …) and Swarmfile will print the one-line reason and the thing to do instead. (`swarmfile add`, `swarmfile stash`, and `swarmfile reset --hard <rev>` don't just explain — they run the mapped command.)

The on-ramp: bring the git you already use (git-LFS)

> For the side-by-side of every way git and Swarmfile meet — git-LFS, `git clone swarmfile://`, and the CLI's git-style commands — see Git and Swarmfile.

You don't have to move a whole team at once. Step one is to keep your source in the git you use today — GitHub/GitLab/self-hosted — and route only the large binary assets to Swarmfile, where Swarmfile acts as a **git-LFS server**: on a managed repository the big files get real storage, dedup, encryption at rest through the transfer agent, and quota instead of bloating your git host; on an unencrypted repository they are plaintext like the rest of it.

1. In the dashboard, open your repository → **git-LFS** tab → **Generate git-LFS credential**.
2. Commit the `.lfsconfig` it shows to your repo.
3. Install the transfer agent on each machine that pushes (`swarmfile-lfs install`; it ships with the Desktop App). A managed repository refuses stock direct LFS uploads, so the agent is required there.
4. Run the setup commands it shows once in your working copy (they store the credential and force basic auth).

After that, `git lfs push` / `pull` / `checkout` move big files through Swarmfile with no change to your VCS or workflow. git-LFS works on **managed-key** and **unencrypted** repositories; it isn't available on end-to-end encrypted repositories (a plain `git-lfs` client can't hold the per-user key).

A few things worth knowing up front (all covered in the Git-LFS guide):

- **Pushes need the small transfer agent on a managed repository** (it ships with the desktop app or installs standalone); on an unencrypted repository stock git-LFS still works up to ~5 GiB, and the agent lifts that to 256 GiB per object with chunked, resumable uploads. Pulls normally need no agent; a very large object's first stock pull may be paced while the hub prepares it (the client retries); the mount reads it either way.
- **Locking** — `git lfs lock` / `unlock` / `locks` work, including for files that live *only* in git and were never written to the drive.
- **No lock-in** — `git lfs fetch --all` pulls every object back with stock git-LFS and no Swarmfile software, so you can repoint `.lfsconfig` at any other LFS host and leave. Your source was never in Swarmfile to begin with.
- **Onto the drive** — run `swarmfile-lfs reflect` after a push to have those pushed objects appear as first-class files on the mounted Swarmfile drive (reusing the uploaded blocks, no second copy).
- **Encryption posture** — on a managed repository, uploads go through the transfer agent and are encrypted at rest, any size; stock direct pushes are refused with `lfs_agent_required`. An unencrypted repository is plaintext by definition. Objects a managed repository took before this rule are converted by an operator-run migration; until it completes, don't assume every older object is encrypted. The guide has the full table and the older-objects note.
- **GitHub, a little more automatic** — on a GitHub remote, the git-LFS tab can install the Swarmfile GitHub App, open the `.lfsconfig` pull request for you, and reflect each push onto the drive without a local `reflect` (reflection still needs the transfer agent — a stock `git-lfs push` stores the object but doesn't place it on the drive).

The bridge is where you start, not where you stop. The fuller destination is making Swarmfile the home for the *whole* repository: native version control — branch, merge, history and rollback — over a drive you mount and open files from, so nothing has to live behind a clone.

`swarmfile git import <url>` takes an existing GitHub/GitLab/local repository there in one command — the source's default branch becomes the repository's, commits keep their upstream ids (annotated tags keep their exact objects), and re-running it syncs new upstream commits into the same repository (fidelity limits in Known Limitations). Move at your own pace; the LFS on-ramp buys you the storage win today.

Moving from Perforce

If your team runs Perforce (Helix Core) today, most of Swarmfile will feel familiar: changelists, exclusive checkouts of binary files, stream-style branches, review before a change lands. The big difference is that there's no workspace to sync. Your repository is a **mounted drive**: files stream on demand, and a save is on its way to the team the moment you make it.

This page maps Perforce concepts onto Swarmfile, explains how to bring a depot across, and is direct about what doesn't carry over.

Concept map

Perforce	Swarmfile	Notes
Server / depot	Organization / **repository**	A repository is the unit of storage, history, and permissions.
Stream	**Branch**	Branches follow the streams model: a full working line over shared, content-addressed storage. Forking a branch copies no file data. See Branches & Merging.
Workspace (client spec) + `p4 sync`	**The mounted drive** — no sync	Files appear on demand and stay current as teammates save. There's no client view to maintain and no sync to run. `swarmfile switch <branch>` moves a mount to another branch in place.
`p4 edit` / `p4 add` / `p4 delete`	Just open, save, create, or delete on the drive	Every change is tracked automatically.
Pending changelist	**Changelist**	Open one with `swarmfile changelist open -m "…"` ; saves stage into it until you submit. You can hold several at once and choose which is current, like `p4 … -c` . See Version Control.
`p4 submit`	`swarmfile changelist submit` (or `swarmfile commit -m "…"`)	Applies atomically. Repositories can also run in the default auto-landed mode, where each save lands as it's made and you name commits when it suits you.
Submitted changelist number	**Commit** number (`#482`)	`swarmfile log` , `swarmfile show 482` , `swarmfile diff` .
`p4 shelve` / `p4 unshelve`	**Park** / **resume**	`swarmfile changelist park` sets aside every open changelist on the mount (kept on the server); `changelist resume` brings it back. See Setting work aside.
Exclusive checkout (`+l` filetype, `p4 lock`)	**Edit lock**	`swarmfile lock <path>` reserves a file for up to 30 days (7 by default); others can ask for it back with `swarmfile unlock-request create <path>` . See Entry locks.
Revit / SolidWorks exclusive open	**Worksharing**, no plugin	On Windows, the application's own exclusive open is enforced across machines. See Worksharing.
`p4 opened` / who has it	**Presence**	See who has a file open or locked, in the dashboard and the Desktop App.
Working disconnected	**Pack & Go**	Reserve a folder or repository for offline editing and land the changes when you're back. See Working Offline.
`p4 undo` / rollback	`swarmfile revert` , `swarmfile restore` , `swarmfile rollback`	Revert a commit, restore a file, folder, or repository to a past point — each as a new, undoable commit. See Trash, History & Rollback.
Labels	**Tags**	`swarmfile tag create v1.0` — an immutable name for a commit.
Protections table	**Folder and file ACLs**	Allow/deny read, write, or admin, inherited down the tree. See Permissions.
Helix Swarm review	**Merge requests**	Review on a branch before it lands, with approvals, requested reviewers, and diffs. **Protected branches** refuse changes that haven't come through an approved merge request, and protection rules can request reviewers automatically (CODEOWNERS-style).
Triggers	**Webhooks** and the **runner**	Signed webhooks on commits, merges, and more; the runner runs jobs on a branch when a commit or tag lands.
Proxy / edge server	**LAN-first peers** and **seed nodes**	Machines in the same office share data directly; a seed node on your own hardware keeps a warm local copy.

Bringing a depot across

There's no Perforce history importer. Swarmfile can't read a Perforce server's revision history and replay it as commits. What you can bring across is the **current state** of a depot or stream, and that's usually what a team needs to start working.

1. **Get the head revision on disk.** In a Perforce workspace that maps the depot or stream you're moving, `p4 sync` to the revision you want to start from.
2. **Create the Swarmfile repository**, and decide its branches (one per stream you're moving, usually starting with `main`).
3. **Import with `swarmfile-migrate` .** Dry-run first to check what would be imported and that your excludes are right:

```
swarmfile-migrate --source /p4/workspaces/game-main \
  --org-id acme-games --repository-id game \
  --exclude ".p4config" --exclude "**/.p4ignore" \
  --dry-run

swarmfile-migrate --source /p4/workspaces/game-main \
  --org-id acme-games --repository-id game \
  --exclude ".p4config" --exclude "**/.p4ignore" \
  --state-db ./migrate-state.db --report-json ./migrate-report.json
```

`--state-db` makes a large import resumable, each file is checked against the source after upload (on by default; `--verify=false` skips it), and `--report-json` gives you a record of everything that arrived. The `swarmfile-migrate` reference has every flag, and Migrating Existing Data walks through a full run.

4. **Commit and tag the import** (for example `swarmfile tag create imported-from-p4`) so there's a named starting point.
5. **Recreate permissions and exclusive-checkout habits**: ACLs for restricted folders, and edit locks (or Windows worksharing) for the binary files your team used to `+l`.
6. **Keep Perforce read-only for older history.** Leave the server up in read-only mode (or archive it) for as long as you need to look up who changed what before the move. Swarmfile's history starts at the import.

Moving one stream at a time works fine; Swarmfile doesn't need to own the whole depot on day one.

Where it's genuinely different

- **No sync, no have-list.** You never decide which revision your workspace holds; the drive shows the branch as it is. Large files you haven't opened take no disk space until you do. To keep a set of files local for the road, use Pack & Go.
- **Saves are shared immediately by default.** Perforce shares nothing until submit. Swarmfile's default mode lands each save as it's made; switch a repository to staged mode, or open a changelist, when you want Perforce-style batching.
- **History moves forward.** Reverts, restores, and rollbacks are recorded as new commits rather than edits to old ones, and `commit --amend` changes only a message.
- **Whole-file versions.** Like Perforce for binaries, every version is a whole file (stored deduplicated, so unchanged parts of a large file cost nothing to keep).

Not supported

- **No Perforce history importer.** Import the head revision as above and keep the old server for history.
- **No game-engine source-control plugins.** There's no Unreal Editor or Unity source-control provider for Swarmfile, so the editors' built-in check-out and submit buttons don't talk to it. Teams work on the mounted drive directly and use the Desktop App or the `swarmfile` CLI for locks and commits.
- **No Perforce-compatible commands.** The `swarmfile` CLI uses its own verbs (and understands many git habits, see Coming from Git); it doesn't accept `p4` command lines.

If one of these is a blocker for your move, tell us — it helps us decide what to build next.

Migrating Existing Data

`swarmfile-migrate` is a standalone CLI for bulk-importing data you already have into a Swarmfile repository — a NAS share, a flat list of file paths, or an S3-compatible bucket. Use it the first time you bring an existing archive onto Swarmfile, rather than dragging everything through your mounted drive by hand: it talks to the hub directly rather than through a mount, uploads files concurrently, and resumes from a local state database if the run is interrupted.

Bringing a **git repository** rather than a file tree? Use `swarmfile git import` instead — it carries the commit history, branches and tags. `swarmfile-migrate` moves files, not version-control history.

Looking for something narrower? `swarmfile-seed` places a single file at a repository path, and `swarmfile-mirror` copies a whole repository from one hub to another — both run the same pipeline with no engine.

Figure omitted in print (animation): Importing an existing NAS library: one command reads the share, verifies every file by content, and the library appears as an ordinary drive on another machine. - https://swarmfile.com/demo/nas-migration.cast

No Desktop App on the machine — a build server, a container, a machine you don't administer? The migrator is also published as a standalone open-source build (API-key auth, no OIDC) at swarmfile-migrate, installed with `curl -fsSL https://get.swarmfile.com/migrate | sh`. It runs the same upload pipeline and verification, but it is a separate tool with its own flags, so the examples on this page are written for the desktop build:

	Desktop build (installed with the app)	**Standalone build**
Target	`--hub-url` and `--repository-id` (falling back to `SWARMFILE_HUB_URL` / `SWARMFILE_REPOSITORY_ID`), `--org-id`	`--hub-url`, `--org-id`, `--repository-id`, all required
Credential	`SWARMFILE_API_KEY` from the environment, or the `--oidc-*` flags	`--token` (defaults to `SWARMFILE_API_KEY`)
Resume state	`--state-db`, default `<source>/.swarmfile-migration/migrate.db`	`--state-path`, default `./.swarmfile-migrate-<repository-id>.sqlite`
Only in this build	`--dry-run`, `--report-json`, `--exclude-from`	`--ec` (erasure coding; the desktop build turns it on with `SWARMFILE_EC_ENABLED=on` instead)

Both builds share `--source`, `--source-list`, the S3 source flags, `--exclude`, `--retry`, `--verify`, `--force` and `--start-at`. The state file format is the same, so to resume a run in the other build, pass the state file explicitly (`--state-db` on one side, `--state-path` on the other).

This page is a practical walkthrough. `swarmfile-migrate` is installed on `PATH` with the desktop app on every platform. For the complete flag reference, see swarmfile-migrate.

A realistic example

Importing a local repository directory, skipping cache and scratch files:

```
swarmfile-migrate --source /Volumes/nas/RepositoryArchive \
  --hub-url https://hub.swarmfile.com \
  --org-id acme-films --repository-id feature-01 \
  --exclude "*.cache" --exclude "**/tmp/**"
```

This walks the source directory, applies the gitignore-style exclude patterns to skip anything you don't want copied, and uploads the rest into the target repository.

What to expect while it runs

Migrate reports progress as it goes and retries failed transfers automatically, with backoff, up to 3 attempts by default (`--retry`) — a dropped connection partway through a large file doesn't fail the whole run, it retries that transfer. If the process itself is interrupted (you kill it, the machine sleeps, the network drops entirely), it's safe to just run the same command again — a file that had exhausted its retries in the earlier run is still eligible on the next invocation.

Resume

Migrate keeps a local state database tracking what's already been uploaded. On a second run with the same source and target, it picks up where it left off instead of re-uploading everything from scratch. You don't need a separate `--resume` step or flag — re-running the same invocation is the resume mechanism.

Verification

After each file uploads, migrate checks its content against the source — a CID/hash comparison — so a successful run means the data landed intact, not just that bytes were sent. This is on by default; pass `--verify=false` to skip it if you trust the transport and want to roughly halve I/O on a very large migration. Migrate can also produce a JSON report of the run, which is the artifact to keep if you need to audit exactly what was migrated, skipped, or retried.

Figure omitted in print (animation): The standalone swarmfile-migrate imports an existing tree: the first pass is killed mid-flight, the same command again resumes exactly where the state file stopped — only the files that never landed move — then every file is re-chunked and checked against its recorded manifest. - https://swarmfile.com/demo/oss-migrate.cast

Deployment Topologies

Swarmfile is the same product in every deployment below. What changes is where the bytes travel — and that is a setting, not a different install or a migration. You can start central, move to LAN-first when a second person in the same office starts complaining about the same download, and be back again the same day if your security team asks.

Most topologies below are shown as a real recording. A few have nothing to *show* in a terminal — a second site, a headless seed node serving an office — so those are drawn as a diagram instead of faked as a recording.

One thing is true of every topology here, so it is worth watching first: opening a file never downloads the whole file — only the bytes your app reads plus the bounded read-ahead window described in Performance and Disk Usage. See Working with Files for that recording.

Central: hub and spoke

The shape most teams already run. One central store; every machine talks to it and never to each other.

Figure omitted in print (animation): A central share: a file saved on one machine appears on another and both fetched it from the central store — zero machine-to-machine transfers. - https://swarmfile.com/demo/hub-and-spoke.cast

Choose this when machine-to-machine traffic on the corporate network is disallowed, when your sites have no meaningful local network between them, or simply because it is the model everyone already understands. It works, and the drive behaves identically.

The cost is at the end of that recording: **zero blocks received from other machines**. Ten editors opening the same 200 GB sequence means ten downloads of it.

LAN-first: one download per office

The same repository, with machines allowed to serve each other.

Figure omitted in print (animation): Two machines sharing one drive: a file saved on one appears on the other, and the peer-transfer count proves the bytes came over the LAN. A machine is taken offline and the file stays readable. - https://swarmfile.com/demo/lan-resilience.cast

The first person to open a file pulls it from the cloud. Everyone else in that office gets it from them, over the local network, at local-network speed. The recording ends with the same counter the central one does — this time 12 blocks.

The second half matters just as much: a machine is taken **offline** and the file is still readable from the others. Erasure coding and peer replication mean the office does not depend on whoever happened to create a file still being at their desk.

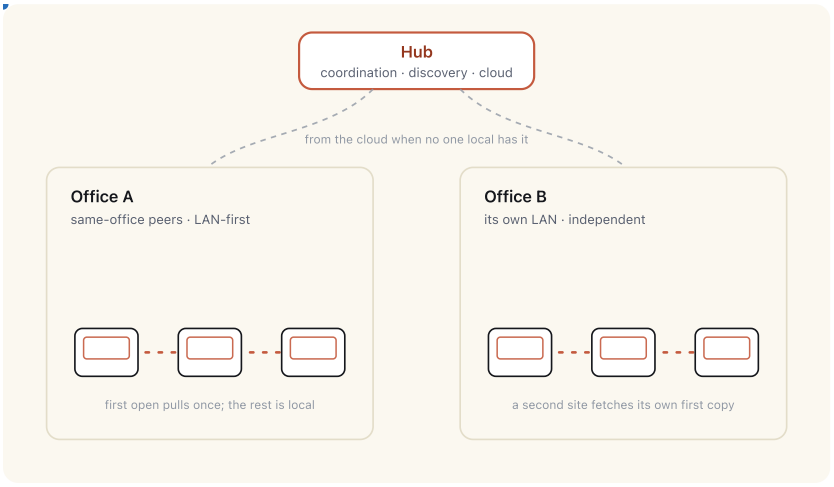
LAN-first also applies *before* a file reaches the cloud at all. When someone asks for part of a file that is still uploading — `swarmfile fetch`, or a follow job tracking a viewer — the uploading machine hands those blocks straight off its own disk to a peer in the same office, rather than everyone waiting for them to climb the office uplink and come back down from storage. That needs the repository's blocks to be peer-servable (an encrypted repository, or a public one); blocks that don't qualify wait for the cloud.

Serving a peer straight off the uploader's disk also only happens on a LAN, deliberately: over the internet it would put every block on the same congested uplink twice, once to the peer and once to the cloud, which is slower for everybody. It is also the reason this workflow is far more comfortable in one building than across two — on a LAN the constraint is the uploader's own disk, not its uplink.

Choose this when several people work in the same building on the same material. It is the default for a reason.

Multiple sites

A studio with people in two buildings, or two cities, tells each machine which office it is in. Machines then **prefer** peers in their own office: same-site peers are offered first, then any always-on seed nodes, then whatever else is around.



Preference rather than a hard boundary is the useful behavior. A colleague two desks away is the obvious place to get a file from, and that is what happens in practice.

But if your site is quiet — one person in early, everyone else still asleep in another timezone — a machine is not forbidden from using a distant peer that is **directly reachable**; a peer reachable only through the public relay is skipped by default and the read uses the cloud (`SWARMFILE_ALLOW_RELAY_FETCH=1` opts in on both sides,

throttled). It falls back to the cloud when nothing better is available. You get locality where locality exists, without a rule that strands someone working alone.

Nothing about this needs configuring per repository. Sites differ; the repository, its history and its permissions do not. Add a second person at a quiet site and that site starts sharing locally on its own.

One person, working remotely

A single editor at home has no peers by definition, and the drive behaves like the central topology above — everything comes from the cloud.

Which raises the obvious question: what happens when the cloud isn't there? A file you've been working with stays fully readable, because it's already on your machine — and a file you never opened fails at once with a clear answer rather than hanging your application, because there's no connection to fetch it over. When the connection returns, everything is simply there again.

Figure omitted in print (animation): The connection to the coordination service drops: a file already opened still reads in full from the local machine, while one never downloaded fails immediately instead of hanging; reconnecting restores everything. - https://swarmfile.com/demo/offline-resilience.cast

One thing does change underneath: erasure coding exists to mask WAN latency, and it switches itself off once a machine can see two or more peers on its local network. A solo remote machine keeps it on and a busy office machine does not, without anyone choosing. If you are evaluating Swarmfile with one person on a home connection, that is the configuration you are testing.

Seeding from what you already have

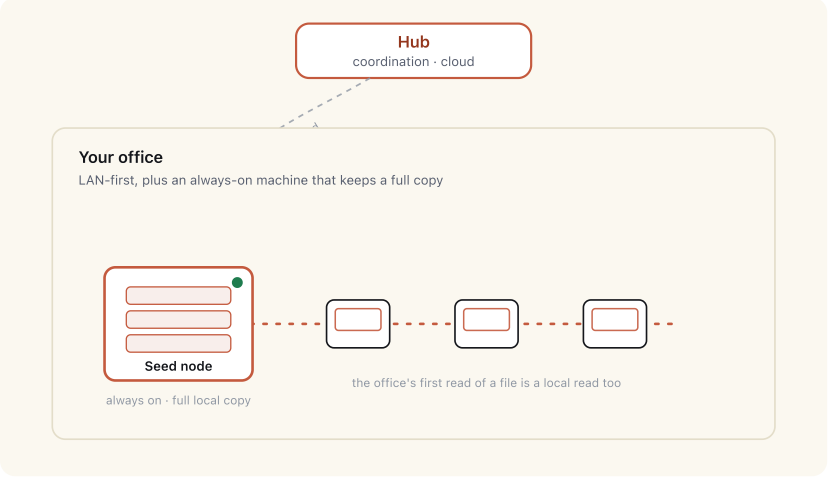
Neither topology is any use if getting your existing library in means a weekend of copying.

Figure omitted in print (animation): Importing an existing NAS library: one command reads the share, verifies every file by content, and the library appears as an ordinary drive on another machine. - https://swarmfile.com/demo/nas-migration.cast

The import reads your NAS and never writes to it, so you can point it at a live share. It is resumable — interrupt it and run it again — and every file is verified by content, not by size and timestamp. Full detail in Migrating Existing Data.

Keeping the NAS as a warm tier

Importing does not mean retiring the hardware. A machine you own — the NAS, an old workstation, a dedicated box — can join permanently as a **seed node**: it fetches the whole repository, mirrors every block to cloud storage, and keeps a warm local copy of the working set to serve everyone in the office over the LAN.



The difference from an ordinary machine is that a seed node is always there. LAN-first already means the first person to open a file supplies everyone else, but that depends on that person being at their desk with the machine awake. A seed node removes the dependency: its cache always holds the recent working set, and older blocks re-fetch on demand (the whole repository is mirrored to cloud storage, so nothing is ever unavailable).

Seed nodes are a **Pro plan feature**. This is enforced rather than advertised — a seed node on a plan without the entitlement is refused by the service, and `swarmfile doctor` reports it plainly rather than leaving a repeating warning in a log nobody reads. Setup, sizing and hardware guidance is in Self-Hosted Seed Nodes.

The same shape works in a data center rather than an office, for a team with no single main site: an always-on warm tier that belongs to nobody in particular.

Larger offices: deciding where the shards live

In an office of a handful of machines, whoever opens a file ends up holding it, and that is enough. Past a certain size, leaving your fault tolerance to whoever happened to open what is not a plan.

Figure omitted in print (animation): Deliberate shard placement in a five-machine office: turning it on assigns each piece of a file across three of the five machines, and swarmfile shards names exactly which — a decision, not chance. - https://swarmfile.com/demo/shard-placement.cast

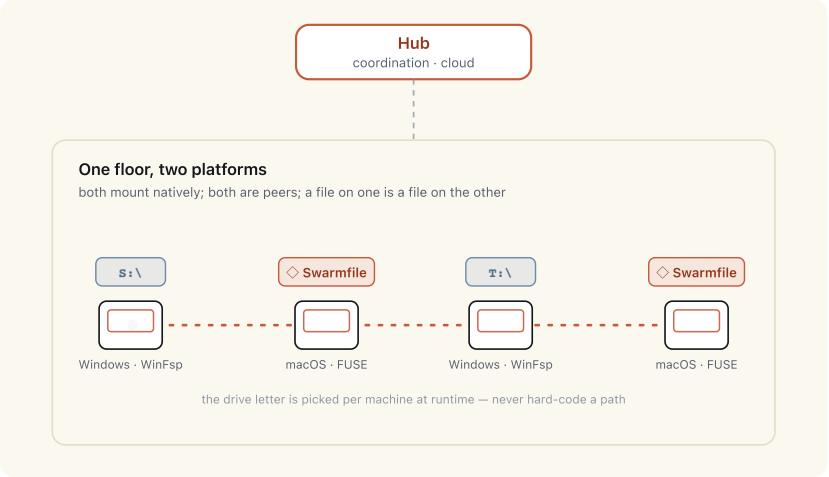
Turning on deliberate shard placement makes it a decision instead: each piece of a file is assigned across the office's machines at a replication factor of three, so any given file survives losing machines rather than surviving because someone happened to have opened it. `swarmfile shards <path>` shows the live assignment for any file — which machines should hold each piece, and which ones actually do.

Worth turning on when an office is large enough that "who has a copy?" stops being a question you can answer by asking around.

One caveat on corporate networks: deliberate shard placement needs a roster of LAN peers to spread across, and LAN peers are normally discovered by mDNS multicast. On segmented networks where mDNS multicast is blocked, set `lan_from_office` (`SWARMFILE_LAN_FROM_OFFICE`, or the config-file key — see Engine Config File) so same-office peers count as LAN. Both the `office_id` label and this flag apply live via `swarmfile office set <office_id> [--lan-from-office]` within one peer-discovery tick (~60s, no restart) — unlike enabling shard placement itself, which needs a restart.

Mixed Windows and macOS offices

A normal post or design floor is not one platform, and Swarmfile does not ask it to be. Windows machines mount through WinFsp and get a drive letter; macOS mounts through FUSE and gets a volume. Both are ordinary local paths to the applications on top, both are peers to each other, and a file written on one is a file on the other.



One practical consequence worth knowing up front: **the drive's location is decided per machine, at runtime**. On Windows, a taken drive letter is only silently relocated to a free one when it was never an explicit choice — still sitting on the app's own default or a letter it auto-picked — and a free candidate actually exists.

If you (or a script) explicitly set the letter, Swarmfile honors that choice and reports the conflict by naming the occupant instead of quietly moving you off it; likewise if every candidate letter is also taken, the mount fails and names the occupant rather than relocating. So no script, template or onboarding document should hard-code a path — read it from the Desktop App's Status pill, or from the app, which knows.

Render farms and headless machines

Not every machine that needs the repository has somebody sitting at it. Render nodes, watch folders, transcode boxes and QC stations are usually many readers and few writers, which is the shape LAN-first is best at: the first node to pull a frame sequence supplies the rest of the farm locally instead of every node fetching it independently.

Two things make this work well in practice. Mount the farm **read-only** where it only needs to read, so an automation cannot write into the repository by accident. And give the farm a **seed node** — a machine that keeps the repository warm and mirrored — so the first read of the day is a local read too, rather than depending on whichever workstation happens to be awake.

If the farm itself is cloud-hosted — a cluster of render nodes in your own cloud account rather than machines in a physical office — there's no single LAN to put that one seed node on. The farm can still get redundancy among its own nodes instead of leaning on a single point of failure: enable seeding and set the same `office_id` on each node (no shared network required — see Self-Hosted Seed Nodes).

Each seed node independently fetches every block and mirrors the full repository to cloud storage, so the repository's cloud copy is complete on its own — a stronger guarantee than deliberate shard placement's replication factor of 3 — and there's no extra step needed on top of it for an all-seed-node farm.

The other thing an unattended machine usually needs is to be pinned to an *exact* version, not "whatever main happens to be right now" — a render should use the frames as they were when the job was submitted, even if someone pushes a fix five minutes later. Tag the commit a job was submitted against, then have each node branch from that tag: one command, no Desktop App or browser interaction, and the node's copy is frozen to precisely that content regardless of what lands on main afterward.

Figure omitted in print (animation): A headless render node signs in with an API key, tags the submitted version, forks a branch pinned to that tag, and hydrates the scope resident before the job. An artist then commits a new scene to main — and the render node, on its pinned branch, still shows only the frames that were submitted. - https://swarmfile.com/demo/render-farm.cast

```
swarmfile commit -m "submit render job 482"
swarmfile tag create render-482
swarmfile branch create farm-482 --from-tag render-482
swarmfile branch switch farm-482
```

`tag create` with no `--at-seq` tags the branch's current head — exactly the commit the submission script just made, with no separate step to look up its commit number.

Mounting sparse is right for interactive work, but a render node has no interactivity to hide latency behind: the first frame it opens still has to fetch before the job can proceed, and if that node has no LAN peer or seed node nearby, every frame is a cloud round trip on the critical path. `swarmfile hydrate start` forces a scope fully resident up front instead — download and pin every file under a path (or the whole repository as this mount's branch sees it) before the job starts, rather than fetching lazily as frames are opened:

```
swarmfile hydrate start ./Repositories/shots/seqA --yes
```

Hydrated files are pinned against the cache's normal eviction, so they stay resident until you explicitly `swarmfile hydrate release` them (or delete the local copies outright with `swarmfile hydrate free-space` once the job is done) — useful for a node that needs to keep working with no network at all for the length of a job. Pair it with the branch-from-tag pin above: hydrate the exact scope a job was submitted against, and the node has everything it needs, locally, before a single frame renders.

A farm node authenticates with a repository-scoped **API key**, not a human session. Mint one from the web dashboard, the Desktop App, or the CLI — an admin or owner picks a name and a single repository, and gets back a secret shown exactly once:

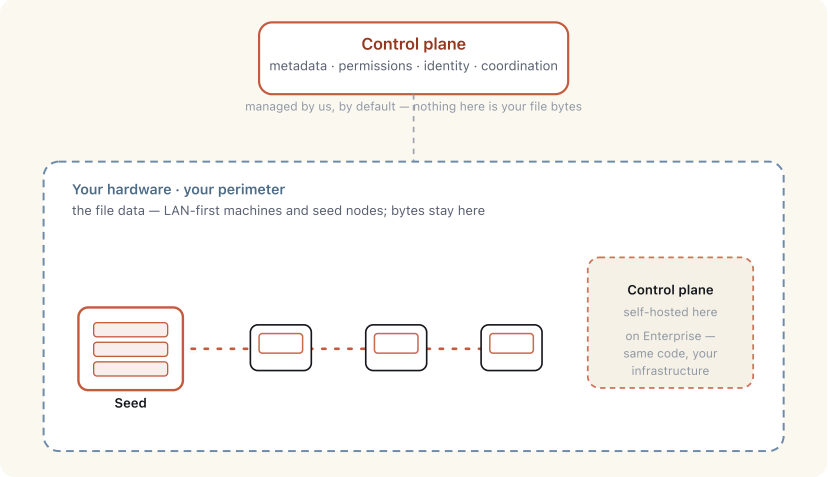
```
swarmfile api-key create farm-node-12 --repository proj_9f2a
```

Set it as `SWARMFILE_API_KEY` on the node (alongside `SWARMFILE_CONFIG` pointing at a per-repository config file) and the engine skips sign-in entirely — no browser, no refresh token, no impersonating whoever happened to mint it. The key is walled off to that one repository: it can't read or write anything else in the org, even repositories the admin who minted it can see, and revoking it (`swarmfile api-key revoke <id>`, or from the dashboard/Desktop App) kills that one node immediately without touching anyone's own session. See CLI: swarmfile for the full `branch` / `tag` / `hydrate` / `api-key` reference.

For the coding-agent variant of this shape — several agents, each with its own labeled mount and branch, sharing one dependency cache — see Coding Agents.

What runs where — and what you can host yourself

Worth being direct about, because it comes up in every security review.



On your hardware: the file data. Dedicated storage is the default for new paid orgs and available to any paid org still on shared storage (existing files are copied into it in the background) — an isolated bucket of their own rather than a shared one — and on Enterprise bring-your-own-storage can make an S3-compatible bucket on your own account the org's primary block storage.

Seed nodes keep the repository warm locally (their cache budget bounds the disk; every block is mirrored to your storage), and LAN-first means bytes move machine-to-machine without a round trip to anyone's cloud. A studio with a seed node in the building serves most reads from its own hardware.

Where it lives: every paid plan can pin an organization's metadata and block storage to a real EU or US jurisdiction at signup, at no extra cost (Free uses shared storage and isn't eligible), and an individual repository can carry its own residency pin independent of the org's. Enterprise can additionally provision FedRAMP / FedRAMP-High storage regions through sales (storage only — Swarmfile is not FedRAMP-authorized).

Managed by us, by default: the control plane — metadata, permissions, identity, and the coordination that lets machines find each other. Most customers run this as a hosted service.

Self-hostable on Enterprise: the control plane can also run entirely on infrastructure you control, on a self-hostable runtime compatible with the platform it's built on — the same code, not a fork or a lesser rewrite. This is available on Enterprise today; talk to us about what it would involve for your environment. A fully air-gapped deployment, with no outside connectivity at all, is not available today.

So Swarmfile suits an organization that wants its **bulk data** on infrastructure it controls, even on the standard hosted plan — and if you need the **coordination layer** inside your own perimeter too, that's an Enterprise option rather than something the rest of this page's topologies can get you. Raise it early in a procurement conversation so it gets scoped correctly.

Encryption is the thing worth reading next if this section is what you came for: per-repository keys, with an end-to-end tier where the service cannot read your content at all. See Security.

Guests, clients and consultants

Not everyone who needs a file needs a machine in your topology. External collaborators get a named, revocable read — or read-and-comment — view of one folder, without consuming a paid seat (private-repository guests count against the plan allowance and can bill overage; public-repository guests are free) and with a full audit trail — a colorist, a client reviewing a cut, a consulting engineer. Covered in Sharing & Collaboration.

Choosing

If this is true	Use
Machine-to-machine traffic is not permitted on your network	Central, enforced org-wide
Your people are spread across homes and cities, rarely co-located	Central
Several people share a building and open the same large files	LAN-first
You have multiple offices	LAN-first at each; they are independent
You have an existing NAS you want to keep using	LAN-first plus a seed node (Pro)
Nobody has a main office, but you want a warm copy somewhere	A seed node in a datacenter (Pro)
An office is big enough that "who has a copy?" is unanswerable	LAN-first plus deliberate shard placement
Someone outside the company needs to see one folder	An external collaborator, not a topology change
Your floor is a mix of Windows and macOS	Either topology; both mount natively
Many machines read, few write (a render farm)	LAN-first, mounted read-only, plus a seed node
Everything including the control plane must be inside your perimeter	Enterprise self-hosted control plane (fully air-gapped is not available today) — see "What runs where" above

Switching is a setting, not a migration: the files, history, and permissions are unaffected. Try one, measure it, change your mind.

Two things worth knowing about who holds that setting. Each machine can choose for itself — but an organization can also set its data-plane policy to **hub-only for everyone**, and that decision overrides the local setting on every machine, whatever it is set to. If your security position is "no machine-to-machine traffic on our network", that is enforceable centrally rather than left to each person to configure correctly.

The reverse is not true: an org that permits LAN-first does not compel it. A machine on a network where peer traffic is blocked simply finds no peers and falls back to the cloud, which is the central topology arriving at itself.

Working with Files

Your mounted drive is a normal filesystem to any application — Resolve, Premiere, Revit, QGIS, Explorer, Finder, whatever you point at it. This page covers what's different about working with files on Swarmfile: the web dashboard's view into the same data, and the mechanics of saving, presence, and locking that run underneath.

Browsing

The Files view

The web dashboard's **Files** view is a file explorer. Browse the repository's folders and files in the left pane — each row shows its size and how recently it changed — and open one to see an inline preview (images, PDFs, code, Markdown, and generated thumbnails for other types) with its metadata alongside. It is a separate view onto the same storage as your mounted drive, useful for checking history or activity without opening a native app.

Browsing the drive and app

On the mount, browsing is normal folder navigation and costs no disk space: listing a directory does not download anything in it. In the Desktop App:

- The left-hand pane lists your repositories and drives, grouped **Pinned**, **Recent**, **Mounted** and **All**. Type in the filter (or press `/`) to narrow a long list; Cmd/Ctrl+K jumps to a repository or page.
- Each mounted repository expands into a **folder tree**; pick a folder to jump the file list to it. A teammate's new, renamed or deleted folder appears as the change syncs.
- The activity bar's pages (Branches, History, Tags, Releases, Trash, Merge requests, Locks, CI runs, Data sources and Repository settings) open in the canvas under the same repository/drive/branch header; Back returns to the previous page.

Moving things around works by drag-and-drop, with a menu fallback: in the web dashboard drag a row onto a folder, the repository root, or a breadcrumb to move it — dropping on the file area itself moves it into the folder you're viewing — or use the row's right-click menu → **Move...** to pick a destination folder. In the Desktop App drag a file from the file list onto a folder in the left tree, or onto a drive's own row to drop it at that drive's top level. A move keeps the entry's identity, so its history, comments and locks follow it; moving a folder into its own subtree is refused.

When someone else saves a file, it appears in your file manager within a couple of seconds. Your operating system caches directory listings for a short window, so a folder you already have open can be a moment behind; reopening it or pressing refresh always shows the current state.

A window that already has the file open keeps the version it opened — each open handle is served one consistent version, so a teammate's save never swaps bytes under a running app. Reopen the file to see the new version; the Desktop App posts a one-time notice ("... was updated — reopen it to see the new version") when one of your open files has been superseded. Fresh opens and the web dashboard see the new version immediately.

A single folder holding tens or hundreds of thousands of files — a point-cloud tile set, a scan-output dump — is loaded on demand: the web dashboard's Files view fetches a folder's contents when you expand it and renders at most a couple of hundred rows at a time (with a "Show all" reveal), and the Desktop App's file picker only renders the rows currently in view, so both stay responsive on a folder that size instead of the browser or app choking on it.

The web dashboard additionally renders thumbnails and inline previews — including video posters, scrubbable proxies, and point-cloud previews — so you can eyeball a folder without opening a native app. See File Previews.

Opening and saving

Opening a file streams the bytes your application reads — and only those bytes. Those bytes come from the nearest place that has them: the local cache first, then a teammate or another of your machines on the same LAN, then a self-hosted seed node if your org runs one, then peers elsewhere on the swarm, and Swarmfile cloud storage last. WAN-heavy reads can also use adaptive Reed-Solomon erasure coding (10+4) so a few slow peers don't stall the transfer.

Figure omitted in print (animation): A machine lists a 64 MB file it has never fetched and its local cache does not grow at all; it then reads one megabyte from the middle and the cache grows by a few megabytes, not sixty-four. - https://swarmfile.com/demo/sparse-streaming.cast

The recording above is the whole idea in three commands: listing a file costs nothing, and reading part of one costs that part. Scale it up and it is why a 200 GB sequence opens on a laptop with 40 GB free. Saving is asynchronous: a write returns as soon as it's queued, not once it's fully uploaded. A durable local upload queue handles the actual transfer in the background, with automatic retry on failure.

Figure omitted in print (animation): A 48 MB save returns at local-disk speed while dozens of blocks are still queued for upload, and the queue then drains to zero on its own. - https://swarmfile.com/demo/zero-stall-writes.cast

That recording is the same point from the writing side: the save returns at local-disk speed with dozens of blocks still queued behind it. Because the queue is durable, it survives a restart or a dropped network connection — pending uploads resume automatically rather than being lost. You are never blocked waiting for a save to finish uploading before you can keep working.

Reading the sync status

The Desktop App's header keeps one line of truth about that queue. While work is in flight it says **Syncing N** — N counts the work still on its way to the hub, **one per save** (a 40 GB save counts once, not once per block): saves whose bytes are still uploading, new files whose creation hasn't been confirmed yet, and saves whose bytes are already up but whose commit hasn't landed.

Synced means the queue is empty, not merely that a transfer looks quiet: a save whose commit is still being retried keeps the header at *Syncing* until it lands. A busy hub or a sign-in that has lapsed only pauses the queue; neither uses up a save's retries, so a save doesn't give up while you sign back in.

Three of those states are worth knowing by name:

- **Waiting on plan** — everything left is a save the hub is refusing on plan grounds, almost always storage (the org's allowance is full) or the org's spend cap. The retries are automatic, but nothing moves until the limit is raised; a quota/billing-block notification tells you which limit, and the plan panel shows the allowance. An owner or admin raises the spend cap under **Billing → Spend cap**.

- **N changes not syncing** — the hub keeps refusing these saves for another reason, so retrying isn't making progress. They're safe on this computer and Swarmfile keeps trying; the attention indicator points to the **Status pill** in the header, which opens the panel showing the hub's reason. If the hub refused a save because part of its content never arrived, Swarmfile re-sends that content from this computer on its own before asking you for anything. The Status panel lists each file that is still stuck — why, and for how long — with **Discard this change...**, which drops the unsynced change and puts the file back to its cloud version. That can't be undone, so it asks first. From a terminal: `swarmfile sync stuck` and `swarmfile sync discard <entry-id>`.
- **N changes failed** — some work stopped retrying on its own. The header's attention indicator names each one, and the banner above the file list offers **Retry**; an upload that failed can also simply be saved again. On Windows, saving the file again also clears the error badge Explorer shows on it.

The header counts the repository you're looking at. Saves you made in another repository keep uploading in the background after you switch away from it, and the Status pill adds **N saving in other repositories** (or names the repository when only one has saves left) until they land.

This repository can read **Synced** while that's true. A change in another repository that the hub is refusing is named there too, and stays on this computer until it's resolved in that repository. `swarmfile status` prints the same: its pending uploads line adds "(N more in other repositories)", followed by one line per repository with its saves still uploading or held.

Pending creates

A file you create is listed in the Desktop App immediately, before the hub confirms it. The row reads **Syncing — this one isn't on the hub yet** until the create lands (seconds on a healthy connection), and you can already rename or delete it from the app — applied on this computer, not waiting for the hub.

- A create the hub refuses but *holds* (root-create authority on a protected repository, for example) names the reason on the row instead and keeps retrying; see Permissions.
- A teammate's newly created file shows **Waiting for the hub** until confirmed; it can still be retracted by its creator.
- Moving or copying a file whose create has not landed, or moving/copying/uploading/importing anything into a not-yet-confirmed folder, waits for the hub.

While a large file is still uploading

The queue draining in the background is invisible on a fast link. On a slow one it isn't: a very large file on a modest uplink can take hours or days to finish. Until it does, everyone else keeps reading the last committed version — the in-flight save never replaces it. (A file's *first* upload is the exception: it can be opened while it is still arriving — see Opening an in-progress upload directly.)

While that transfer is running, the file's page in the web dashboard shows an **Uploading** badge, along with which machine is sending it. The row itself still describes the version everyone can currently read — same size, same date, and Download still works — because a version in progress never replaces the one people are already using. Nothing becomes unavailable because a colleague started saving.

If you need part of that new version before it has all arrived, you can ask for it, and the uploading machine will send that part first:

```
swarmfile fetch ./Repositories/reel-04.r3d --tail 209715200
```

That is the last 200 MB of a file still in transit. The request travels to whichever machine is uploading, that machine moves the blocks covering your range to the front of its queue, and you get them in roughly the time it takes to send *that range* — rather than waiting out the whole file. Uploads are sent in order from the start of the file, so without asking, the end is the last thing to arrive.

It is worth knowing what this does and doesn't do. It changes **which bytes arrive first**, not how fast the link is: the last 20% of a 5 TB file is still 1 TB, and 1 TB still takes as long as 1 TB takes. What it removes is the wait for the other 4 TB you didn't need.

The bytes land in your local cache, pinned so they aren't evicted before you open the file. Release them with `swarmfile hydrate release` when you're done — see the CLI reference — or delete them outright with Free up space. Reading the file normally through the mount is unaffected and gives you the committed version — unless the mount has been opted into streaming, below.

Watching a file as it uploads

A single range is the right shape for "give me the last 200 MB". It is the wrong shape for watching, because a viewer moves: it plays forward, and it seeks. `--follow` tracks that.

```
# Start following from the beginning
swarmfile fetch ./Repositories/reel-04.r3d --follow --version pending

# …and tell it where the viewer has got to
swarmfile fetch-seek 5000000
swarmfile fetch-status
```

Instead of completing at a fixed range, a follow job keeps a buffer ahead of wherever the reader is and re-targets when it moves. The uploading machine is told about the new position immediately, and — this is the part that matters — the new request **replaces** the old one rather than queueing behind it, so it stops working on the stretch you scrubbed away from. A follow job ends on its own when it reaches the end of the file, or after five minutes with no `fetch-seek` — a status that stops advancing means it finished, not that it broke.

How far ahead it buffers is measured, not configured: the read position's own rate of advance *is* the consumption rate, so a 6 Mbit/s proxy gets a small window and a 200 Mbit/s master gets a proportionally larger one, both covering the same number of seconds. `SWARMFILE_STREAM_BUFFER_SECS` sets that number (default 30). A seek is not mistaken for playback.

Opening an in-progress upload directly

Everything above puts bytes in the cache. There is also a mode where a file that has never finished uploading is simply **openable** — it appears at its full eventual size and reads work, waiting briefly where the bytes have not landed yet.

This is always on. A read into a range that hasn't arrived has to wait, and a filesystem operation that waits too long doesn't stall one read — it takes the whole drive with it. The wait here is strictly bounded (15 seconds, see `SWARMFILE_STREAM_READ_WAIT_SECS`) and a read that outruns the upload returns "try again" rather than an I/O error, precisely so an application doesn't conclude the file is damaged. See the config reference for the full behavior.

Two things to know about the scope:

- It applies only to files with **no** committed version yet — a new master landing for the first time. A file you are already reading never changes size or content underneath you because somebody started a new save. That is the same rule the rest of the product follows, and it matters more than the feature does.
- If the uploading machine is on your **LAN**, it will serve those blocks to you directly, from its own disk, before they have reached the cloud at all. Across the internet it won't: that would put every block on the same congested uplink twice. Direct serving also requires the repository to be **encrypted or public** — a *private* repository's plaintext blocks are never handed out peer-to-peer; a public repository's cleartext blocks are already anonymous-readable, so they are. Where blocks aren't peer-servable, in-flight reads wait for the cloud like any other.

The real limit. None of this makes the link faster. It makes the right bytes arrive first. A 4 TB file on a 100 Mbit/s uplink delivers about 12 MB/s however well it is ordered — comfortable for a proxy, not for a full-resolution master. The reliable version of this workflow is "watch the proxy while the master uploads", and on a LAN, where the uploader's own disk is the limit rather than its uplink, considerably more than that.

Freeing up space

Everything you open stays in a local cache so the next read is instant, and the cache trims itself when it reaches its size limit. The *first* read of a cloud-only file is the one exception: the engine fetches on demand and reads ahead as playback advances, so a large media file can take a few seconds to settle while it builds that lead — and it only ever pulls the byte ranges you touch, never the whole file. Once those bytes are cached, the file is local-fast like any other.

To make room on purpose — or to hand a laptop's disk back before traveling — use **Free up space**. It deletes the downloaded copy so the file is cloud-only on this computer; the file stays listed and downloads again the moment something opens it. The file list shows that state beside the size: a quiet **cloud** tag (or **partly local** once some of it is cached) marks rows whose full size is on the hub, not on the disk.

- **A file or folder:** in the Desktop App, open the row's menu and choose **Free up space**. On Windows, you can also right-click it in Explorer and choose **Swarmfile → Free up space**.
- **The whole repository:** in the Desktop App, open **⋯ Repository actions** on the active repository in the left rail and choose **Free up space (whole repository)…**. It shows roughly how much it will free before you confirm, and it also clears older versions of files that are still cached.
- **From a terminal:** `swarmfile hydrate free-space [path]`; add `--dry-run` to see what it would free without deleting anything.

It is safe to run at any time:

- **Changes that haven't synced yet are never touched.** A file with a save still on its way to the cloud is skipped, and the result says how many were kept.
- **Make available offline is turned off** for what you free up, since keeping it offline and freeing its space contradict each other.
- **Content something else is holding stays**, such as work staged in a changelist, a Pack & Go reservation, or a file that is open right now. The result names the reason per file — open, staged, or reserved — and says whether it clears on its own, instead of lumping every hold together.

Executable files

A file can be marked executable: a script, a build tool, anything you run with `./name`.

On macOS and Linux, `chmod +x` on the drive sets it and `chmod -x` clears it; the drive shows executable files as `0755` and other files as `0644`. Windows has no executable bit, so edits there leave the flag as it was. From any platform, `swarmfile chmod +x <path>` (or `-x`) sets it.

The flag syncs to everyone like any other change, works offline (it's sent when you reconnect), survives safe-saves that replace the file, and becomes mode `100755` in a git clone. On macOS another machine's change can take a couple of seconds to show, the length of the system's file-attribute cache.

Windows alternate data streams

On Windows, NTFS alternate data streams (`file.ext:streamname`) are supported on the mount: applications can create, read, write, enumerate and delete them, a stream's content travels the same upload and commit pipeline as file content and syncs to other clients, and a stream survives a copy or a rename of the base file.

The one exception is `Zone.Identifier` — Windows' Mark of the Web — which stays on the machine that wrote it and never syncs (see Sync Exclusions). Renaming a stream on its own is refused, and Windows extended attributes are a separate mechanism and aren't implemented. See Filesystem Compatibility for the verified rows and remaining gaps.

Deleting

Delete from the drive exactly as you would any other file — your file manager, or `rm`. Deleted files go to Trash rather than disappearing, and are restorable for as long as your organization's retention window allows: see Trash, History & Rollback.

Folder deletes and files in use

Two behaviors worth knowing. On the mounted drive a folder is removed only once it's empty: your file manager, `rm -r` and `rmdir /s` empty it for you, one file at a time (PowerShell's `Remove-Item -Recurse` is a known non-starter through the mount — use `rmdir /s` or `Directory.Delete` there), while a plain non-recursive delete of a folder that still has files is refused, as on any disk.

In the dashboard, deleting a folder removes its contents with it in one step (they all go to Trash together), so the confirmation there names what's inside; a very large folder still leaves the list at once, the delete showing progress in the toast while its contents follow as a background job.

Restoring a folder you deleted from the drive offers its files back too; see Trash, History & Rollback. And if a file is open or locked by someone else, the delete is refused rather than queued: your file manager reports it, and the Desktop App's Presence view shows who has it.

Applied immediately, confirmed in the background

Renaming or moving a confirmed file, or deleting a file or empty folder, takes effect on your machine at once, so a slow or briefly unreachable link can't stall it. The change is sent in the background and shown as a lightweight overlay until the hub confirms it, then re-verified and refreshed in place.

Another machine that has already received the change shows the new name right away too, with that row marked **Waiting for the hub** instead of **Synced** until the confirmation lands; one that hasn't received it yet keeps the old name until then.

Unconfirmed work is held and retried rather than dropped; if the hub refuses it (for example, you don't have write access), it's retracted and the entry returns.

A refused save-by-replace (an app writing a temporary file and renaming it over the original) brings the original back with your unsynced edits to it intact and readable at once.

The dashboard's history, Trash, merge-request and RFI views catch up as it lands.

The behavior is on by default; an operator can turn it off with `deferred_namespace: false` in the engine config (`SWARMFILE_DEFERRED_RENAME=0`), which makes these operations wait for the hub again.

Saves propagate the same way. Once a local flush commits, the update reaches other machines by gossip before the hub round-trip confirms it, and a teammate's row shows **Waiting for the hub** — with the size you saved in its tooltip — as soon as the projection arrives.

The file's *content* is the exception: a projected version is display-only, so every machine keeps reading the version the hub has confirmed until the confirmation lands, and an abandoned save is retracted rather than served.

Queued saves refresh on a 60-second loop. Size and modification-time display still come from the hub. A machine can be told to fetch a projected version early: Settings → **Prefetch projected updates** (off by default) warms the new version in the background so it opens immediately once the hub confirms it.

Presence

The dashboard shows a real-time "who's editing what" indicator. This is tied to the entry lock lifecycle, not a free-running heartbeat — it reflects files that are actually open and locked, not just which teammates are online. If presence shows someone on a file, they hold (or recently held) a lock on it. This is the authoritative "who has this file" signal: it means a real Swarmfile lock exists, and writing is genuinely blocked while it does.

A teammate appears when they first change a file and drops off when they close it — an app that opens a file read-write only to show it (QuickTime Player opens every movie that way) takes no lock and never appears. A dashboard action that locks a file only briefly, such as rolling back a version, never shows as editing.

When a teammate is creating many files at once, the Files view also shows "N files arriving from ..." until the hub has recorded them all.

In the Desktop App, a presence row that arrived as a LAN hint from a teammate's engine rather than in the latest hub snapshot carries a muted `LAN` marker, so a hint-derived "editing" state is visibly distinct from hub truth. Presence is only in the signed-in dashboard, never on public repository pages or share links.

Separate from locks, a teammate can opt in to share what they are viewing. With Settings → **Share what I'm viewing** on (off by default, per machine), a file they have open right now shows a muted **viewing** marker on your rows and in the Presence view — alongside, not instead of, the lock-derived editing state, and it never blocks anyone. Viewing hints travel peer-to-peer only: the hub never stores them, they expire on their own (after roughly 90 seconds), and they clear as soon as the teammate moves on. Several viewers of one file roll up into one badge with their names in its tooltip.

"In use" signal (Windows)

There's a second, softer indicator that answers a related but different question: not "who holds a lock" but "is some Windows application sitting on this file right now". Many desktop apps — Excel, and a number of CAD tools — open a file for writing and deny write-sharing to everyone else while they have it open, without ever taking a Swarmfile lock. When the Windows drive sees an open like that, it reports it best-effort so teammates get a heads-up before they start editing the same file.

Where it shows up: an informational ℹ️ **in use** pill in the web dashboard's Files table, and in the Desktop App's file list, naming who has the file open and on which machine. It's styled as a note on purpose, not an alarm.

Two things to be clear about:

- **The pill is a heads-up; the enforcement is separate.** The pill itself doesn't block anything — it's a best-effort display. But the *open* it reflects (write intent with write-sharing denied) is exactly what worksharing enforces across machines: on Windows, a second machine that opens the file the same way is refused, so two people generally can't open the same central model for editing at once. The pill tells you; worksharing stops you. To reserve a file you aren't actively holding open in an app, use an entry lock, below.
- **It is distinct from Presence.** Presence (above) is driven by Swarmfile's own explicit lock lifecycle. The "in use" pill is a passive read of what a Windows app is doing to the file — it has no lock lifecycle of its own; the cross-machine blocking rides on the underlying open (see worksharing, above). The two can disagree (a file can show "in use" without an explicit entry lock, and vice versa), and that's expected.

Because it's best-effort and poll-driven, it's not instantaneous: the dashboard and Desktop App refresh it about once a minute, and a file that stays open unusually long can eventually drop off the signal even while it's still open. Treat it as a hint, not a source of truth. It's Windows-only — other platforms don't surface this open mode.

Entry locks

Locking a file blocks other users from writing to it until you release the lock or it expires. There are two kinds, and they behave differently:

- **The automatic entry lock** a file takes from the moment your application first writes to it until it closes the file. Opening a file read-write does not take it on its own; the first write, truncate, or overwrite does, and if someone else holds the file that write is refused ("try again") rather than the open. On Windows, opening a file for writing while someone on another computer is editing it is refused at the open instead, as "file in use", which apps report the way they report any file open elsewhere. It has a **60-second TTL** and is renewed with a heartbeat while the file stays open — you don't need to manually extend it. If your app crashes or your machine loses connectivity, this lock expires on its own within 60 seconds rather than staying stuck. If the lock is lost while the file is still open (it expired, or someone else took it), macOS and Linux refuse the next save — the app sees a generic `EIO` ("Input/output error"); the Desktop App names the real reason. Windows keeps the save on your computer instead. It syncs normally if the lock can be taken again; if the file was changed elsewhere in the meantime, your save becomes a conflict to resolve rather than overwriting the other version. Two saves racing on the same file with no lock held never silently overwrite each other

either: the losing version is kept — as a parked conflict to resolve, or as a copy beside the file (`report (conflict-1a2b3c4d).pdf`; a dotfile or extension-less name just gets the suffix) — so no edit is dropped. (The deferred-namespace opt-out changes *when* the racing save is applied — synchronously instead of through the queue. In that mode a safe-save aside still lands its copy, but a racing rename surfaces as a save error instead of a copy, and the next attempt tries again — nothing is ever dropped silently.) Until it does one or the other, the Desktop App's file list marks the file as kept on this computer, and counts it under the files that need attention.

- **An explicit edit lock** you take yourself. `swarmfile lock <path>` is long-lived — 7 days by default, up to 30 — and its lease is renewed while you hold it. A `git lfs lock` takes a fixed 30-day lease that nothing renews, so a lapsed one releases itself; take it again if the work outlives it. Both last until you release them or the lease lapses; see Entry locks in the CLI reference.

A bulk Pack & Go reservation is a different mechanism: it reserves a whole folder or repository for offline work rather than locking one file, though it blocks writes the same way while it's active.

The lock is keyed to the **user and machine**, not to the mount: two drives of the same engine on one computer share that identity, and share the lock: closing the file in one drive doesn't release it while another drive still has it open. An entry lock taken through one drive does not by itself keep a sibling drive out. When several drives on one machine must not edit the same file — a fleet of agents, say — open them with exclusive-write mode.

The web dashboard's repository-scoped **Locks** tab (admins and owners only) lists every file currently locked in the repository — standard and 30-day edit locks alike — with who holds each one, and lets an admin release their own lock from there directly. A plain member releases their own edit lock the same way they took it — close the file (the short-lived automatic kind clears on its own) or `swarmfile unlock <path>` for an explicit edit lock — rather than through the dashboard. Releasing someone **else's** lock, admin or not, still goes through the request-unlock flow below.

Asking someone to release a lock

If you need a file someone else is holding open, right-click it in the web dashboard's Files view and choose **Request unlock** (also reachable from the Windows Explorer `Swarmfile ▸` submenu on a mounted drive, or from the CLI with `swarmfile unlock-request create <path>`).

The current holder gets a notification — in-app only, not emailed — with a **Grant**/**Deny** right on the notification row, whether that's a modal in the Desktop App naming who's asking or the web dashboard's Inbox (or, from a script, `swarmfile unlock-request list` / `respond` to see and answer pending requests headlessly).

They can release the lock from there, or ignore it and let it expire on its own the normal way — a short-lived entry lock clears when they close the file or within 60 seconds if their app or machine has gone away; an explicit edit lock runs out at the end of its lease (up to the 30-day maximum).

There's no force-release button anywhere — asking is the mechanism, full stop; a still-pending request you made can be withdrawn from the same file's right-click menu (**Cancel unlock request**) or with `swarmfile unlock-request cancel <request-id>`.

The one exception is the git-LFS surface: an entry lock and a `git lfs lock` on the same file are the **same** server-side lock, so a `git lfs unlock --force` can break another user's lock (git-LFS's own `--force`, not a drive/dashboard action).

Byte-range locking

Entry locks cover a whole file. For native worksharing apps that need to lock just part of a file — the way a Revit-class BIM tool locks the elements one person is editing without blocking everyone else out of the same model — Swarmfile exposes a separate, lower-level byte-range locking API. This is what a CAD/BIM plugin integration would call directly.

The API is enabled only on repositories set to the Revit-worksharing repository type — at creation, or later from **Repository settings** or `swarmfile repository set-type revit_worksharing` (see Worksharing); the engine runs the byte-range lock manager only there, and other repositories refuse those requests.

The swarmfile-brlock reference client, available to plugin developers on request, documents the API's operations.

File and folder watch

Watch is distinct from the general "someone else changed a file" bell notification you see for activity across a repository. Watch lets you subscribe to a specific file or an entire folder and get notified when it changes — useful for scripting a rebuild step or keeping an external tool in sync with a subtree of a repository without polling it.

Subscribe with `swarmfile watch <path>`, list your subscriptions with `swarmfile watches`, and stop with `swarmfile unwatch <path>` (see `watch`).

If the drive stops responding

Occasionally the drive can stop answering: files won't open, a Finder or Explorer window sits there, and the application waiting on it may not even force-quit cleanly.

That behavior is deliberate, favoring write safety. The drive is mounted to *wait* rather than to fail, because a filesystem that returns an error partway through a write can leave an application believing a save succeeded when it didn't — which on a multi-hundred-gigabyte repository file means silent corruption. The cost of that choice is that a drive which stops answering waits indefinitely instead of erroring.

Swarmfile watches for this and reconnects the drive on its own where it can. While that's happening the Desktop App reports the drive as unavailable rather than continuing to show a healthy mount; when it reports the drive as available, it is.

When it doesn't clear by itself, open the Desktop App, go to Settings → Diagnostics, and use **Repair drive** (macOS and Windows; Linux has no in-place repair, so restart the engine or remount instead). It reconnects the drive in a few seconds and touches nothing else — your queued uploads are held in a durable local queue, so nothing waiting to upload is lost, and you don't need to quit and reopen Swarmfile.

If the drive never appeared at all, **Repair drive** has nothing to cycle: an all-slots-taken failure (`fuse_t_helper_ports_exhausted`) needs a slot freed first. FUSE-T's 18 mount-helper slots are shared with every FUSE-T app on the Mac. Close a drive or quit the app holding one, then use **Try again** or `swarmfile mounts retry` for a boot mount that never attached; a boot mount that attached earlier needs a restart, and a failed runtime mount is re-opened. `swarmfile doctor` names the holders.

If the drive keeps dropping instead of recovering — the Desktop App's "keeps disconnecting" card — it is usually a leftover mount from a killed session. The engine clears those automatically before each mount, so **Repair drive** normally reconnects it; if Diagnostics still reports a stuck mount, only a computer restart clears it (the state lives in the kernel, not the install), and **Reinstall…** is for a genuinely broken install.

For working out whether the underlying problem is your network, your account, or the service, run swarmfile doctor — its output is what support will ask for first.

Working Offline (Pack & Go)

Most of the time your drive is a live view of the repository: files stream on demand and saves sync in the background. But sometimes you need to work somewhere the network doesn't reach — a set visit, a flight, a site with no usable uplink. Swarmfile handles that, and there are two different things you can set up before you go:

- **Reading offline** — download a scope so every file in it opens with no network.
- **Writing offline** — *reserve* a scope so it stays editable with no network, and nobody else can change those files while you're gone. This is **Pack & Go**.

You can also just disconnect and keep working — saves queue durably and sync when you're back (see Casual offline below). Pack & Go is for when you want to be sure the files you need are readable *and* writable while you're away.

The organization's offline access window

One organization setting can bound all offline work: an **offline access window** (owners and admins set it under **Settings → Organization**). It defaults to **7 days** (set it to 0 to turn it off, or choose another window from 1 hour to 30 days). When a window is in force, a computer that hasn't reached Swarmfile within that window **pauses local file access** — reads and writes are refused, even for files kept offline — until it reconnects; the Desktop App explains the pause on the drive, and it clears by itself as soon as the hub is reachable again (no restart needed).

It exists so an organization can put a ceiling on how long a machine that stops reaching Swarmfile can keep serving its cached copies. A Pack & Go reservation doesn't override it mid-trip, so make sure the window covers how long you'll be away (lengthen it, up to 30 days, if your team works offline for longer stretches).

Casual offline (no preparation)

If you only need the files you've already been working on, you don't have to do anything special.

- **Saves keep working.** Every write returns immediately against a durable local queue, so a dropped connection doesn't block your application. The queue survives quitting or restarting Swarmfile and resumes when the network is back — see Reading the sync status.
- **Renames, moves and deletes keep working.** A rename or move of a synced file or folder, and deleting a file or empty folder, applies on the drive immediately and is sent to the hub when it's reachable — no waiting for a connection. If the hub later refuses it (a protected branch, say), Swarmfile puts the original name back and reports why. That includes an app that saves by writing a temporary file and renaming it over the original, as most editors do: if that replace is refused, the original file comes back with your unsynced edits to it kept and shown on your drive straight away, and they still upload.
- **Reads work for anything still in your cache.** Files you've opened recently stay local until the cache trims itself, and reading a file you already have open keeps working through a brief outage.
- **Reads do not work for files you never downloaded.** A file you haven't opened yet streams from the hub or a LAN peer, so with no network it isn't there.

The catch is collisions: if a teammate changes a file while you're editing it offline, Swarmfile catches it when your save lands and records a **conflict** rather than silently keeping both. Writes to that file are refused until you resolve it. Pack & Go avoids the surprise by telling everyone else the file is yours before you leave.

Figure omitted in print (animation): The connection to the coordination service drops: a file already opened still reads in full from the local machine, while one never downloaded fails immediately instead of hanging; reconnecting restores everything. - https://swarmfile.com/demo/offline-resilience.cast

Download a scope for reading

Use **Make available offline** when you want the content resident but aren't going to edit it.

- **One file or folder:** in the Desktop App, open the row's menu and choose **Make available offline**.
- **The whole branch:** in the Desktop App, open the **⋯ Repository actions** menu on the active repository in the left rail and choose **Download entire branch (keep offline)**.
- **From a terminal:** `swarmfile hydrate start [path]`, then `swarmfile hydrate status` to watch it.

While it downloads, the Desktop App shows progress and lets you cancel. When it finishes, the files are pinned against the cache's normal cleanup, so they stay put until you release them.

To undo it:

- **Stop keeping it offline: ⋯ Repository actions** (left rail) → **Stop downloading / keeping the repository offline**, or `swarmfile hydrate release [path]`. This drops the offline mark; the content stays in the cache until the cache needs the space.
- **Give the disk space back now: Free up space** on a file/folder row, **Free up space (whole repository)…** under **⋯ Repository actions** in the left rail, or `swarmfile hydrate free-space [path]`. Unsynced changes are never touched — see Freeing up space.

Downloading for reading does **not** stop anyone else from writing to those files, and it does not reserve them for you. That's what Pack & Go adds.

Before you go: reserve a scope (Pack & Go)

Pack & Go does two things at once: it downloads every file in the scope, and it reserves the whole scope with one long-lived offline reservation. The reservation is enforced by the hub, so while it's active nobody else can write those files — and you can keep writing them with no hub at all. The hub refuses a reservation that overlaps one another machine already holds, or that covers a file someone else is editing right now, and names who is in the way. Active reservations are visible in the Desktop App's **Locks** panel, with who holds them and when they expire.

In the Desktop App: open **⋯ Repository actions** on the active repository in the left rail and choose **Reserve entire branch for offline (Pack & Go)…**. It shows the size before you confirm, lets you pick a reservation length (7, 14, or 30 days), and reports progress; if the scope is blocked, it tells you who holds it and reserves nothing.

From a terminal:

```
# Reserve a folder for a two-week trip: the lease is extended to cover it
swarmfile offline prepare ./Repositories/shots/seqA --offline-days 14

# Check progress, resume one a restart interrupted, or release everything
swarmfile offline status
swarmfile offline resume
swarmfile offline return ./Repositories/shots/seqA

# Stop a prepare that's still running (reservations already taken are kept)
swarmfile offline cancel
```

`offline prepare` follows the download job to completion; add `--no-wait` to print its id and return immediately (follow it with `offline status`). `offline cancel` stops a running prepare — the reservations already taken are kept, so re-running `prepare` or `resume` picks up from where it stopped.

Reserve only what you need rather than the whole repository where you can — the locks you take are locks your teammates can't use, and a smaller scope means a faster prepare.

How long a reservation lasts

- The lease defaults to **7 days** and can't exceed **30 days**.
- It is **only renewed while you're online** — at startup and every six hours after that. A reservation does not extend itself while you're disconnected, so pick a length that covers the whole trip.
- If a lease lapses while you're still offline, the file is no longer exclusively yours: the engine warns you as the expiry approaches, and anyone else can claim the file once it does. Passing `--offline-days` (how long you expect to be away) **extends the lease to cover the trip** when it would otherwise lapse; `--duration-secs` sets an exact lease (up to 30 days) and, if you give an explicit one shorter than the trip, the command warns rather than overriding your choice.
- A single prepare covers up to 100,000 files. A larger scope is refused with a message telling you to narrow it.
- If a prepare is interrupted (a crash, a restart), `swarmfile offline status` names the interrupted scope and `swarmfile offline resume` completes it without re-reserving what you already hold. `offline status` also prints what this machine holds right now (file checkouts and folder/repository reservations).
- **Switching a mount to another branch releases the reservation(s) that mount held on the branch you're leaving.** A reservation protects one branch's paths, and after the switch the engine can neither renew nor release it, so Swarmfile hands it back instead of leaving your teammates blocked until it expires. If you'll still need offline work on the other branch, reserve there after switching. **Switching the mount to another repository or organization does the same** for the reservations on the repository you're leaving. Per-file checkouts are different: they stay yours, keep renewing on the repository they were taken in (after a switch within the same organization), and survive quitting and restarting Swarmfile; after a switch to another organization they wait, and resume when you switch back.

Coming back

Choose **Return from offline (release reservations)** under **⋯ Repository actions** in the left rail, or run `swarmfile offline return` with no path to release every reservation this machine holds. A path releases the reservations at or under that folder. Releasing doesn't delete any content — the files stay in your cache until it needs the space, and you can free them deliberately as above. If saves are still syncing, the release waits a few seconds for them and then tells you how many are still going; they land on their own, but until they do a teammate could take one of those files and force a conflict.

While you're offline

Inside a reserved or downloaded scope:

- Opening and reading files works from the local copy.
- Saving works; saves queue locally and sync when you reconnect.
- Creating files and folders works; the durable create queue lands them when the hub is reachable again.
- Changelists (staged workgroups) keep working.

Anything that needs the hub doesn't: listing someone else's live presence, taking a new lock on a file, comparing history, comment threads, and merge requests all wait until you're back.

Renaming, moving or deleting an already-synced file is the exception: it applies locally and syncs when you're back (see Casual offline).

Any existing file on a mount you can write to can be edited offline, whether or not it's inside a reserved scope: the save is queued and reconciled when you're back. What a reservation buys you is the guarantee — nobody else can change those files while you're away, so your saves land cleanly. Outside a reservation you may get a **conflict** if a teammate changed the file first (see below); to be sure the files you need are covered, include their folder in the prepare.

Coming back and resolving conflicts

When the network returns, queued saves land on their own. The Desktop App's header stays on **Syncing N** until the queue is empty — see Reading the sync status.

If someone changed a file you edited while you were away, you'll get a **conflict** instead of a silent overwrite. The file is read-only until it's resolved (a save fails with "permission denied", or "access denied" on Windows), and the Desktop App shows it on the **Branches** page (on the activity bar, the icon strip down the left edge), including the Conflicts section. From the CLI:

```
swarmfile conflicts                    # list blocked files
swarmfile conflicts resolve <path> --auto          # three-way merge when possible
swarmfile conflicts resolve <path> --winner local  # or remote, or keep-both
```

`--auto` exits with code 2 when it can't merge cleanly, so a script can tell "a person must choose" apart from a real failure. See Branches & Merging for the conflict model, and the CLI reference for every `conflicts` flag.

For administrators

Bulk Pack & Go can be turned off on a machine or fleet with the `SWARMFILE_PACK_AND_GO_POLICY` environment variable on the engine process (`disabled` refuses both bulk prepare and single-file checkout locks). Any value outside `allowed` / `disabled` **fails closed** — a typo disables Pack & Go rather than allowing it — and the engine says so at startup (and `swarmfile-doctor` reports it), so a typo surfaces immediately instead of when someone first tries to pack.

On a managed deployment the service can additionally deliver an organization-wide Pack & Go policy to each engine; the stricter of the organization policy and the machine's environment variable applies. See Engine Config File.

Related

- Working with Files — saving, sync status, freeing up space, and locks.
- CLI: swarmfile — `hydrate`, `offline`, `fetch`, and `conflicts` in full.
- Engine Config File — `SWARMFILE_PACK_AND_GO_POLICY` and the streaming/cache settings.

Multiple Mounts

A Swarmfile engine can host more than one mount at a time, so you can work in two repositories — or two branches of one repository — side by side on the same machine, each as its own drive. This is the quick, human-facing version of the setup described for automated fleets in Coding Agents.

When to open a second mount

- **Two repositories at once** — a live repository and a prep repository, or two clients you switch between all day.
- **Two branches of one repository** — compare a look-dev branch against main without switching the whole drive back and forth.
- **An agent or script that needs its own workspace** — a second mount keeps its changelists and locks separate from yours.

Each mount is a full, symmetric mount: its own branch, its own open changelists, its own lock state. Nothing you do in one changes the branch the other is on.

Opening one

In the Desktop App, hover a repository in the left rail and click its **+** to open that repository as another drive, or use the **+** half of the mount switcher (the drive name at the top of the window), or **+ Open another drive...** inside the switcher. Each opens the **Open a drive** dialog, where you pick:

- the **repository** (preselected when you came from a repository's **+**) and optionally a **branch**;
- an optional **name** so you can tell drives apart ("agent-2", "client-b") — the same thing the CLI calls a label (`--label`).

Swarmfile chooses where the drive appears — the next free drive letter on Windows, and a new folder beside your main drive on macOS and Linux, named after the drive (or its branch) — and the dialog shows the exact letter or folder before you open it. To pick a specific drive letter or an empty folder of your own, open **Advanced**.

If a drive can't open, the dialog says why straight away — for a taken drive letter it offers the next free one — and the failed drive never becomes the current one. A drive that is slow to attach closes the dialog and shows **Connecting...** in the list (it can't be switched to until it's ready); Swarmfile switches to it once it's live, unless you've moved to another drive in the meantime, and a notice says how it went.

From the CLI, the same thing:

```
swarmfile mounts open Y: --label client-b                   # Windows: a drive letter
swarmfile mounts open auto --label client-b                 # Windows: next free letter
swarmfile mounts open ~/swarmfile/client-b --label client-b  # macOS/Linux: a folder
swarmfile mounts open Y: --label dev --branch look-dev-warm  # pin this mount to a branch
swarmfile mounts open Y: --label side --repository proj_…      # a different repository (same org)
swarmfile mounts open ~/agent-2 --exclusive                 # exclusive-write mode (agent fleets)
```

On Windows the folder or drive letter must be free — an existing non-empty folder is refused rather than mounted over, while an existing **empty** folder is accepted (the app's folder picker only returns folders that already exist, so this is the normal second-mount shape). On macOS and Linux the mount point is a normal folder, and the same path can't be used by two mounts at once. See Filesystem Compatibility for the per-platform details.

Keeping two agents off the same file

Locks protect a file against *other people and machines*, but two mounts of one engine share one machine identity — so by default nothing stops two agent mounts from editing the same file at the same time; the later save comes back as a conflict to resolve rather than silently winning. If you want the second writer refused at open instead (and the file protected from rename, delete or atomic-save replace while it's open), open each agent's mount with `--exclusive`:

```
swarmfile mounts open ~/agent-1 --exclusive --label agent-1
swarmfile mounts open ~/agent-2 --exclusive --label agent-2
```

Exclusive-write mode is opt-in per mount. While one exclusive mount has a file open for writing, another exclusive mount on the same repository+branch has its write-open refused — with `EAGAIN` on macOS and Linux, and on Windows with the sharing violation apps report as "file in use" — and a rename, delete, or atomic-save replace of that file is refused too, so the holder's file can't be moved, replaced, or removed out from under it.

The refusal is recorded (with the holding mount) in the tray's blocked-writes view and in `swarmfile status`. Closing the file — or finishing its save — releases the claim. A mount opened **without** `--exclusive` neither takes nor honors claims, so your human drive and non-exclusive mounts behave exactly as before; the protection is between the mounts that opted in.

In the desktop app the same option is the **Prevent another drive from editing a file this one has open** checkbox under **Advanced** in the **Open a drive** dialog, and an exclusive drive carries a lock badge in the drive list. Like the flag, it is fixed when the drive opens — close and reopen the drive to change it.

Same dialog, one field down: **Conflict handling** pins this drive's conflict mode — the per-drive form of `mounts open --conflict-mode` (CLI reference). Leave it on the repository setting unless the drive is an automated writer, which needs a refused save to fail its `fsync` (Linux also the close-time `flush`; Windows `FlushFileBuffers` — a macOS `close()` cannot carry the error by design): **Hard: fail the save** makes that call return a stale-version error instead of parking the conflict for later resolution, so the writer's loop can re-read and retry.

Fixed once the drive is open, like the other open-time options here — a pinned mode shows on the drive row and in `mounts list` / `mounts status`, and a successful open warns if the engine didn't apply it.

Working with several mounts

- `swarmfile mounts list` — every mount, its label, branch, repository, and which one is `current`.
- `swarmfile mounts current <id>` — choose which mount commands without an explicit target should act on.

- `swarmfile --mount <id> status` — run any command against one specific mount, without changing the current one. Write the flag **before** the subcommand.
- `swarmfile mounts branch <id> <branch>` — move one mount to another branch without touching the others.
- `swarmfile mounts close <id>` — shut one down. It refuses while a changelist is open (`--force` abandons it), and it always refuses the boot mount.

The Desktop App does the same: each repository in the left rail lists its open drives underneath, each row showing the drive's path, branch and status with a control to make it current and a **×** to close it (the boot drive's **×** has no close action — it explains that the drive can't be closed and that switching its repository repoints it).

Sparse mounts

Sometimes a mount should show only part of the repository — the one folder a coding agent works in, or a single discipline's directory on a giant media repository. `swarmfile mounts open <path> --sparse "<glob>"` opens a **sparse mount** that exposes only the paths matching the include globs; `--exclude` takes the complementary form. The glob dialect matches `materialize --sparse`: a glob matching a directory includes its subtree, a leading `!` excludes an earlier match, and a bare name matches at any depth.

A sparse mount is honest about the boundary: an excluded path is invisible at that drive (no listing entry, an open fails as if it weren't there, and creating or renaming onto it is refused). The scope is per mount, so a second, full mount of the same repository on the same machine is unaffected. Offline sets respect the scope — `swarmfile hydrate start` on a sparse mount can narrow inside it but never widen past it, and Pack & Go takes exactly what the mount exposes. When a render node needs the whole repository resident instead, `swarmfile hydrate start` with no path pins everything under the mount's scope.

See the CLI reference for the full flag set.

Limits and practical notes

- **macOS:** FUSE-T provides 18 mount-helper slots per Mac, shared with every FUSE-T app on it (not a macOS mount limit). If an `open` fails with `fuse_t_helper_ports_exhausted`, close a drive or quit the app holding a slot, then retry; `swarmfile doctor` names the holders. A boot mount that never attached can retry in place with `swarmfile mounts retry`; one that attached earlier needs a restart.
- **One engine, one cache directory.** Don't point two engines at the same cache directory or mount point. Running a separate engine per mount is a valid isolation choice, but it must have its own `SWARMFILE_CACHE_DIR`.
- **Shared scratch cache.** All mounts of the same repository on a machine can share one scratch directory for tool caches, so a second mount doesn't re-download dependencies — see Coding Agents.
- **Switching the whole engine** is different from picking a current mount: `swarmfile workspace switch` moves the engine's active org/repository — usually in place, without a restart. If the hot path cannot complete, the engine drains and restarts instead (the OS supervisor relaunches it), so the drives briefly unmount and return. Mounts of the repository being left follow the switch to the new repository's active branch; mounts on other repositories stay put — and one on a different repository blocks an org switch until it is closed (the app names both before you confirm). See Switching the active organization or repository.

Where to go next

- CLI: swarmfile → mounts — every `mounts` subcommand and flag.
- Working Offline (Pack & Go) — reserving a scope on one mount for offline work.
- Coding Agents — the headless, one-mount-per-agent version of this setup.

File Previews

Browsing a repository shouldn't mean opening a native app just to check you're looking at the right file. The web dashboard renders previews and thumbnails for common media directly, without downloading the whole file to your machine — the preview is generated server-side and you're sent a small image, not the 4 GB source.

This is a dashboard and share-link feature. It doesn't change anything on the mounted drive: there, a file is just a file, and your own applications open it.

Thumbnails in the file browser

The Files view renders a thumbnail for any file type Swarmfile can rasterize, so a folder of renders or plates reads as a contact sheet rather than a list of names. Thumbnails are generated on the server and cached, so the first view of a file pays for generation and every view after is instant.

Thumbnails are produced for:

- **Images** — PNG, JPEG, GIF, WebP, BMP, TIFF, AVIF, and **Photoshop (** `.psd` **)** files. The PSD path flattens the composite, so you get the image a designer sees, not a layer dump.
- **PDFs** — the first embedded JPEG image found in the file, not a true page render. This works well for the common case of a scanned or photo-heavy PDF, but a PDF with no embedded JPEG (pure vector graphics, text-only, or an embedded image in a different format) gets no thumbnail. Thumbnails use an embedded image; full first-page rendering isn't offered. (The full inline preview below is unaffected — it embeds the real PDF via the browser's own viewer, not this thumbnail path.)
- **Video** — a poster frame (see below).
- **Point clouds** — a top-down preview (see below).

Inline preview

Clicking a file opens an inline preview modal, without leaving the dashboard. Images, PDFs, and plain-text files render directly in the browser. PSDs and large TIFFs show their generated thumbnail instead of rendering the raw file inline — the browser can't display the source, but the poster tells you what it is. Video shows a poster and, up to 16 MiB, a scrubbable 720p proxy.

Rendering a source file inline is capped at 5 MiB — a browser tab isn't a useful place to view anything larger, and assembling a bigger file just to display it inline holds the server working for no real benefit. Past that size the modal falls back to the type's generated thumbnail or proxy instead (video proxies cover sources up to 16 MiB — see below); for a type with neither, it tells you to download the file. Downloading itself has no size limit.

Anonymous visitors on a public release page get a smaller inline cap of **256 KiB**, and only images and text render inline there — video and audio stream by range in the player instead (see Public repositories).

Video posters and scrubbable proxies

Video gets two levels of preview, both generated on the server rather than shipping the source anywhere:

- **A poster frame** — a single still, shown as the file's thumbnail, for `.mp4`, `.mov`, `.m4v`, `.webm`, `.mkv`, `.avi`, `.wmv`, `.mpg`, and `.mpeg`.
- **A scrubbable proxy** — a 720p H.264 MP4 the browser can play and scrub through, transcoded in the background from the full source and served with HTTP range requests. This lets a reviewer scrub a cut in the dashboard without pulling the master.

A poster frame is extracted from the first 16 MiB of the source, so it works even for a very large video — as long as the frame index sits near the start, which is the usual case for web-friendly MP4s. The scrubbable proxy needs the whole file, so it's generated only for sources up to 16 MiB; a larger video gets no proxy.

Container previews are rate-limited, per repository and again across the whole org (500 per repository and 2,000 across the org per day by default). An organization's owner can raise or remove the caps under **Settings → Usage → Preview generation budgets** (`0` = no limit); a blank field keeps the default.

Once a budget is used up, new video/point-cloud previews are refused with a typed `429` until the next day — the response says whether the repository's or the organization's daily budget ran out and when it resets — but already-generated previews are cached and stay free to view. The dashboard flags the refusal with a "Previews are paused" banner rather than leaving a folder of empty thumbnails, and for an owner the banner links straight to those budget controls.

Raw and camera-original formats (`.r3d`, `.braw`, `.mxf`, and similar) are **deliberately excluded** — they aren't reliably decodable without the right vendor codec, so rather than render a wrong or broken frame, Swarmfile shows the generic file icon and leaves them to a native app.

Point-cloud previews

`.las` and `.laz` point clouds get a rendered top-down preview, so a folder of LiDAR or photogrammetry tiles is browsable by eye instead of by filename. Like video, this is generated and cached server-side from the source.

`.e57` and `.ply` are **not previewed** in this version — `.e57` needs a plugin and `.ply` lacks the georeferencing the top-down render depends on. The source cap for a point-cloud preview is 32 MiB.

Previews in share links

A share link carries the same inline preview for images, PDFs, and text, so an external reviewer with no account can see the shared item in their browser. See Sharing & Collaboration for how share links work, and Working with Files for the mount side.

What isn't previewed

Limits, so you don't go looking for something that isn't there:

- **End-to-end-encrypted content has no server-side preview.** Previews are generated on the server, and for an E2E repository the server cannot decrypt the file — so E2E files show a generic icon, not a thumbnail or proxy. This is the deliberate trade of the E2E tier; see Security.
- **CAD/BIM files (DWG, RVT) show a generic icon; native design-file rendering isn't offered.** Image, PDF, PSD, video, and point-cloud previews are live.
- **Caching is content-addressed.** Editing a file changes its content id, so its preview regenerates on the next view rather than serving a stale one — the first view after an edit pays for generation again.

Search

Search runs on a full-text index and covers entry names (files and folders), repository names, and people (principal names). It's a metadata search, not a content search — it finds things by name, not by what's inside a file. Three scoped exceptions: the public Explore page also matches repository descriptions, topics, paths and README text; a public repository's release view offers **"Search file contents..."** over the text files of that repository's indexed release; and the dashboard omnibox searches RFI and submittal title and body text with a repository selected (see below).

Where to search from

Dashboard omnibox. The global search bar in the web dashboard is the fastest way to jump to a file, repository, or person by name. With a repository selected it also searches that repository's RFIs and submittals by title and body — see RFIs & Submittals — filtered to the ones you actually have read access to, the same as everything else it returns; on org-level views (no repository selected) it returns name matches only.

Desktop App. The file picker's search box searches the current repository's files and folders by name as you type — the quickest way to jump to a file while you're already working in the app. If the hub can't be reached, it falls back to the local cache and labels the results *"from this machine's cache; hub search is unavailable"* so a thin result is explained rather than mysterious.

`swarmfile-search` **CLI.** A standalone binary for searching from a terminal or a script. Give it plain text and it searches across entries, repositories, and people — an org-wide query (no `--repository`) returns repositories and people only; file and folder hits need a repository scope. Cap results with `--limit`, or force it to search against the local cache or the hub with `--local-only` / `--remote`. Full flag reference: swarmfile-search.

The standalone binary stays name-only — to search RFIs and submittals from a terminal, use `swarmfile rfi search` against a mounted repository.

Public repositories on Explore

Search on the public Explore page works a little differently: it matches public repositories by name, description, topics, org, slug, file paths, and README text, so visitors can find a repository by something they remember from inside it. It only ever returns repositories that are deliberately published as public. The listing itself sorts by **Most downloaded** (the default), **Most starred**, **Recently published**, or **Recently updated**.

How it resolves results

The `swarmfile-search` **CLI** resolves against its local metadata cache first and falls back to the hub, so it keeps working with a flaky or dropped connection to the extent your cache is already populated — a repository you've browsed recently stays searchable offline, while one you've never opened won't show up until the hub is reachable again.

Deferred renames show under their projected name. If a rename or move hasn't been confirmed by the hub yet — yours, or one your engine received from a teammate — the local-cache path searches it under the new name: an ancestor rename reprojects full paths, and a match on the old confirmed name is shown under the projected one. The dashboard omnibox searches the hub directly, so it sees confirmed names only. See Working with Files.

The **dashboard omnibox** always searches the hub directly: it runs in your browser, which has no local cache to fall back on, so a slow result means the request is still in flight rather than the index being broken. The panel says **"Searching the hub..."** while a query runs and labels the results as coming from the hub.

Notifications & Inbox

Swarmfile tells you when something needs your attention — an upload failed, a file you have was changed by someone else, someone mentioned you in a comment — rather than leaving you to notice it yourself. Notifications are per-user: everyone on the org gets their own, based on their own activity and preferences.

There are three surfaces, and they show the same events:

- **The web dashboard's notification bell**, with an inbox you can scroll back through.
- **Desktop App toasts** on the desktop, for the same events as they happen on the machine you're working on.
- `swarmfile notifications` from the CLI — `list` / `unread-count` / `read` / `read-all` against the same inbox, for a script or a headless machine that needs to check (or clear) what's waiting without a browser or the Desktop App open. See the CLI reference.

What generates a notification

You'll get a notification for:

- **A failed upload** — a save that couldn't be uploaded after retries. Because saving is asynchronous, this is how you find out something didn't land from anywhere; in the Desktop App the header also counts failed changes until they're dealt with (see Working with Files).
- **Quarantine** — the *applied* notice goes to the org's owners, deliberately not to the quarantined account; the person released gets the release notice. A blocked user can send one **review request** per 24 hours from the dashboard; it reaches the org's owners/admins as a bell/tray notice only and never lifts the block. If your own saves start being refused, the Desktop App's "Why did my save fail?" shows the reason. See Security for what triggers quarantine.
- **A quota or billing block on upload** — an upload refused because the org hit a storage limit or has a lapsed subscription, so you know why writes started failing rather than seeing an opaque error.
- **A file you have was changed by someone else** — the repository-wide "someone edited this" signal.
- **Comments and `@`-mentions** — a new comment on a file, or someone tagging you specifically.
- **Watched files and folders** — a change to something you explicitly subscribed to with watch.
- **Unlock requests** — someone asking you to release a lock on a file you're holding.
- **Merge Request activity** — one opened, reviewed (approved, changes requested, or just commented on), merged, closed, or reopened; a draft becoming ready for review; you being assigned, asked to review, or unassigned (and a review request withdrawn); a review being dismissed (sent to the dismissed reviewer and the merge request's author); and labels being added or removed. See Branches & Merging.
- **A hosted-CI run failing** — the run's starter is told when a run ends in failure (a timeout, a cancel, a queue expiry, a blocked start or an infrastructure fault included); on a protected branch the branch's recent committers get it too. Bell/tray and the personal webhook, not email. See Hosted CI.
- **A repository nearing its storage limit** — sent to the org's owners and admins when a repository's file information and history pass the early-warning mark, again if new files and versions get paused at the limit, and once more when it recovers. The repository's **Storage** tab explains what to do. See Repository storage limit.
- **A teammate asking for git access** — sent to the org's owners and admins when someone tries `git clone` / `git push` on a repository that hasn't been enabled for git yet. It arrives once per repository, not once per attempt, and names the fix: Clone a repository with git.
- **Public discussions** — a thread published on a public repository (see RFIs & Submittals) goes to the org's owners and admins by bell and email, once per thread. A reader's **Report** of a thread is a bell-only notice to them (once per thread, however many reports arrive), and a thread's **Make private** rings the bell for members who posted on it. Discussion reply emails to the people on the page — members and visitors alike — carry their own one-click unsubscribe per repository instead of appearing in these settings.
- **Public issues**: a new issue on a public repository (see Public Repositories & Org Profiles) goes to the org's owners and admins by bell and email.
- **Upstream released**: when a repository your org forked publishes a release your fork doesn't have yet, the fork org's owners and admins get one notification per release, by bell and email. The bell entry is filed under your fork; the email names the upstream and the new tag and links to the upstream repository's public page.
- **Your org's dedicated storage finishing provisioning** — sent to the person who enabled the switch (or created the first repository on a paid plan) when the bucket is live, since that can complete well after the request returned. See Dedicated Storage Isolation.
- **Your org's request budgets or spend cap changing** — the org's owners and admins get a bell/tray notice when an owner or admin changes the org's request budgets or spend cap, including when a spend-cap threshold is crossed or a metered write is refused. Bell/tray only; the Activity feed and audit log are the trail.
- **An org-wide rollback finishing** — owners and admins get a bell/tray notice (no email) with the restored/refused counts; the web bell opens that job's result, which lists every repository's outcome, while the desktop tray notice opens the Quarantine view, and the audit log keeps it. See Operations.
- **A commit dispute**: two derivations of the same commit disagree. The org's owners and admins are told (bell/tray only) while git holds that commit back, and told again when Swarmfile settles it on its own, usually within a minute or two. When a differing derivation arrives for a commit that is already confirmed, the notice says the confirmed commit is still served and nothing is held back. See Clone a repository with git.
- **An org webhook being auto-disabled** — when a webhook is turned off after repeated delivery failures, the org's owners and admins and the webhook's creator are told by bell and email so they can fix the endpoint and re-enable it. See Webhooks.

One event is desktop-only: when a found update has sat uninstalled for a week, the Desktop App raises a one-time notification reminding you. It doesn't appear in the bell or by email, and nothing ever installs on its own — see Release Channels & Updates.

Email notifications

Some notifications can also be delivered by email, so an important one reaches you when you don't have the dashboard open or the Desktop App running. The email-able events are the ones you'd want to know about while away from your desk:

- **Uploads, quota and storage:** failed uploads, quarantine (the applied notice to owners, the release to the affected user), a quota/billing block, a repository nearing or hitting its storage limit.
- **Repository activity:** a file changed by someone else, a public access request, a public RFI, a public discussion or a public issue filed or published on a repository you own or administer, an upstream release your fork doesn't have yet, comments, and `@`-mentions.
- **Merge requests:** being assigned to a merge request, having your review requested on one, having changes requested on one you opened.
- **RFI/submittal activity:** created, assigned, response posted, revisions requested, due soon, overdue, closed, voided, and the ball-in-court digest.

- **Org operations:** an org webhook being auto-disabled, and your org's dedicated storage finishing provisioning.

Every other Merge Request event (opened, approved, merged, closed, or reopened) stays bell/toast-only; those happen often enough on an active repository that emailing every one of them would be noise, not signal.

Owners and admins of an org with public repositories also get a **weekly public-repository report** by email (stars, followers, downloads, the newest release and anything needing attention); it has its own row in the settings table and no bell entry. See Tools for maintainers.

Watched-file changes and unlock requests are **in-app only** — they show in the bell and as Desktop App toasts, but aren't emailed.

Email is on by default for the events listed above, with two exceptions: **"a file you have was changed by someone else"** and **new comments on a file** ship muted — on an active repository those fire for ordinary teammate activity, and emailing every one of them is noise. The bell still records both. You control every kind individually, so you can turn those two on, or mute others; mentions, failed uploads, quota blocks, quarantine and the other attention kinds stay on unless you mute them.

Quiet hours

You can set a quiet-hours window — a start and end time in your own timezone — during which we won't email you, so an overnight render failure doesn't wake you. A notification raised during quiet hours still appears in the in-app inbox immediately; only the email is suppressed for the window.

Where to configure it

Email preferences live in the web dashboard's **Settings → Notifications**: the master on/off switch, quiet hours, your timezone, and the per-kind toggles (including the RFI/submittal kinds, each of which can be muted without touching the rest).

The table lists every notification kind — kinds that never send email (watched-file changes, unlock requests, MR opened/merged/closed/reopened, CI runs, and the other bell-only events listed above) show a dash in the Email column, and a configured personal webhook is their per-kind control. Set them once and they apply to your account across every machine you sign in on.

`swarmfile notifications preferences get` / `set` reads and writes the on/off switch, quiet hours and timezone from the CLI (clearing either of the latter is dashboard-only — the CLI has no null form), and `preferences set --kind file_changed_by_other=off` sets a per-kind override (repeat `--kind` to set several; `default` clears that kind's override — email and webhook — back to the kind's defaults: the master switch plus that kind's own default, which is off for the two high-volume kinds above and on for the rest). The dashboard remains the friendliest place to see them all at once.

Personal webhook

A third delivery surface, alongside the bell and email: a signed HTTP callback to a URL only you configure, with the same per-kind mute controls email already has. It's a fire-and-forget best-effort channel — no delivery log or retry, held to the same bar as email rather than the fuller treatment an org-wide webhook gets. See Webhooks for setup and how to verify the signature. One asymmetry worth knowing: a per-kind email override turns that kind's email on even when the email master switch is off, but the webhook's own on/off switch is a hard gate — its per-kind rows only apply while the webhook is enabled.

Notifications vs. the activity feed

The notification inbox is **yours** — the events relevant to you personally. It's distinct from the org **Activity** feed, an org-wide audit log open to any member (ACL-filtered — you see events on entries you can read; quarantine incidents and org-policy events stay owner-only within it). Its CSV export is more restricted still: owner-only, and it needs the Pro plan's `audit_log` entitlement. If you're looking for a complete record rather than your own attention queue, that's Operations → Audit log, not this.

That Activity feed is the same stream wherever you meet it — three surfaces onto one feed:

- **The web dashboard's Activity view** — the full, scrollable, ACL-filtered record described above.
- **The Desktop App's activity feed** — the right-rail, cross-machine feed of what's happening in the repository you're currently working in. Scoped to the active repository rather than the whole org, it's the at-a-glance version for while you're at your desk. Runs of the same person doing the same thing collapse into one row you can expand; updates to system files (`.DS_Store` , Office lock and temp files, macOS safe-save `*.sb-…` leftovers) stay hidden behind a **Show system-file updates** toggle rather than being deleted. If you select a file, the feed narrows to that file's events.
- **`swarmfile activity` (`list` / `export` / `tail`)** — the terminal equivalent: `list` is point-in-time, `tail` follows the feed live (by polling), and `export` writes the owner-only CSV. See the CLI reference.

Same events underneath; pick whichever surface fits what you're doing.

> Not to be confused with `swarmfile tail`, which streams the engine's **local** event bus for the mounted repository (comments, watched-file changes, lock conflicts) rather than this org-wide Activity feed.

Where to go next

- Sharing & Collaboration — comments, mentions, and watch, which are what generate most notifications.
- Working with Files — asynchronous saves and the upload queue, which the failed-upload notification reports on.
- Webhooks — the personal webhook channel in full, plus the org-wide version an owner can set up.

Sync Exclusions

A mounted repository normally syncs everything: any file you write shows up for every other machine mounting it. That's the right default, but not everything belongs on the hub — `node_modules/`, build output, local caches. Swarmfile reads `.gitignore` the same way git does and treats a match the same way sync does: the file stays exactly where you put it and works normally on your machine, it just never uploads.

This is on by default, the same way a `.gitignore` in a fresh git repo is just expected to work. When you create a repository from one of the ready-made templates (Media, Git, Revit, Geospatial), Swarmfile seeds a starter ignore file for that workflow — edit or delete it like any other file.

`.gitignore` vs `.swarmfileignore`

If your repository is already a git repo, its `.gitignore` is enough — Swarmfile reads the same file, with the same syntax, and nothing extra to maintain.

`.swarmfileignore` exists for two other cases: a repository with no git repo at all (a CAD or media repository versioned only by Swarmfile itself), and excludes that are only about sync, not about git — a local scratch folder or a cache directory you'd never want in a git commit either, but don't want to mix into a `.gitignore` that other tooling also reads. The two files use identical pattern syntax and combine additively: a directory can have both, and either one can exclude a path the other doesn't.

Nested rules, just like git

Rules cascade the way `git help gitignore` describes: a `.gitignore` in a subdirectory is layered on top of the repository root's, and if the two disagree, the deeper one wins — including a `!pattern` negation un-ignoring something a shallower rule excluded.

`.git/info/exclude` and your global `core.excludesFile` are honored too, at the bottom of the precedence order: global excludes, then `.git/info/exclude`, then the repository root's own `.gitignore` / `.swarmfileignore`, then anything closer to the file. Within one directory, a `.swarmfileignore` line is applied after that directory's `.gitignore` lines, so it can override a rule the `.gitignore` set for the same directory.

What a teammate actually sees

An ignored file isn't deleted or blocked — it behaves exactly like an untracked file in git. You can create it, write to it, and read it back on the machine that made it, same as anything else on the mount. What changes is visibility to everyone else: the file never uploads, so it doesn't appear in `ls`, Explorer, Finder, or the web dashboard's Files view on any machine whose own ignore rules match it. It isn't hidden metadata with empty content — it's absent from the listing entirely, the same as a file that was simply never created there.

There's one exception worth knowing: opening the exact path directly (not browsing to it) still works, but only if the bytes are already sitting on that machine — because this machine wrote them, for instance. If they aren't there, Swarmfile won't fetch them from the hub just because you asked for the path by name; an ignored file is never fetched, only ever served from what's already local.

The web dashboard's Files view has a **Show ignored files** toggle for anyone who needs to see the unfiltered truth — an admin auditing what's actually sitting on the hub, or checking whether something is worth cleaning up (see below). With it off (the default), the dashboard matches whatever an ordinary mount would show you.

Ignore rules and releases

Sync-ignore hides paths from listings and stops future syncs — it does **not** keep anything out of a release. A release publishes the whole tree of its tag, and a path that's already on the hub (including one that synced before your rule existed) still goes in. The only file that keeps paths out of a release is the committed `.swarmfile/publicignore.yml`. The publish preflight calls out exactly this gap before anything is public: tracked paths your `.gitignore` / `.swarmfileignore` hide that would still be published, by name.

If a file was already synced before you ignored it

Ignoring a path doesn't retroactively unsync it. A file that made it to the hub before your `.gitignore` rule existed — or one created by a teammate's machine that doesn't have the matching rule — keeps its existing hub-side content indefinitely. This is exactly how git treats a file you stop tracking with `git rm --cached`: the ignore rule stops *future* changes from syncing, it doesn't erase what's already there.

To actually remove that old content, run:

```
swarmfile ignore-clean
```

It scans your repository (or a subtree, if you pass a path) for anything already on the hub that your current ignore rules would exclude, shows you exactly what it found before touching anything:

```
3 item(s) already on the hub match your ignore rules, totaling 340.0 MiB:
  node_modules/ (directory)
  dist/ (directory)
  .env.local (file, 2.1 KiB)
This will remove them from the hub for everyone — recoverable from Trash for a while, not instant or permanent.
Proceed? [y/N]
```

and only deletes after you confirm (or pass `--yes`). Deletion goes through the same Trash system as any other delete, so it's recoverable, not instant or permanent — but it does remove that content for *everyone*, not just you.

Be careful running this on a shared repository if a teammate's machine might not have picked up the same `.gitignore` yet, or might be relying on something inside a directory you're about to clean up: `ignore-clean` only knows about *your* machine's current ignore rules, not anyone else's. It won't touch a path currently locked by

someone else — the delete is refused, not silently skipped — but that only catches a file open for editing right now, not one a teammate's tooling reads without holding a lock.

macOS metadata files never sync

On macOS, two kinds of file the system writes by itself stay on your machine without any ignore rule: AppleDouble `._name` files (where macOS keeps a file's Finder tags, comments, and other extended attributes on the drive) and `.DS_Store` (Finder's per-folder view settings). They're meaningless to anyone else, and syncing them would put one extra hub entry beside nearly every file and folder a Mac user touches. They're hidden from listings on the machine that wrote them and never upload. This applies even with sync exclusion turned off, and no whitelist rule can override it.

Two limits keep it from touching anyone's real data. It only applies on macOS: on Windows or Linux, a `._x` or `.DS_Store` you create came from an unzipped archive or an imported repo, so it syncs like any other file. It also only applies when the file is created: one already on the hub stays listed, readable, and synced.

Office owner files never sync

Word, Excel, and PowerPoint write a small `~$name` owner file beside a document while it's open — it records who has the document open, and the app deletes it when the document closes. Swarmfile keeps those files on the machine that wrote them and never syncs them, with no ignore rule: it applies even with sync exclusion turned off, and no whitelist rule can override it. Unlike the macOS files above, this applies on every platform Office runs on. It's also create-time only, the same as the macOS rule — an owner file that already reached the hub stays listed and synced.

One thing this deliberately does not cover: Word's `~WRD####.tmp` / `~WRL####.tmp` save scratch files still sync. Word renames those over the document as part of saving, so keeping them local-only would strand the save; they normally disappear the moment a save completes. If a save fails or Word crashes, a leftover scratch file can show up as an ordinary repository file until you delete it.

macOS safe-save scratch files stay local

When an editor on macOS saves a document, it may stage the save in a scratch folder named `<name>.sb-<8 hex>-<6 letters or digits>` beside the file (TextEdit's save is the usual source: it creates the folder, writes the new copy inside it, renames that copy over the document, then removes the folder). Swarmfile recognises that exact shape and holds its hub registration back long enough for a normal save to remove the folder first, so a routine save never publishes one. This isn't an exclusion rule — it isn't governed by `.gitignore` / `.swarmfileignore`, and no `!` line turns it off.

Like the Office rule above it is create-time only, with the same caveat: a save that stalls past that window, or an app that crashes mid-save, can leave a scratch folder that surfaces as an ordinary repository file — empty and unopenable, named like `report.docx.sb-1a2b3c4d-Ab12Cd` — until the app's next save removes it or you delete it. Listings and the Activity feed hide the shape behind **Show system files** either way.

Git metadata never syncs

A repository's `.git` directory is machine-local metadata, and syncing it is pure churn: packfiles, refs and hook samples change on every git operation, each change becomes its own hub entry and upload, and no teammate's machine can use another machine's checkout state.

So Swarmfile treats `.git` — the directory, or the `.git` file a linked worktree or submodule uses as its gitlink — as **local-only on every platform**, with no ignore rule and with sync exclusion turned off: `git init`, `git clone` and every commit inside a mounted repository stay on your machine, and nothing a current build puts under `.git` is ever uploaded, staged for publishing, or listed to teammates.

No `!` rule can unmask it. Only the exact name `.git` is matched — `.gitignore`, `.gitattributes`, `.gitmodules` and `.github` are ordinary repository files and sync like any other.

One move caveat: a file that is already synced can't be moved *into* `.git`, and a synced entry can't be renamed *to* the name `.git` — the operation is refused (an invalid-argument error), because there is no hub `.git` to move it under and a synced entry can't become local-only. Create or save the file inside `.git` and it stays local, as above.

`.git` content that already reached the hub keeps its stored copy until it is cleaned up, the same as any file that synced before an ignore rule existed. While sync exclusion is on (the default) it is no longer shown to mounts or the web dashboard and stops syncing; with the switch off, pre-existing content behaves exactly as the other built-ins' does — it stays listed and keeps syncing.

Turn sync exclusion back on to hide it and clean it: `swarmfile ignore-clean` lists those `.git` trees as candidates alongside anything your own ignore rules match, so a one-time cleanup removes them for everyone. `swarmfile check-ignore .git/HEAD` reports it as `ignored — excluded by built-in rule (git metadata): '.git'`.

Windows Mark of the Web never syncs

Windows tags a file you download with a `Zone.Identifier` alternate data stream, the "Mark of the Web": it records that this machine got the file from the internet, which is what makes Office open it in Protected View and Windows warn before running it.

That is about how *your* machine got the file, not about the file, so Swarmfile keeps it on the machine that wrote it and never syncs it. Copying a downloaded file onto the drive works as usual, and the stream reads back on that machine, but your mark is never sent to a teammate. Other alternate data streams still sync. Like the rules above, this needs no ignore rule and nothing overrides it.

Unlike them, it also covers marks already on the hub: a stored mark is never copied onto a machine that doesn't already have it, and the hub deletes those stored copies on its own. One a machine already picked up from the hub stays there, though, and just stops syncing: Swarmfile can't tell it apart from that machine's own mark, and removing it would remove a security warning, so it errs toward keeping the warning.

Rules apply across the team, with one caveat

Because `.gitignore` and `.swarmfileignore` are themselves ordinary synced files, everyone mounting the repository ends up with the same exclusion rules once those files have synced to their machine — you don't need to hand-distribute them. The one moment this isn't instant: on a machine that just mounted the repository and hasn't fetched anything yet, there's a brief window where a not-yet-fetched ignore file contributes no rules, so sync proceeds normally until it arrives in the background. This resolves itself within moments, not something to work around.

Checking whether a path is ignored

If a file isn't showing up where you expect — or one you meant to exclude keeps syncing anyway — ask directly instead of guessing:

```
swarmfile check-ignore ./Repositories/node_modules/pkg.js
```

```
ignored — excluded by node_modules ignore rules: `node_modules/`
```

For a path nothing excludes, it just says `not ignored`. The output names both which directory's rules made the call and the exact pattern that matched — useful when a nested `.gitignore` is overriding one closer to the repository root, or a `!negation` is un-ignoring something you expected to stay hidden.

Turning it off

Sync exclusion is on by default. Turn it off from the Desktop App's Settings panel (**Sync-exclude .gitignore'd files**), or set `sync_ignore_enabled` to `false` in that machine's `config.json`, or export `SWARMFILE_SYNC_IGNORE_ENABLED=0`.

With it off, no path is ever hidden from listings, excluded from upload, or refused on read by a `.gitignore` or `.swarmfileignore` rule. The built-ins are not governed by this switch where they act: a newly created macOS metadata file, Office owner file or `.git` entry is local-only whichever way it points, and Windows' Mark of the Web is never synced — but content already on the hub follows each section's own note above.

Flipping this switch — either direction — restarts the engine. It changes how the mount itself behaves, not something that can be swapped out from under it while it's running.

Example

A typical `.gitignore` at a repository's root:

```
node_modules/
dist/
*.log
```

With sync exclusion on, `node_modules/` and `dist/` build up locally as normal — `npm install` and your build both work exactly as they would on disk — but nothing under them ever uploads, and teammates never see those directories appear in their own mount. (`.git/` needs no line here: git metadata never syncs, on any platform — see above.)

An imported dependency layer is the one listing exception. `swarmfile workspace warm --local` records a machine-local `node_modules` as a `local_layer` entry: it is listed on the branch and machine that imported it (so package resolution works), but it is excluded from upload like any ignored path and no other machine sees it. See Coding Agents → Installed dependency trees.

Performance & Disk Usage

Swarmfile is fast by default for the two things people notice — opening files they have touched before, and saving without waiting. When it isn't, it's almost always one of two resources: local disk (the cache) or the network path a file took (LAN peer, seed node, or cloud). This page covers both, plus how to keep the cache from growing into a laptop's free space.

The local cache

Every file you read is copied into a local block cache and kept there until space is needed. The cache is **not** a copy of the repository — it holds what you have actually touched, newest first, and trims itself as it fills.

- **Default size: 10 GiB per open repository.** The engine also has an opt-in shared cap (`SWARMFILE_SHARED_CACHE_CAP=1`) that makes 10 GiB a total across every open repository on the machine.
- Change it in the Desktop App's settings, or from a terminal: `swarmfile cache set 50` (GiB), or `swarmfile cache status` .
- The mount always reports **at least 1 GiB free** to applications, so an app's "disk full" is about your actual local disk, not the cache being full-but-evictable.
- A **seed/NAS node** is the exception: it fetches whole repositories (mirroring them to cloud storage) and keeps them in a warm cache with its own budget. Size `SWARMFILE_CACHE_MAX_BYTES` for the working set you want resident — the sample seed config suggests 100 GiB as a starting point — and remember that a block evicted past the budget simply re-fetches on the next read.

Files you want resident on purpose — a render source tree, a site archive — should be pinned rather than left to the LRU:

```
swarmfile hydrate start ./shots/seq010 --yes   # download and pin (no prompt)
swarmfile hydrate status                       # progress of the current job
swarmfile hydrate release ./shots/seq010       # unpin when done
```

Pinned content can exceed the cache cap; it is never evicted, so don't pin more than the disk can hold.

Making reads fast

Reads come from, in order of preference: the local cache, a **directly reachable** peer that has the file (**LAN first**; then **same-office seed nodes and peers, then other-office seeds and peers**, each bucket sorted by measured round-trip time), then cloud storage (directly from cloud storage, not through the hub). A peer reachable only through the public relay is skipped by default (`SWARMFILE_ALLOW_RELAY_FETCH=1` opts in on both sides). Two things follow:

- **The first person to open a file pays for it; everyone else in the office gets it over the LAN.** If transfers look like they are coming from the cloud when peers exist, check `swarmfile status` for the peer count, then run `swarmfile-doctor` : the `mdns listen` and `host firewall` checks name the usual causes.
- **Erasure coding spreads the load where it applies.** Saved blocks large enough to be sharded (≥64 KiB) on WAN-heavy or cloud-backed saves are Reed-Solomon 10+4, so a read of those can reconstruct from any 10 of 14 shards instead of stalling on one machine, and a slow or offline peer costs redundancy rather than the read.
  - The office LAN path usually fetches the stored chunk directly — erasure coding is skipped when two or more LAN peers already hold it — so this applies to the saves that used it.
  - With deliberate shard placement on a seed-node roster, those shards land on guaranteed homes across the office instead of wherever caching happened to put them. The exact rules are in Storage Format.
- **Pre-hydrate the hot set.** For a render farm or a shoot day, `hydrate start` before the job starts beats streaming during it. `swarmfile uploads --wait` is the matching pre-flight when you need an upload to have landed before reading it elsewhere.

Figure omitted in print (animation): A media repository with a 256 MB master, mounted on a laptop that has fetched none of it: listing the folder costs nothing, and opening the file pays only for the chunk-aligned bytes it touches — the stats view is the meter. - https://swarmfile.com/demo/oss-mount.cast

For a single very large file, ask for the part you need instead of the whole thing:

```
swarmfile fetch ./plate.exr --tail 209715200      # last 200 MB
swarmfile fetch ./plate.exr --follow              # track a moving reader (playback)
```

The engine also **reads ahead** as you go. Video containers (`.mp4` , `.mov` , camera and NLE media) start on the first sequential read and keep roughly 15 seconds of playback buffered ahead, up to 64 MiB; other file types ramp from the third sequential read, once about 2 MiB has been covered, with a 16 MiB ceiling.

Scrubbing is bounded too: a seek drops the window to one small batch and it only widens again after steady forward reading, so dragging a timeline does not pull a full window per stop. A cold, cloud-only first open spends a second or two building that lead before it settles; a warm cache or a LAN peer removes the wait. Tune it in the Desktop App's **Settings → Read-ahead** (seconds; blank = the file type's default — a larger lead keeps high-bitrate playback smoother at the cost of fetching more ahead).

Metadata-heavy workloads (thousands of small files, or an app that stats before it opens) benefit from a larger attribute cache: `swarmfile attr-cache set 10` (seconds; macOS/Linux — the engine restarts to apply it, and Windows mounts are unaffected). CAD tools that open external references can leave xref prefetch on (the default) or turn it off for a pathological reference graph — see Environment Variable Index.

Where the bytes came from

To see that preference order in action — on a live repository, per file — open the Desktop App's **Data sources** page (activity bar → **Data sources**). It shows, for each file and in total, how much of what you read came from **this computer**, from **a colleague's computer** (a directly reachable LAN or WAN peer; relay-only peers are skipped), and from **the cloud**, with first-read times.

A read-ahead line underneath reports how much the engine fetched speculatively, how much of that was actually read, how many seeks (run resets) the window saw, and the file with the most unread bytes — the same numbers `swarmfile doctor` prints as **read-ahead efficiency** (doctor additionally reports the seek-storm share

and its depth, which the page omits); a window that seeked without prefetching shows **Read-ahead idle** beside its seek count.

Reset counters starts a fresh measurement window; the process-wide bandwidth numbers restart with it. A one-line version sits in the header's **Status** popover.

The same numbers are on the command line — the form to use when they need to end up in a spreadsheet or a bug report:

```
swarmfile stats              # per-file table
swarmfile stats --json       # full report: version, machine, mount path/branch, sizes, timings, bytes by source
swarmfile stats --csv        # one row per file for a spreadsheet
swarmfile stats --reset      # zero the counters after printing
```

For a genuinely cold start — a benchmark, a demo, or "is the LAN actually being used?" — `swarmfile demo reset` empties this machine's cache for one repository (the same whole-repository sweep as **Free up space**, including offline pins; never unsynced saves) and zeroes those counters, so the next run measures from zero. Run it on every machine involved; see `swarmfile demo reset`.

Bandwidth and schedules

Throttling is **off by default**. If your office link is shared and Swarmfile's background traffic is crowding out people's calls, turn it on with a real link size first:

```
swarmfile throttle enable \
  --download-mbps 500 --upload-mbps 100 \
  --office-hours-start 08:00 --office-hours-end 18:00 \
  --office-hours-wan-pct 20 --off-hours-wan-pct 80
```

That allows 20% of the link to WAN transfers during office hours and 80% outside them; LAN traffic and app-requested reads are not held back by the schedule. Status and changes: `swarmfile throttle status` / `swarmfile throttle disable`.

The same control lives in the Desktop App: **Settings → General → Limit bandwidth**, and the tray menu's **Bandwidth...** item opens it directly. The same destination answers the `swarmfile://bandwidth` deep link, so a shortcut or a link you send a colleague lands on the throttle card rather than the top of Settings. It is per computer, not per organization.

Two behaviors worth knowing:

- Turning throttling on with **no bandwidth set** applies a guessed symmetric cap (about 100 Mbps), so always set the link size.
- Interactive traffic (opens, small reads, saves' control calls) is marked for priority over bulk transfer (DSCP, when your network honors it). You rarely need to touch the DSCP values; they exist for networks that prioritize by tag — see the Environment Variable Index.

Uploads are designed not to block

A save returns as soon as the bytes are safely queued locally; the upload continues in the background, survives restarts, and resumes by itself. A 500 GB file on a 100 Mbps uplink will take days to reach the cloud, and that is expected — not a stuck sync. `swarmfile uploads` shows what is still in flight (across all machines, not just yours), and `--wait` blocks until nothing is. Other people can fetch the part they need before the upload completes — see While a large file is still uploading.

If a save isn't landing, it is on the **stuck** list, not the upload list: `swarmfile sync stuck` says why and for how long, and the Desktop App's header says **N changes not syncing** or **N changes failed**. Nothing in the queue is silently dropped: a transient hub refusal backs off and retries, while a refusal a person can clear — plan, membership, auth policy, an archived repository, root-create authority — is *held* until the block clears. A removed member's own unsynced saves stay on their machine (ciphertext once the key is dropped) and upload if access returns.

A burst of many small files is a different shape from one large one. Each save needs a handful of control-plane calls, and Swarmfile paces them to the hub's request allowance for your plan, so a mass overwrite of hundreds of files converges over minutes while the header counts up to **Synced**. That is deliberate pacing, not a stall — the work is durable throughout, and a restart resumes it.

Freeing up local disk

In rough order of how much they return:

1. **Let the cache do it.** Lowering the cap (`swarmfile cache set 5`) applies immediately and evicts unpinned content down to the new size, no restart. (`swarmfile cache reclaim` looks similar but isn't a general eviction command: it removes an old cache directory layout that is otherwise only kept as a read-only fallback, and it has a `--dry-run` to preview.)
2. **Release deliberate pins** you no longer need: `swarmfile hydrate release <path>` followed by `swarmfile hydrate free-space <path>` (or **Free up space** in the tray). Free up space never touches unsynced changes.
3. **Remove content that should never have synced.** If a repository synced `node_modules/` or render caches before an ignore rule existed, `swarmfile check-ignore <path>` confirms the rule, and `swarmfile ignore-clean` removes the already-synced content (a soft delete; recoverable from Trash).
4. **Empty Trash and old branches.** `swarmfile trash purge <entry-id>` and `swarmfile branch prune` shrink the **hub's** storage; freeing server-side space is not about your laptop, but it matters for the org's allowance.
5. **Build caches are separate.** Tool caches live in the repository scratch directory, not the block cache: `swarmfile scratch gc` reclaims stale ones, and `swarmfile mounts scratch-dir <id>` prints where they are. See Coding Agents.
6. **Installed dependency trees count too.** A `--local` layer lives in the engine cache under `layers/` (its imported blocks are local-only), and a `--share-deps` tree lives in the repository's `.swarmfile/deps/` and is also hub storage. Remove old `--local` layers with `swarmfile workspace deps gc --kind local`; the import and platform limits are in Known Limitations.

If the Desktop App's disk usage is still surprising, check the size of the cache directory itself (`~/.cache/swarmfile`, `%LOCALAPPDATA%\swarmfile`) and compare it to the configured cap — pinned/hydrated content legitimately exceeds the cap, and `swarmfile hydrate status` lists what is pinned.

Diagnosing "everything is slow"

1. `swarmfile status` — peers in the office, in-flight uploads, commits waiting, stuck saves.
2. `swarmfile doctor` — DNS, hub reachability, mDNS, firewall, and the LAN-consent checks.
3. Check whether the slowness is **first open** (network) or **every open** (local disk or an app doing its own scanning). First opens are expected to be slow for a file nobody nearby has; repeat opens should be cache-speed.
4. Very large repositories: a repository with millions of files and versions can make full-repository browsing and history feel heavier — history retention and repository splitting are in Operations.

Where to go next

- Working with Files — sync status, freeing up space, and watching an upload.
- Working Offline (Pack & Go) — reservations for field work.
- Deployment Topologies — seed nodes and LAN-first office design.
- Environment Variable Index — every cache, streaming, and QoS tunable.

Version Control

Swarmfile versions every file the way a VCS versions code — except the files are whole binaries (video masters, CAD assemblies, GIS rasters) instead of text. There's no meaningful line-level diff for a 200GB ProRes file or a Revit model, so the model is built around whole-file commits and whole-file history instead of line hunks; text files still get read-only line attribution (`blame --lines`) and content diffs (`diff --format=unified|side-by-side`). If you're used to git or Perforce, the mental model carries over; the mechanics underneath don't.

Changesets

A **changeset** is a named, revertable commit that can span multiple files, ordered in a DAG the same way git commits are — you can always roll every file in it back to its pre-commit state as one unit.

A revert or point-in-time restore (`checkout`) covering more than the hub's interactive limit (about 5,000 entries) runs as a durable background job: the CLI waits for it by default (Ctrl-C or `--no-wait` detaches with exit `3` , followed with `swarmfile op status`), and the result appears in the log once it lands. Whether the underlying saves actually land *atomically* (all at once, invisible until they do) depends on which of the two modes below produced it.

Swarmfile supports two ways of producing changesets, and you pick the one that matches how you work.

Mode A: auto-landed (default)

By default, your saves land as commits automatically as you work, and immediately: each save is live on HEAD the moment it uploads, visible to teammates right away — it isn't held back waiting on anything else.

What Swarmfile groups is the *changeset* those saves get filed under: consecutive saves are batched into one open changeset while you're actively working, and that changeset closes and commits once saves go idle for `changeset_idle_secs` (default 5 minutes, configurable via `SWARMFILE_CHANGESET_IDLE_SECS` or the config file). A burst of edits becomes one named, revertable commit; the next burst after a break becomes another.

You don't open or close anything yourself — this is the lower-friction mode and it's what most people want most of the time. One exception: a save whose bytes are identical to the version already on HEAD files nothing — there's no change to commit, so it doesn't create a new version (an app that rewrites a file unchanged on close therefore can't spam history).

```
swarmfile commit -m "Color grade pass 2 on interview B-roll"
```

`commit` opens and submits a changeset in a single step — the Mode A pattern, useful when you want to attach a specific message to a specific save rather than relying on the automatic one.

Mode B: staged, explicit submit

For work you want to land as one deliberate, reviewed unit — a multi-file asset swap, a batch of renders that only make sense together — open a changelist, stage saves into it, and submit (or cancel) the whole thing as one atomic transaction. It's the same shape as a git index: nothing is visible to teammates until you submit, unlike Mode A where each save is already live the moment it happens.

```
swarmfile changelist open -m "Swap all Act 2 plates to final grade"
# ...work happens, saves are staged into the open changelist...
swarmfile changelist status
swarmfile changelist submit
```

`changelist submit -m "message"` sends the commit message at submit time (git-style); without `-m` the group's open-time label is kept, and a group with neither is committed without a message. Submitting checks write access to every file in the changelist at that moment, so a save staged before a permission change doesn't bypass it; a refused submit leaves the changelist staged, nothing lost.

If you change your mind partway through, `swarmfile changelist cancel` discards the staged changes instead of submitting them.

On the Desktop App, the **Changes** tab shows the same work with a git-style commit box: type a summary (and an optional description) and press **Publish**. The message is required, just like a git commit, and is sent with the publish. A group's open-time label, if it has one, prefills the summary. The switcher's per-row publish (acting on another open group) sends no message — that group keeps its label, or is committed without a message when it has none.

Multiple changelists at once

You're not limited to one open changelist. Open a second while the first is still open, and each save stages into whichever one is *current* — the same idea as Perforce's `-c` flag:

```
swarmfile changelist open -m "Swap all Act 2 plates to final grade"
swarmfile changelist open -m "Quick fix: wrong LUT on shot 12"
# saves now stage into the second changelist, since it's current
swarmfile changelist list
#   #a1b2c3d4 "Swap all Act 2 plates to final grade" — 6 file(s) staged
# * #e5f6a7b8 "Quick fix: wrong LUT on shot 12" — 1 file(s) staged
swarmfile changelist current a1b2c3d4
```

`changelist list` shows every open changelist with a `*` marking which one is current; `changelist current <id>` switches. This lets you juggle two unrelated in-flight edits without submitting one to start the next — no need to interrupt a large staged batch just to make an unrelated quick fix.

If a file was staged into the wrong changelist, move it without unstaging and restaging by hand:

```
swarmfile changelist reopen "Renders/Act2/shot12.mov" --to e5f6a7b8
swarmfile changelist reopen --to e5f6a7b8 --from a1b2c3d4
```

The first form moves a single staged file; the second (no path) moves every staged file from one changelist to another in one atomic step. Either can target a specific changelist without switching to it first — `submit` / `cancel` also both accept `--id <changeset>` for the same reason.

Setting work aside

If you need to leave the branch or repository but aren't ready to submit or cancel, **park** the work instead — it's set aside and kept alive on the hub, then restored when you come back to that scope or explicitly:

```
swarmfile changelist park       # set every open changelist aside
swarmfile changelist parked     # list parked work, with age and file count
swarmfile changelist resume     # restore parked work for this branch
swarmfile changelist discard    # abandon it (asks first; --yes skips)
```

Parked work is a **set, not a stack**: `resume` restores *all* parked work for the branch, and `discard` abandons all of it (or one, with `--changeset <id>`). A branch or `workspace switch` into a scope with staged work offers "Park for later", which does the same thing before the scope flips; the Desktop App's **Parked changes** panel lists, resumes, and discards the same work.

If you're coming from git, `swarmfile stash` maps here: `stash` → `park`, `stash pop` → `resume`, `stash list` → `parked`, `stash drop` → `discard`.

Viewing history

`swarmfile changesets list` shows the commit log, newest first. `swarmfile log` gives you the same history as a compact colored feed, closer to `git log --oneline`, when you just want a quick scan.

Commit ids

A Swarmfile commit's id is Swarmfile's own hash (64 hex characters), and that is what the web commit log and `swarmfile log` show. It is **not** git's SHA-1: a `git clone` of the same repository builds git commits from these commits, and each git commit's message carries a `Swarmfile-Commit: <id>` trailer naming the Swarmfile commit it came from.

A hash in `git log` therefore won't match the one on the Commits page — that's expected, and the trailer is the mapping. When a `git push` reported its git commit id, the **Commits** page shows that SHA-1 on a second line under the Swarmfile id (`git 2a8c9d6…`), marked *unconfirmed* until a second principal's report confirms it. See What a clone contains.

Checkpoints

In the default auto-landed mode, saves are collected into one open changeset and committed after a quiet period (5 minutes; `SWARMFILE_CHANGESET_IDLE_SECS` overrides it). `swarmfile checkpoint` closes the current batch **now**, giving you a commit at the point you choose — before a risky refactor, or after each agent turn:

```
swarmfile checkpoint
swarmfile checkpoint -m "green build"
```

A checkpoint is an ordinary commit: it appears in `swarmfile log`, carries the branch history, and is restored with `swarmfile restore` like any other. On an `agent/<label>-<6 hex>` workspace the quiet period is 30 seconds (`SWARMFILE_AGENT_CHECKPOINT_IDLE_SECS`, `0` disables the shorter window), so an agent's work is checkpoints, not one giant commit. A staged changelist (Mode B) is deliberately not submitted by `checkpoint` — submit it with `swarmfile commit`.

Checkout and restore

`swarmfile restore` restores a scope to how it looked at a past commit — the whole repository, a folder, or a single entry:

```
swarmfile restore 482 --path "Renders/Act2/final.mov"
swarmfile restore --to 482          # the same ref; the checkpoint spelling
swarmfile restore HEAD~3
```

The commit can be a number (`#N` from `swarmfile log`), `HEAD`, `HEAD~N`, a branch, a tag, a changeset id, or a 64-hex commit hash; a prefix (`seq:` / `head:` / `branch:` / `tag:` / `changeset:` / `hash:`) forces which kind is meant. `swarmfile switch <branch>` is the equivalent for changing which branch the mount works on. Both are the preferred, explicit commands; `swarmfile checkout` still exists as a git-habit shim that overloads the two the way `git checkout` does — a branch name switches, a commit number/`HEAD` / `HEAD~N` restores, and the `checkout --at-seq <N>` form still works — but `switch` / `restore` say exactly which one you mean.

This doesn't rewrite or destroy history — it creates a new commit that matches the old state and adds it to the log. If the restore turns out to be wrong, you can restore again to undo it; nothing is ever permanently lost by restoring.

Trash and per-file rollback

Separate from the changeset system, every entry also has its own lightweight history: a per-user trash for soft-deleted files with a configurable retention window, and point-in-time rollback for individual files. It's simpler than a changeset — no multi-file atomicity, just "what did this one file look like at this timestamp." See Trash, History & Rollback for the full picture.

Where to go next

- Branches & Merging covers forking a branch to isolate risky work and merging it back.
- CLI: swarmfile has the full command reference for `commit`, `changelist`, `changesets`, `log`, `switch`, and `restore`.

Branches & Merging

Swarmfile branches follow the Perforce-streams model, not git branches. A branch is a full working line over the repository's content-addressed storage, not a lightweight pointer into a shared history graph — but because storage blocks are content-addressed and shared, forking a branch doesn't copy any file data. Only the metadata diverges, and only once you actually change something. Fork a 4TB repository and the fork costs you nothing until you start editing.

Branches are repository-scoped

A branch applies to an entire repository, not a folder inside it. You don't branch a subdirectory and leave the rest of the repository on main — every mount of the repository works one branch, in full, independently of every other mount. This keeps the model simple: there's never a question of which branch a given folder is "really" on.

Creating, listing, and archiving

```
swarmfile branch create look-dev-warm --from main
swarmfile branch list
swarmfile branch archive look-dev-warm
```

`branch create` forks a new branch, defaulting to a fork from **the branch this mount is currently working on** if you omit `--from` (so from inside a branch, `branch create` forks that branch — use `--from main` to fork main explicitly). `branch list` shows the active branches on the repository. `branch archive` soft-archives a branch you're done with — the repository's default branch (normally `main`) can't be archived.

A branch create (or a merge) is refused while a file on the source branch is still uploading, so it never silently takes that file's previous version. Retry once the upload finishes. If the uploading machine crashed, see Troubleshooting.

Pruning merged branches

After a batch of branches has landed, `branch merged` lists the ones with nothing left to bring over — a branch-vs-base diff with no changed files and no conflicts (the same computation `swarmfile diff <branch>` shows, so a branch whose diff can't be computed is left out rather than assumed merged). `branch prune` archives all of them in one pass, after naming them and asking; `--yes` skips the prompt for scripts.

```
swarmfile branch merged
swarmfile branch prune
```

`prune` never archives a branch an open mount on this machine is working on — if you're checked out to a merged branch, or another mount here is on one, it's skipped (and `merged` marks it). A protected branch is listed, but archiving it is refused unless you're an org admin (exit 2). (A teammate's mount on another machine isn't visible to this check.)

"Merged" here means *merged into its own base* (the branch the diff is computed against), not specifically into `main` — the diff route only compares a branch to its base, so there's no `--into <branch>` form.

All three operations are also available from:

- **The web dashboard** — the Commits view lists, creates, and archives branches, and can trigger a merge; the **Branches** tab (see Protecting many branches at once) shows every branch's protection status plus protection rules. Both of those dashboard tabs are admin-gated.
- **The Desktop App** — the Branches page does the same, applied immediately, with no restart required for list, create, or archive.
- **The CLI** — `swarmfile branch {list,create,switch,archive,merged,prune,protect,unprotect}`, `swarmfile protection-rule {list,create,delete,add-reviewer,remove-reviewer,list-reviewers}`, `swarmfile tag {list,create,delete}`, `swarmfile merge`, `swarmfile mr {create,list,status,approve,request-changes,review-comment,dismiss-review,merge,ready,close,reopen,comment,comments,assign,unassign,request-review,unrequest-review,label,unlabel}`, and `swarmfile label {list,create,update,delete}`.

A typical use: fork a branch to try a look-dev variant or a design option without duplicating the whole repository, then merge it back if it works out, or archive it if it doesn't.

Branching from a past commit, or a tag

By default, `branch create` forks the source branch as it is *right now*. Give it a commit ref instead (any of the forms below), and it forks the source exactly as it stood at that point — every file carrying the content it had then, in one call, instead of forking at head and then rolling the fork back with `checkout`:

```
swarmfile branch create recover --from-commit 482
swarmfile branch create recover --from-tag v1.0
```

`--from <REF>` takes the full ref vocabulary — a branch, tag, `#N` / `HEAD~N`, changeset id, or 64-hex commit hash — so `--from main` forks another branch, while `--from-commit` / `--from-tag` are the explicit seq/tag spellings. `--from` is optional with either: the targeted commit or tag already names which branch it lived on. If you give `--from` anyway and it disagrees with that branch, the command refuses rather than silently picking one. The fork keeps the state it started from: files inherited from an older point stay readable even after that commit ages past history retention.

This is the primitive a headless or unattended machine — a render farm node, a CI runner — uses to pin itself to an exact historical state with a single command and no Desktop App or browser interaction:

```
swarmfile commit -m "submit render job 482"
swarmfile tag create v1.0
swarmfile branch create render-fix --from-tag v1.0
```

A **tag** is a repository-scoped, immutable name for a commit. It also **pins** that commit against history retention: while the tag exists, checkout/restore to it keeps working even after the commit would otherwise have been pruned, and the content it names stays stored; deleting the tag releases the pin. There's no "move a tag" command in the `swarmfile` CLI; to repoint a name, delete it and create it again (with git access, a lightweight tag can also be moved with `git push -f origin <tag>` — see Cloning a Repository with Git — unless a release was published from it):

```
swarmfile tag list
swarmfile tag create v1.0
swarmfile tag delete v1.0
```

`tag create` defaults to the branch's CURRENT head — exactly what the commit above just produced, no need to look up its commit number first. Give it `--at-seq <n>` explicitly to tag an older commit instead of the head:

```
swarmfile tag create v0.9 --at-seq 482
```

The web dashboard's Commits view offers the same two actions per commit row — "Branch from here" and "Tag" — plus a tag picker next to the branch selector; the Desktop App's Branches page has a source-type picker (branch / commit number / tag) on its create form, and its own Tags page for list/create/delete.

Switching branches

```
swarmfile branch switch look-dev-warm
swarmfile branch switch     # omit the name to switch back to the default branch
```

> Switching branches is hot — no engine restart, no interruption to the mount. The first time a machine visits a branch, switching to it pays a one-time full sync (proportional to that branch's size); a repeat visit to a branch it's already synced is a fast incremental catch-up. Switching is refused while a changelist is open — submit or cancel it first, so staged work is never silently discarded or reattached to the wrong branch. Files open in applications are closed as part of the switch, and their pending changes are saved to the branch you're leaving; an application that keeps writing to such a file afterwards gets an error, and nothing it writes lands on the new branch.

List, create, and archive don't touch the running mount and take effect immediately.

The web dashboard has a separate, metadata-only branch switcher on the Files tab — a GitHub-repo-browser-style `⎇ main` dropdown that scopes which branch's file tree the dashboard is showing you. It never touches a mount: switching it there has no relationship to, and no effect on, which branch any desktop client is actually checked out to. Use it to browse another branch's files from the browser without needing a machine checked out to it.

Merging

```
swarmfile merge
swarmfile merge --record-conflicts
```

Merging a branch

`merge` performs a whole-file three-way merge of your mount's current branch back into its parent — the branch it was forked from. That is usually `main`, but a branch forked from a non-main branch merges back into *that* branch.

- **Direction is implicit.** There is no target flag: a branch knows its parent. To land a branch you are not standing on, name it: `swarmfile merge --from feature-x` merges `feature-x` into its parent without switching mounts.
- **Binary files are winner-take-all.** There is no line-level merge to fall back on; a file changed on both sides since the fork point is a conflict.
- **Conflicts are listed by default.** `--record-conflicts` instead records rows you can resolve interactively, one file at a time.
- **Resolution is scoped to the target.** A recorded conflict belongs to the branch being merged into: `local` is the incoming branch's version and `remote` is the version already on the target; `--winner remote` keeps the target's copy. The next `merge` then lands cleanly, because the source branch is updated to the kept version.

The branch index

Because a branch inherits its files instead of copying them, the hub keeps an index of what each branch inherited. On a very large repository that index is built in steps the first time it is needed:

- A commit, `git push` or file lookup can briefly answer `fork_index_building`. That is temporary; the answer carries a retry interval and the Desktop App keeps the commit, retrying without showing an error.
- A branch that inherited more than the hub can index answers `fork_too_large`. That one is final: no smaller commit on that branch would land either. The Desktop App tells you once per branch, keeps your saves, and suggests asking the repository owner or working on the default branch.

For text conflicts, the engine can try a three-way line merge before you pick a side: `swarmfile merge --diff3` batch-resolves every text-file content conflict it can, then retries the merge once. A repository can make that the default by committing `.swarmfile/merge.yml` with `diff3: true` (see Repository-local files); `--no-diff3` forces it off for one invocation. Anything it can't merge stays a conflict for the resolution flow below, and the web dashboard's merge button offers the same auto-resolve step when a merge comes back conflicted.

Figure omitted in print (animation): Disjoint edits to one file from two mounts on one branch: the second save is refused against the version it read, and one non-interactive line three-way merges both edits onto the same history — no human, no lost work. - https://swarmfile.com/demo/oss-merge.cast

Two more flags help on big or cautious merges. `swarmfile merge --dry-run` prints exactly what would change — added, modified, deleted, renamed, moved, and conflicting files, the same read-only diff `swarmfile diff` shows — and stops without staging anything or taking the branch's merge lock.

`swarmfile merge --wait` blocks until a large background apply finishes, printing progress phase by phase (the command otherwise returns as soon as the merge is accepted); it gives up after ten minutes with exit 2.

And if the target branch moves after the merge was prepared, the merge refuses with `stale_merge` instead of overwriting the newer state — run `merge` again and it lands cleanly.

How this holds up on a very large repository

Branching never copies file content: blocks aren't copied or re-read, only a little metadata per folder, and that metadata is written in parallel. So forking stays fast as a repository grows — thousands of folders are proportionally more metadata, not more file bytes — and the first time a new branch is opened or synced, its folders are likewise read in parallel. On a 100,000-entry tree, a fork that creates 120+ directories completes in tens of seconds.

Because a branch inherits its files instead of copying them, the hub keeps an index of what each branch inherited. On a very large repository that index is built in steps the first time it's needed, so a commit, a `git push` or a file lookup on a new branch can briefly be answered with `fork_index_building`.

That is temporary: the answer carries a retry interval, and the Desktop App keeps the commit and retries it on its own, with no error shown.

A branch that inherited more than the hub can index answers `fork_too_large` instead, and that one is final: no smaller commit on that branch would land either. The Desktop App tells you once per branch that commits there can't be recorded, keeps your saves, and suggests asking the repository owner or working on the default branch.

Merging is metadata-only in the same way. A merge larger than the single-pass staging budget continues as a resumable background job: the command accepts it, the branch's merge status and the Desktop App show it planning and then applying (phases and rows remaining), `swarmfile op status` shows the underlying job, and the target head advances once, when it finishes.

Working out *what* a merge changes is kept short too, so one big merge never makes everyone else on the repository wait.

A merge is worked out on the spot when the target and the branch together hold up to about 5,000 entries.

It is also worked out on the spot when the branch is small even though the repository is large, under three conditions: the branch holds about 1,000 entries or fewer, working out its changes reads no more than about 30,000 entries, and the target and branch together stay under 70,000.

Background merges

Any other merge is planned in the background: `swarmfile merge` and a Merge Request's **Merge** action accept it (`202`), the merge status shows it planning before it applies, and a Merge Request flips to merged once it lands, with its approval, CI and source-head gates checked again at that moment.

A read-only preview of a merge that size, `swarmfile diff <branch>` or a Merge Request's changed-file list, is refused with `merge_over_sync_budget` ("too large to preview") rather than computed while you wait; opening the Merge Request and merging it still work. Approving one you can't preview asks you to confirm that you are approving changes you haven't seen.

Merging a Merge Request that size runs in the background from the web dashboard; an older Desktop App or CLI is told the merge is too large and to use the web dashboard or upgrade. While a background merge is still applying, its Merge Request stays open; if the merge is rolled back, the request stays open and its history notes why.

Automatic conflict resolution (`merge --diff3`, `mr merge --diff3`) refuses the same way at that size; resolve those conflicts one file at a time instead.

Editing the source branch while a merge runs. A merge lands the source branch as it was when the merge was prepared. If the source changes before any file has been applied to the target, the merge is rolled back with nothing applied and refused with `stale_merge`; run it again.

Once files have started landing on the target, the merge finishes rather than stop half-applied, and edits made to the source after that point are not part of it. The source branch is then kept, even if the merge request asked to delete it, so those edits are not lost, and the merge request gets a note saying the source changed while the merge was being applied. Merge the branch again to bring the later edits in.

Canceling a background merge. A merge that is still being planned in the background, with nothing applied yet, can be canceled through the hub API: `POST /merge-requests/:id/merge/cancel` for a merge request, or `POST /branches/merge/cancel` with the source branch as `from` for a direct merge. The person who started the merge or an org owner or admin can cancel it; anyone else gets `forbidden`.

A canceled merge request stays open and its history notes the cancel, and a canceled direct merge reports `merge_cancelled` in its merge status.

A merge that is already landing can't be canceled and answers `merge_in_progress`: let it finish, or have a repository owner roll it back with the repair action (see A merge won't finish). When no background merge is waiting, the cancel answers `invalid_state`.

One background merge per branch at a time. While a merge of a branch is being planned in the background, only the person who started it can merge that branch again (a retry of the same merge). Anyone else who merges the same branch, even a small merge that would otherwise land right away, gets `merge_plan_busy` until the background merge lands, is canceled, or goes idle for a few minutes. The same applies between a merge request's background merge and a direct merge of its source branch. Wait for it to finish, then merge again.

Long-running operations

Long-running work doesn't depend on the terminal staying open. `merge`, `commit`, `checkout`, `revert`, `branch create` / `archive`, `mr merge`, trash `restore` / `purge`, `repository delete`, `materialize`, `ignore-clean`, `hydrate free-space` and closing a mount run as **operations** in the engine: the command waits and prints the result as usual, however long it takes.

Press Ctrl-C and it detaches instead of stopping — the command exits `3` and prints the operation's id — and `--no-wait` returns the id at once; `swarmfile op list` / `op status` / `op wait` / `op cancel` collect the outcome later, even after the engine restarts. (Ctrl-C cannot stop a commit, merge or delete part-way: those complete or fail on their own, and `op cancel` refuses them.)

Commands not on that list use their own route's socket deadline — five seconds for most, up to three minutes for a few — and if one reports `control socket request failed: no answer within …s`, it stopped waiting but the work usually continues or is already done. Check `swarmfile branch list` (or `swarmfile mounts list`) before retrying: re-running `branch create` for an existing name answers that the branch already exists, which means the first one succeeded.

Merge Requests

`swarmfile merge` lands a branch immediately — useful when you're merging your own work and you're confident it's ready. When you want someone else to look at a branch before it lands, open a **Merge Request** instead — from the web dashboard's **Merge Requests** tab, **Merge requests** on the Desktop App's activity bar, or `swarmfile mr create --title "..."`: pick the branch, give it a title, and it's posted for review.

What a review shows

A Merge Request shows reviewers exactly what would change — the same list of added, modified, deleted, renamed, and moved files a direct merge would land, computed fresh every time you open it, so it's never a stale snapshot.

Click a changed text file to expand an inline diff right there, with syntax highlighting and word-level highlighting on modified lines; a file that conflicts with the target branch is flagged right in that list too, and clicking it shows a 3-way diff instead — your changes and the target's changes, each compared against the version both branches started from, so you can see exactly what's colliding before resolving it.

Reviewers leave an **Approve** or **Request changes** verdict (optionally with a note); only the *latest* verdict from each reviewer counts, so changing your mind is just casting a new one.

An approval is also tied to the source-branch commit it was cast against: pushing a new commit afterwards retires every earlier approval, so the approval floor has to be met again on the new head.

A "request changes" verdict is different: a new commit does **not** lift it (see below).

Merge only becomes available once the approval gate is met and nobody's latest verdict is "request changes". On an unprotected target that gate is zero approvals; protecting a branch is what sets a floor, starting at one approval and configurable up to ten (a branch matched by a protection rule uses the rule's count, and when both apply the effective count is the higher of the two).

There's still no required-*specific*-reviewer *enforcement* — raising the count (below) is the whole gating mechanism, not naming who has to be one of the approvers; requesting a specific reviewer (below) is a nudge, not a requirement — anyone's approval still counts toward the floor, requested or not.

Every merge request also carries its own discussion thread — the same kind of comments you'd leave on any file — for review remarks that aren't about one specific line of the diff.

An approval is tied to the source branch's **content** as well as its commit, because not every change makes a new commit. Any change on the source branch after an approval retires it: saving a file (even a save that only touches its timestamp), renaming, moving, deleting or restoring one, permanently deleting one from trash, or editing a folder-profile field value such as "Payment Ref".

An approval therefore always covers what is on the branch at merge time; if anything changed after it, the approval floor has to be met again. The check is deliberately strict, so a change that leaves the files looking the same still counts. Approvals cast before Swarmfile tracked content this way no longer count, so an open merge request may need its approvals cast again.

A "request changes" verdict is not lifted by those content changes, or by a new commit: it keeps blocking until that reviewer casts a new verdict (an approval, or another request for changes; a Comment doesn't replace it) or a maintainer dismisses it. If the reviewer later approves and that approval goes stale, the earlier "request changes" doesn't come back; the approval simply stops counting.

When you approve from the web dashboard, the approval is also tied to the version of the branch the page showed you: if the branch changed after you opened the merge request, the approval is refused with `source_changed` and the page reloads so you can review the new changes first. Comments and "request changes" are never refused this way.

There's also a third, non-blocking **Comment** verdict (`swarmfile mr review-comment <number>`) for feedback that isn't a vote either way — it shows up in the review history same as Approve/Request changes, but never counts toward the approval floor and never blocks a merge. Unlike the other two, a Comment is allowed on your own merge request, since it can't be used to rubber-stamp your own work the way a self-approval could.

Merging a merge request

Merging a Merge Request does exactly what `swarmfile merge` does — same conflict detection, same all-or-nothing landing, same optional "record conflicts for resolution" fallback (and the same `--diff3` auto-resolve override) if the branch drifted since anyone last looked at the diff. It also pins the source branch: if the head moved after the approval gate and CI check ran, the merge refuses with `stale_merge` rather than landing a commit the checks never saw.

A Merge Request that's approved but hits a conflict on merge still 409s, same as a direct merge would; nothing lands until the branch is fixed. Closing one abandons it without merging — nothing about the branch changes — and a closed (never-merged) request can be reopened later if it turns out the work wasn't actually abandoned, from the web dashboard, the Desktop App, or `swarmfile mr reopen <number>`.

Merging also offers an opt-in **delete the source branch** checkbox (`swarmfile mr merge <number> --delete-branch`) — unchecked by default. It only takes effect once the merge itself has actually landed, and archiving is a soft `branch archive`, never a hard delete; the same protected-branch admin gate applies, so if the source branch got protected in the meantime and you're not an admin, the merge still succeeds but the branch stays as-is — you'll see a note saying so rather than the merge itself failing.

Any repository member can open, review, and merge a Merge Request — it isn't admin-gated, unlike RFIs. The one restriction is that a merge request's creator can't review their own — the hub refuses with `self_review` regardless of which surface asks (web, Desktop App, or CLI); the web dashboard and Desktop App both gray out Approve/Request changes on your own merge request rather than letting you find out only after submitting.

Dismissing a review

A "request changes" blocks a merge until the reviewer who cast it reviews again, the way GitHub works: pushing a fix, even a one-character commit, doesn't lift it, so the reviewer gets to check the fix. When that reviewer is away, an org **owner or admin** can dismiss the review instead: **Dismiss review** next to it in the web dashboard's review history, or

```
swarmfile mr dismiss-review 3 <review-id> --reason "Reviewer is on leave; the requested fix is in the latest commit"
```

(the review id is `reviews[].id` in `swarmfile mr status 3 --json`). A reason is required, and it's shown on the merge request beside the dismissed review and recorded in the activity feed; the dismissed reviewer and the merge request's author are notified. A dismissed review stops counting entirely: it no longer blocks, and if it was an approval it no longer counts toward the floor. The reviewer's earlier verdicts don't come back either; they have a say again by reviewing again. An approval can be dismissed the same way, a Comment can't (it never gates anything), and a dismissal can't be undone.

Only an org owner or admin may dismiss (a repository member who can merge can't; the hub answers `dismiss_forbidden`). They can dismiss on any open merge request, their own included: a maintainer can already change the target's branch protection, so refusing them would protect nothing, and it means an org whose only maintainer opened the merge request is never stuck. The reason is required, and the merge request shows who dismissed the review and why. The merge request has to be open, and dismissing the same review twice is harmless (the second dismissal changes nothing and notifies nobody).

A review also stops counting on its own when its reviewer no longer has a seat that can write in the org: they left the organization, or (for an API key) the key was revoked or expired. A request for changes from them stops blocking, and an approval from them stops counting toward the approval floor. Both stay in the review history.

Anyone with access to the repository can still leave a review; what counts is one from a current member, admin, or owner. If Swarmfile can't check the reviewer's seat at that moment, it errs on the safe side: requests for changes keep blocking and approvals don't count until the check succeeds (try again shortly).

Only one merge request can be open for the same source and target branches at a time. Opening a second one is refused with `merge_request_exists`, and the refusal names the merge request that's already open (`existingMergeRequest`), so the review continues there instead of starting over on a parallel one. Reopening a closed merge request is refused the same way while another one for those branches is open.

Carried-over reviews

Closing a merge request doesn't wipe the slate. When you open a new merge request for the same source and target branches, or reopen an older closed one, every request for changes still standing on any merge request for those branches that was closed since the last one that merged carries over, marked as carried over, and blocks it the same way until that reviewer reviews again or a maintainer dismisses it.

"Still standing" means it's that reviewer's latest verdict across those closed merge requests and it wasn't dismissed: a request for changes the reviewer later replaced with an approval doesn't carry, and dismissing a carried copy dismisses it for good.

Nothing carries over from a merge request that was merged, and a later merge clears a request for changes only when it merged the reviewed work: a merge from the same source branch the request was cast on, or a merge request that carried that request and had it resolved or dismissed before merging. Either way the request for changes must have been cast *before* that merge: one cast after it (say, on an older merge request reopened after a later one merged) still carries. A request for changes from the new merge request's own author isn't carried (they can't review their own merge request).

"The same source branch" means the same branch **or a branch with the same name**. Archiving `feat-x`, creating a new `feat-x` from the target and opening a merge request from it still carries the requests for changes from the earlier `feat-x` merge requests into that target. The name that counts is the one the source branch had when each merge request was opened, so renaming a branch later doesn't shed its blocks either.

A contribution proposed from a fork works the same way: its upstream branch is named after the contributor's repository and branch, so a resubmitted proposal from that branch inherits the maintainers' standing requests for changes, even though the earlier proposal's branch was retired when its review ended.

A merge from a *different* branch that only shares the name never clears a block: renaming a blocked branch away, creating a new branch under its old name and merging that leaves the original request for changes standing, both when the original merge request is reopened and when a new one is opened from the original branch.

The limit: changes pushed under a brand-new branch name start a new line of review and carry nothing, so a reviewer who wants to keep blocking reviews the new merge request.

Branch names starting with `contrib/` are reserved for contributions proposed from forks. Creating such a branch is refused, and so is renaming a branch into or out of that namespace or restoring an archived `contrib/` branch (`contrib_branch_reserved`), so a proposal's review history can't be taken over by a rename or a restore.

Reopening a closed merge request that's older than the last one merged from that source (by branch or by name) doesn't bring back the requests for changes it held from before that merge: the merge landed the reviewed work, so those requests are dismissed automatically when it reopens, with the reason naming the merge request that merged. A request for changes cast in the same second as that merge keeps blocking, and the reviewer can always request changes again on the reopened merge request.

A review is recorded only while the merge request is still open: if it's closed at the moment the review arrives, the review is refused (`invalid_state`) rather than written to the closed merge request, so it can't miss being carried over to the next one.

Draft merge requests

A Merge Request doesn't have to be ready for review the moment it exists. Pass `--draft` to hold it back:

```
swarmfile mr create --title "Warmer facade material" --draft
swarmfile mr ready 3
```

A draft behaves like any other merge request for everything except merging and notifications at creation time: comments, reviews, assigning, requesting a review, and labels all work exactly the same, and assigning someone or requesting their review BY HAND still notifies them even while the request is a draft — it's only the initial "here's a new merge request" fan-out, and an auto-reviewer's request-review notification, that are held back at creation, on the assumption a draft isn't actually ready for anyone's general attention yet.

That fan-out isn't deferred to later, either — it's skipped for good: marking a draft ready doesn't retroactively send the notification it held back at creation, so use assignees or requested reviewers if you need to be sure someone specific gets pinged.

What's actually gated is merging: `mr merge` refuses a draft outright, even one that's already picked up an approval. `mr ready 3` (also available from the web dashboard and Desktop App detail view) flips it to a normal, reviewable request. There's no way back to draft once it's marked ready.

Assignees, requested reviewers, and labels

Beyond the approve/reject verdicts above, a Merge Request carries three pieces of bookkeeping — all purely organizational, none of them gating whether `mr merge` is refused:

- **Assignees** — who's on the hook to land it. One or more people, or a group.
- **Requested reviewers** — who's been asked to take a look. Distinct from an *assignee* (who lands it) and distinct from actually having *reviewed* (the Approve/Request changes verdicts above) — requesting someone is a heads-up, not a requirement, so it never changes who has to approve for `mr merge` to unlock.
- **Labels** — a per-repository set of tags an admin defines once (`swarmfile label create bug --color "#FF5733"`), then anyone can attach to any merge request in that repository to filter the list by.

Requested reviewers don't have to be added by hand every time. A branch protection rule can name a set of **auto-reviewers** — CODEOWNERS-style, per target branch — and every merge request opened against a branch a rule matches gets those reviewers requested automatically the moment it's created.

An auto-requested reviewer is attached exactly like a manually-requested one, and only ever adds to the list — it never stops anyone else from requesting a different reviewer by hand.

The one difference is notification timing on a draft: a manual request-review always notifies, but an auto-reviewer added at creation time on a **draft** MR is silently attached with no notification (same reasoning as the "new merge request" fan-out above) — they show up as a requested reviewer either way, just without the ping until the MR is marked ready and something else notifies them. See Protecting many branches at once for how to configure the list.

All three are editable from the web dashboard's detail-view sidebar, or the CLI:

```
swarmfile mr assign 3 <principal-id>
swarmfile mr unassign 3 <principal-id>
swarmfile mr request-review 3 <principal-id>
swarmfile mr unrequest-review 3 <principal-id>
swarmfile mr label 3 <label-id>
swarmfile mr unlabel 3 <label-id>
swarmfile label list
swarmfile label create bug --color "#FF5733"
swarmfile label update <label-id> --name defect --color "#000000"
swarmfile label delete <label-id>
```

Assigning, requesting a review, and attaching a label are all idempotent (doing it twice is a no-op, not an error) and don't require anything from the target — there's no accept/decline step, since these are informational, not access grants. Defining, renaming, or deleting a label is admin-gated; attaching an existing label to a merge request isn't.

From the Desktop App

Merge requests on the activity bar opens a page with a list (filterable by status, refreshing automatically as teammates act — and re-checking periodically while it's open) that drills into a detail view for one request: branch names, the changed-file list, the full review history, the discussion thread, and a review composer — click through to Merge or Close from there, both behind the same confirm step every other consequential Desktop App action uses.

Same as the web dashboard, click a changed text file for its inline diff, or a conflicting file for a 3-way diff. A **+ New Merge Request** button on the list opens a small inline form (source branch, title, optional description, and a "Create as draft" checkbox) rather than needing the CLI or the browser just to propose a review.

A draft shows a **Draft** badge in place of the usual approval pill, and its Merge button is replaced by **Mark ready for review** until you flip it. If a target branch requires CI checks to pass, a merge blocked purely on a failing/missing check says so specifically ("Required checks haven't passed yet") rather than the generic "needs approval" message. Assigning, requesting a review, and labeling are done from the web dashboard or the CLI — the Desktop App shows them read-only.

From the CLI

```
swarmfile mr create --title "Warmer facade material"
swarmfile mr list
swarmfile mr status 3
swarmfile mr approve 3
swarmfile mr request-changes 3 --note "needs another pass"
swarmfile mr merge 3
swarmfile mr close 3
swarmfile mr reopen 3
swarmfile mr comment 3 --message "Looks good once the fascia texture is swapped."
swarmfile mr comments 3
```

`mr create` 's `--from` defaults to whatever branch the mount is currently on, so a CI job that forked its own branch and did the work can open a review with no separate branch-name bookkeeping. `mr status <number>` doubles as a CI gate — its exit code is `0` if approved, `2` if open but not yet approved, `1` on any other error. `mr comment` / `mr comments` post to and list the same discussion thread the web dashboard and Desktop App show.

`mr approve` / `request-changes` / `merge` / `close` / `reopen` put review actions a terminal command away, not just the web dashboard or Desktop App. The hub's `self_review` gate stops the merge request's own creator from reviewing it; an approval from a second credential counts, so treat review-capable credentials like any other approval authority in your org. If you want an independent-approval guarantee, protect the target branch with `--required-approvals` set above 1 (below).

Protecting a branch

`swarmfile merge` and a direct `mr merge` both bypass review the moment nobody insists otherwise. **Protecting** a branch removes that choice for changes arriving *as a merge*: once set, a bare `swarmfile merge` into it (or `mr merge` on a request that hasn't cleared the branch's approval/checks gates) is refused outright with `protected_branch` — the branch's changes have to arrive through an approved Merge Request.

Protection governs **what lands on** the branch, whether it arrives as a merge or a direct write. Once set, a bare `swarmfile merge` (or `mr merge` on a request that hasn't cleared the branch's gates) is refused, and so is every direct write from a mount: saving, editing, renaming, moving, or deleting a file, creating one, staging/committing directly, rolling back, resolving a conflict, or reflecting LFS content all answer `409 protected_branch`. The way through is a branch plus an approved Merge Request; repository ACLs still decide *who may ask*, but they cannot make a protected branch writable.

Protection is enforced by the hub. Staged work on a newly protected branch is held for you to move to a branch, not discarded.

```
swarmfile branch protect release
swarmfile branch protect release --required-approvals 2
swarmfile branch unprotect release
```

Setting or clearing protection is admin-gated — a plain repository member gets refused (the CLI exits `2`, distinguishable from a hard error), same as trying to archive a protected branch. Every protect/unprotect leaves an audit trail: who did it, and when, shows up in the dashboard's Activity feed (owner-visible, same as an ACL-mode change). The web dashboard's Files-tab branch switcher and the Commits toolbar both offer a protect/unprotect toggle for admins, alongside a 🔒 badge everyone sees; the Desktop App's Branches page has the same toggle.

On a protected branch the mergeability rule is "≥ the branch's effective required-approvals (1-10, default 1), 0 requested-changes"; on an unprotected branch the approval half is 0, so a direct `mr merge` needs none until the branch is protected.

A single `changes_requested` verdict still blocks a merge outright no matter how many approvals exist, and that part isn't configurable.

`--required-approvals` (1-10) sets the approval half for this one branch; leaving it off keeps the default of 1 while the flag is set. There's still no required-*specific*-reviewer *enforcement* — raising the count is the whole gating mechanism, not naming who has to be one of the approvers; requesting a specific reviewer flags whose attention it needs without making their approval individually required.

```
swarmfile protection-rule create release --require-checks-to-pass
```

Required checks

`--require-checks-to-pass` exists on **protection rules only**, not on `branch protect` — to gate one exact branch on CI, use a no-wildcard rule like the one above. It adds a second, independent gate alongside approvals: once set, a merge is refused unless the *latest* run of every check reported at the merge request's exact current head — not an earlier commit it happened to pass on — is green.

A rule that last reported at an earlier commit isn't carried forward as still-required, but if no check has reported at the head at all, the gate fails closed rather than passing vacuously — the same "no news is bad news" convention GitHub's required-status-checks uses, so the gate can't be quietly satisfied by a check nobody's runner has gotten around to yet.

This stacks with `--required-approvals` rather than replacing it — both have to clear — and the merge request detail view exposes checks state separately from approval state, so an MR can visibly be "approved, blocked on checks." A check doesn't have to come from Swarmfile's own runner — any CI system that can call an authenticated API can report one.

Protecting many branches at once

Protecting one branch by name is fine for `main`, but not for a `release/*` line that grows every sprint. A **protection rule** protects every branch whose name matches a glob pattern — including ones forked *after* the rule was created, with nothing to re-apply:

```
swarmfile protection-rule create "release/*" --required-approvals 2
swarmfile protection-rule list
swarmfile protection-rule delete "release/*"
```

`*` matches any run of characters, `?` matches exactly one; a pattern with no wildcards (`protection-rule create main`) is just another way to protect one exact branch — equivalent to `branch protect main`, not a second, different mechanism.

A branch counts as protected if its own flag is set OR any rule matches its name, so deleting a rule can un-protect a branch that was never explicitly protected itself.

`--required-approvals` composes the same way: the effective count for a branch is the MAX of its own value (if flag-protected) and every matching rule's — the same "most restrictive wins" resolution the boolean uses, just with the higher number instead of `true`. Same admin gate, same exit-`2` convention, and same audit trail as the per-branch toggle — a rule is arguably the more consequential of the two, since one rule change can flip many branches (including future ones) at once.

The two mechanisms don't override each other: `branch unprotect release/1.0` only ever clears that branch's OWN flag. If a rule like `release/*` still matches it, it stays protected regardless, and every surface says so — the CLI/Desktop App/web response reports the branch's real, effective state after the unprotect, not just "done."

A protection rule is also where auto-reviewers live:

```
swarmfile protection-rule add-reviewer "release/*" <principal-id>
swarmfile protection-rule remove-reviewer "release/*" <principal-id>
swarmfile protection-rule list-reviewers "release/*"
```

Every merge request opened against a branch matching `release/*` requests a review from each configured principal automatically, the moment it's created — no one has to remember to add them by hand. The list is matched against the merge request's *target* branch, same as the rule's own pattern; it's additive to whatever

reviewers get requested manually, never a replacement.

Rules are managed from the CLI or the web dashboard's **Branches** tab (admin-gated, alongside Commits) — a single view of every branch's protection status plus every active rule, since neither the Files-tab switcher nor the Commits toolbar shows more than one branch at a time.

The same **Branches** tab has an **Edit** button on any already-protected branch to raise its required-approval count without unprotecting it first; the Desktop App's Branches page has the same Edit button on its own protected-branch rows, an inline number field replacing the row's usual buttons rather than a separate dialog.

Protection rules themselves — creating or deleting a `release/*`-style pattern — are CLI/web-only; a Desktop App user sees a rule-protected branch's 🔒 badge (with an `×N` suffix once the count is above 1) correctly either way, since the hub resolves the effective protection state and count server-side before any client sees them, just can't create or delete a rule from the Desktop App.

Protecting a branch — by flag or by rule — is only as strong as the Merge Request review it forces you through, and it inherits the caveat above: the hub stops an MR's own creator from approving it, while an approval from a second credential counts.

`--required-approvals` above 1 raises the bar — clearing a 2-reviewer rule takes two approval credentials, not one — but no server-side check can prove that two accounts are two different humans without identity verification. If you protect `main` specifically to guarantee a human looked at every merge into it before it landed, keep review-capable credentials out of any automation that opens merge requests.

A different kind of conflict

Branch merges aren't the only place conflicts happen — and the other kind is more common. If you and a teammate both edit the same file on the *same* branch while one of you was offline, Swarmfile catches it when the offline edit tries to land: the change was made against a base version that's no longer current.

Rather than silently keeping both copies and leaving you to notice, it records a conflict and **refuses further writes to that file** until it's resolved.

That's why the file shows as read-only on macOS and Linux and an application's save fails with "permission denied" ("access denied" on Windows), and why `swarmfile conflicts` exists — a filesystem has no way to say "someone else changed this too."

The web file browser badges the row with **Conflict** so it's visible without opening the file, the Desktop App's **Conflicts** page lists it with its path (and the branch, when it isn't the one this mount is on), and the CLI is where a terminal sees the reason.

Resolving is a deliberate choice between several outcomes. In the web dashboard and the Desktop App (both open the same picker):

- **Keep mine** — your offline version becomes the current head; the other edit is superseded.
- **Keep theirs** — keep the version already on the hub. Your offline edit isn't discarded: it's saved in the file's **history** (as a version owned by you), so you can restore it later with a rollback or from the History view. Choose **Keep both** instead when you want your version visible alongside the file right away.
- **Keep both** — the hub keeps the current head, and your version lands alongside it as a sibling file with a conflict suffix, so nothing is lost and you reconcile them by hand.
- **Try an automatic merge** — for file types with a registered merge tool (currently plain text and common source/config formats), click "Try automatic merge" and Swarmfile runs a three-way line merge of your edit against theirs. If it succeeds cleanly, a fourth choice appears — the merged result — for you to review and accept instead of picking a side. If the file type isn't supported, or both edits touched the same lines, the attempt just fails harmlessly and you're left with the three choices above.

Resolving from the CLI

From the CLI, `swarmfile conflicts resolve <path>` covers the first three non-interactively via `--winner local|remote|keep-both`.

`--auto` runs the same built-in three-way merge the picker's "Try automatic merge" does — no external tool and no git config needed, and it exits 2 when a real conflict remains (so a script can tell "a human must choose" apart from "the engine was unreachable" and fall back); add `--all` to sweep every conflicted file on this mount's branch in one pass (rows on other branches are counted and skipped unless `--include-other-branches` is given — pass it or `merge --diff3` when landing a merge, whose conflict is recorded under the target branch).

`--tool [name]` is the option the picker doesn't have: it hands the conflict to an **external mergetool** — kdiff3, Beyond Compare, VS Code's merge editor, anything speaking git's own `$BASE` / `$LOCAL` / `$REMOTE` / `$MERGED` protocol — auto-discovered from your git config. Unlike the built-in merge, `--tool` works on any file type, binary included, since the tool does the actual reconciling rather than Swarmfile attempting a line-based diff.

See CLI: swarmfile for the full flag reference. `--tool` is CLI-only by design: an external tool needs a real terminal/GUI session to run in, which only a foreground CLI process has.

Like branch merging, whole-file is always available as a fallback here — a binary asset never has a line-level merge to fall back on, which is why keep-mine/keep-theirs/keep-both always work regardless of file type. Until you resolve the conflict, the file stays read-only; once you do, writes resume with no restart.

An unresolved conflict also stops every other operation that would move that file's content: a branch merge, a point-in-time restore, a commit revert, and a rollback all refuse with `conflict_unresolved` until it's resolved. Resolve it first, then re-run the operation.

Reviewing conflict history

Resolving a conflict does not erase what happened. The conflict journal keeps the lifecycle — **open**, **resolved**, or **abandoned** (withdrawn by a delete or a namespace cleanup) — together with its **origin** (a content save or a branch/MR merge), the resolver, and, for a merge, the merge request that produced it.

From the CLI:

```
swarmfile conflicts list --state resolved   # or abandoned / all
swarmfile conflicts show src/app.ts        # all states; --json for scripts
```

`conflicts show` prints each row's state, kind, origin, resolution and time. The web **History** view shows the same per-entry history under the file's versions, and the Desktop App's **Conflicts** page keeps a **Recently resolved** list. A Merge Request's page also shows **Your conflict history on this request** — caller-scoped, so you see the rows you raised, not a coworker's.

History is read-gated: the repository-wide list only ever shows your own rows; the per-entry view may show a coworker's row, but only if you can already read that file. Hosted repositories keep resolved rows for at least 30 days; the self-hosted community server currently keeps them until you prune the repository database.

Safe-save copies appear in the history too. When an app's save lands beside a teammate's version as a conflict copy (`notes.txt.sb-…`), the journal records a `replace` or `aside_delete` row linking the copy to the live occupant. It never makes the file read-only. Resolve it by its id:

```
swarmfile conflicts show notes.txt --json   # print the conflict id
swarmfile conflicts resolve notes.txt --conflict-id <id> --winner keep-both
swarmfile conflicts resolve notes.txt --conflict-id <id> --winner abandon
```

`keep-both` leaves the copy in place; `abandon` soft-deletes it (hosted: it stays recoverable from Trash; the self-hosted community server has no Trash UI yet and marks the row deleted in its database). Neither moves the occupant's head.

Where to go next

- Version Control covers changesets, checkout, and per-file history — the model branches sit on top of.
- CLI: swarmfile has the full command reference for `branch` and `merge`.

Trash, History & Rollback

Every change to a file is timestamped, and you can always roll back to an earlier point in time. This page covers the lightweight, per-entry mechanism for that: trash and per-file history. For the multi-file, atomic changeset-and-checkout system, see Version Control — that's a different, heavier tool for coordinating a whole set of changes together, not a replacement for this one.

Trash

Deleting a file doesn't remove it immediately. It soft-deletes into a per-user trash, visible under the repository's **Trash** tab in the web dashboard (open the repository, then Trash in its tab strip). There is no cross-repository trash view — each repository's trash is its own.

You can delete from either place — your file manager (Finder, Explorer, or the command line, on the mounted drive) or the web dashboard. Both do the same thing: the file goes to trash, not away. Deleting a folder takes its contents with it. The dashboard does that in one step; your file manager removes the files inside one at a time and the folder last.

Restoring a folder brings back the folder and everything that was deleted with it in the same step. A folder deleted from your file manager went one file at a time, though, so when you restore it, the web dashboard lists the files that were deleted inside it just before it and asks whether to restore them too. It asks rather than restoring them automatically, because a file you deleted on purpose a moment earlier looks the same.

Retention is per-user, not just per-org: your own account setting, if you've set one, wins; otherwise it falls back to the org's default; otherwise a global 90-day default applies. Set your own under **Settings → Account → Trash & history retention** (1 to 3650 days), or clear it there to go back to the organization's window.

An org's default retention window is settable anywhere from 1 to 3650 days and controls how long a deleted file stays recoverable before it's permanently purged, for anyone in the org who hasn't set their own override. Until that window elapses, restoring a file from trash brings it back in place, at its original path. Retention windows are applied by a daily cleanup, so a change takes effect at the next daily pass rather than instantly.

From the CLI, `swarmfile trash list` shows your own soft-deleted entries, `swarmfile trash restore <entry-id>` brings one back (add `--with-nearby` to also bring back what was deleted inside a folder just before it), and `swarmfile trash purge <entry-id>` deletes it permanently — "Delete Forever," ahead of the retention window, not recoverable. `--preview` on `purge` reports how many descendants a folder would take with it before you commit to removing anything.

Because it can't be undone, `purge` asks for confirmation first; `--yes` skips the prompt (a non-interactive shell or `--json` needs it).

A very large folder restore or purge runs as a durable background job rather than an inline cascade: the command waits for it by default, and Ctrl-C or `--no-wait` detaches with exit `3`, followed with `swarmfile op status` / `op wait` (long operations). `op status` shows the job's `done` / `total` progress while it runs, and in the dashboard the same operation shows progress on the row's own button — a restored folder reappears at once, with its contents filling in behind it.

Undo and redo in the Desktop App

The Desktop App keeps a journal of the file operations you make on this computer, so you can take back a delete, rename, move or restore without going through Trash or history.

- **Drive menu:** click the drive name at the top of the app. It lists **This device's recent operations**, each with its own **Undo**, plus **Undo last operation** and **Redo last operation**.
- **Operation journal:** on the activity bar, **Operation journal** shows the full list, newest first, with each row's state and time; **Load older operations** pages further back. Each row offers **Undo**, or **Redo** once it has been undone.

A multi-select delete or move undoes as one. Deleting or moving several files at once from the app is recorded as one gesture (the row says *one gesture of N items*), and Undo or Redo on any of its rows acts on all of them. If some files can't be reversed, the app names them and why. The rest still go through, and undoing again retries only the ones that were refused. A new file you deleted before it finished uploading is gone for good, so it's named as not undone rather than failing the rest.

What undo will and won't do:

- **It's this computer only.** The journal is local: it doesn't show teammates' changes, or your own from another computer, and it never rewrites history.
- **Nothing newer is overwritten.** If a teammate changed a file after your operation, undoing it is refused, and so is undoing while a file still has local changes waiting to sync (try again once they've synced).
- **Edits aren't in it.** Changes to a file's contents go back through its history, not undo. Restoring a folder can't be undone from here; delete it again instead.
- **Changes made in Finder or Explorer** on the mounted drive appear in the journal but can't be undone from it; use Trash or the file's history.
- **Old rows age out:** the journal keeps 90 days and up to 2,000 operations per repository.

The same journal is available from a terminal as `swarmfile undo` and `swarmfile redo`.

History and rollback

Because every change is timestamped, a file's full edit history is available at any time, and you can roll back to any point in that history — not just the most recent save. This is scoped to a single entry: rolling back a file doesn't touch anything else in the repository.

Every saved version vs. named commits

New repositories start on **Keep every saved version**: each automatic save stays restorable, so any point in a file's history can be brought back — not just its named commits.

If you'd rather keep storage leaner — on a busy, high-save-volume repository, say — choose **Named commits only** when creating the repository, or switch later in the repository's settings (or with `swarmfile repository set-keep-full-history false`): individual save versions then expire after the retention window, and committed changesets are kept for that same window — tag one to keep it indefinitely (Keep every saved version keeps everything).

The setting is per repository, not per file; an org owner can change which choice new repositories start with under **Settings → Organization** (which applies only to repositories created afterwards).

When history is kept for less time

On a repository set to **Named commits only**, a file's history is kept for your retention window, with one exception: if a repository's stored file information grows very large (millions of files and versions), Swarmfile automatically keeps history for less time — never below a minimum of 7 days by default — so the repository stays under its storage limit. It returns to the full window once the repository is comfortably below that point again.

Org owners and admins can see the window currently in force, and why, on the repository's **Repository settings** tab. See Repository storage limit.

Line attribution

For a text file, open its **History** panel in the web dashboard (managed/plaintext) or the Desktop App (including end-to-end encrypted repositories) and switch to **Line attribution**.

Swarmfile reads the saved versions locally and shows which version and writer last introduced or replaced each line, including the agent/machine label when one is set. Lines copied into the target by a clean merge keep the source version's writer and age.

The view is read-only and separate from rollback; Synthetic hub-side versions (`d1:`) can be attributed but not restored locally, and history that starts mid-file is marked as truncated instead of guessing. The CLI equivalent is `swarmfile blame ––lines <path>` .

Trash and history vs. version control

Trash and history operate one entry at a time, with no concept of grouping changes across files. If you need to restore a whole folder or repository to how it looked before a coordinated set of changes — a batch of asset updates that need to move together, for example — that's what the changeset/checkout system in Version Control is for.

Line Attribution

Line attribution shows **who last changed each line** of a text file and which saved version introduced or replaced it. It is a read-only view over the repository's existing version history — it never writes line data to the hub and never needs a Git clone.

Where to find it

- **Web dashboard and Desktop App:** open a text file's **History** panel and choose **Line attribution**. The web view can also open the comparison for a line's version; the Desktop App jumps to that version in the history list.
- **CLI:** `swarmfile blame --lines <path>` prints the ranges (and `--format=gutter` a blame-style gutter). `--json` returns the same ranges for scripts.

Attribution is available for files the client can decrypt and read. End-to-end encrypted repositories work in the Desktop App (the key never reaches the hub); the web view cannot decrypt them and points at the Desktop App.

Reading the view

Each range names the writer and the version's own time. Copied source rows from a clean merge keep the **source** writer and age, not the merge time; a merge line is labeled like an ordinary save. A line the history cannot resolve is reported as `unknown` rather than credited to the wrong writer. If the retained window starts mid-file, the view says the history is truncated and marks the prefix unknown.

Source copies: a clean merge copies up to 30 source versions into the target as internal side rows. Ordinary history hides them; the web History panel's **Show source copies** button (and the Desktop App, when a line range points at one) reveals them with a `source copy` badge. They are display-only provenance and cannot be rolled back from the versions table.

Limits

- Text files only; binaries have no meaningful line-level diff.
- Copied source rows are a hosted-hub feature; the community server's history has no copied source side rows, so clean merges there have no source-age attribution.
- A version over 8 MiB, a file over 50,000 lines, or more than 64 MiB of version text stops the walk and is reported as truncated; an oversized **current** version is a hard error instead. The walk also has a wall-clock bound.
- The CLI/engine reads up to `--max-versions` (default 100), enforced at page granularity (a page can add up to 50 versions beyond the limit); the web panel keeps 32 content versions and at most 96 copied side rows. Older versions may be outside either window. Managed and plaintext content is read through the same signed version URL as the diff view.
- A version read timeout is reported as `content_unavailable` (there is no separate `content_timeout` code).
- **Staged changelist work is not included** until it is submitted; the panel says so.
- Exact agent attribution requires the save to come from the agent's own engine session. The line label carries the machine/agent identity the history row recorded; several agents sharing one mount are not separated by line.

Privacy

The view reads only versions the caller may already read. Copied source rows are gated on the source entry's ACL; the hub hides a row the caller cannot read and redacts the copied source CID from the merge row. A revoked source entry is refused by the engine before cached content is served.

See also Trash, History & Rollback and the CLI reference.

Verifiable History

Version history is only as trustworthy as whoever keeps it. Swarmfile's history is built so you don't have to take our word for it: every commit is **hash-chained** to the ones before it, the service **signs checkpoints** of each branch with a published key, and a standalone tool, `swarmfile-verify-history`, lets you check a repository's history yourself — on an audit host or in CI, with no Desktop App or mount.

This page explains what that gives you and where its limits are. For flags and exit codes, see the CLI reference.

Hash-chained commits

Every commit has a **commit hash** computed from its own facts: the content-addressed id of the repository tree it records (which in turn is built from the ids of every file's contents) and the hashes of its parent commits. Because each commit's hash covers its parents' hashes, a commit's hash covers the whole history behind it. Change one file in one old commit, or remove a commit from the middle, and every hash after it stops matching.

Commits get their hash once they're **confirmed**. Swarmfile doesn't compute commit hashes on the server: a client derives the commit (the Desktop App does this after a commit, or `swarmfile git index` on demand), and a second, independent derivation confirms it — a teammate's app, a fetch, or a repository API key used in CI; an owner can also allow their own app to self-confirm.

Commits made where no app runs (a web upload, a revert, a merge) are derived by Swarmfile's server, which counts as one independent derivation: the first matching derivation from a client confirms them.

Commits pushed with git work the other way round: the push derives them, and the hosted hub's own derivation — queued as soon as the push's claim lands, normally confirmed within seconds — confirms them; a community server confirms a member key's push from its own tree derivation at claim time (a claim it cannot reproduce waits for another key's claim; an owner key's own first claim is recorded confirmed directly — the community server has no self-confirm setting).

Self-confirm covers only the owner's own app, so on a self-confirm repository these commits stay provisional until a client derives them (a fetch is enough). Until then a commit is *provisional*: it's in your history and fully usable, it just isn't part of the hash chain yet. `swarmfile git status` reports how many commits on a branch are still unconfirmed.

Signed checkpoints

Hash chaining proves the history is internally consistent. It doesn't, on its own, stop someone who controls the server from replacing the whole chain with a different, equally consistent one. That's what checkpoints are for.

Periodically (hourly, for every branch that has changed since its last checkpoint) the Swarmfile service signs the branch's latest **confirmed** commit hash with an **Ed25519** key. The verifier requires every checkpoint published for a branch to land on the chain it walks, so a history that has lost or replaced a checkpointed commit fails verification.

The signing key can be rotated; the service publishes its current and retired public keys so older checkpoints stay verifiable. Verifiable history is offered where the service has checkpoint signing enabled, and the verifier reports it as unavailable where it isn't (see What it proves, and what it doesn't).

The self-hosted server ships with its own signing key: it mints one into its data directory on first use and signs the branch's confirmed head when a verifier asks, so the same check works against your own server.

Checking it yourself

`swarmfile-verify-history` is installed with every Desktop App and runs on its own. The verifier itself is open source — source, releases and conformance checks at swarmfile-verify-history — so checking the history doesn't require trusting our binary either.

```
swarmfile-verify-history \
  --hub-url https://hub.swarmfile.com/orgs/<org-id>/repositories/<repository-id> \
  --org-id <org-id> \
  --repository-id <repository-id> \
  --branch main \
  --header "Authorization: Bearer $SWARMFILE_API_KEY" \
  --pin-key ./swarmfile-history-key.pub
```

It fetches the branch head, then walks back commit by commit to the start of history (or to a commit you name with `--checkpoint-hash`), recomputing every hash and checking every parent link. It then fetches the branch's checkpoints and checks each signature. Any mismatch is a failure with a non-zero exit, so you can gate a release pipeline on it; `--json` gives a machine-readable report.

Figure omitted in print (animation): swarmfile-verify-history recomputes a branch's commits from the canonical encoding. Then one commit's recorded message is edited directly in the server's database, and the next run names the commit it broke and exits non-zero. The chain is recomputed, not taken on the hub's word. - https://swarmfile.com/demo/oss-tamper-verify.cast

Pin the key. Without `--pin-key`, the verifier checks signatures against the public key the same service is serving you, which proves the checkpoints match *that* key but not that the key is genuine. Pass the key you recorded out-of-band (from an earlier run, or from us directly) with `--pin-key` and the run only accepts signatures from keys you supplied. That's the mode that gives you independent assurance. Repeat `--pin-key` to accept a rotated key alongside the old one.

Key ↔ origin (self-hosted). `--check-host-identity` fetches the server's signed `/.well-known/swarmfile.json`, requires its `origin` to match the repository URL and its key to be the active checkpoint key (or a pinned one).

A self-hosted server serves it when `SWARMFILE_SERVER_PUBLIC_URL` is set to its public origin; the hosted hub serves it at the hub origin when its dispatch worker carries the same history-signing secret and `HUB_PUBLIC_ORIGIN` (otherwise it fails closed with 503). It is optional; the chain/checkpoint check above runs either way.

The wire shape and a synthetic conformance fixture ship in the verifier's public-repo template (`docs/host-identity.md`, `fixtures/host-identity.json`) and publish with its next release.

A portable origin manifest. `--write-provenance origin.json` records a verified origin: the canonical origin URL, the head's commit facts/tree, the signed checkpoints and the key ring. Later, `--provenance origin.json --hub-url <mirror repository>` verifies the mirror offline — no origin contact — by re-hashing the manifest's commit, checking the head checkpoint signature (and any `--pin-key`), then accepting a matching mirror head **commit** or **tree**.

A plaintext destination written by `swarmfile-mirror --provenance-out` preserves the origin's block CIDs and entry lineage, so its tree and usually its confirmed commit match; an ordinary plaintext mirror copy has a different commit DAG and different entry lineages, so it does not tree-match; only a provenance-mode copy preserves the tree; a managed/E2E **mirror** copy re-seals and will not match; a same-server `swarmfile-server admin copy-repository` copies the key/data without re-sealing and preserves the managed commit DAG.

A manifest is a file, so `--provenance` requires `--pin-key` (the key obtained out-of-band) and should also pass `--origin <URL>` to bind the origin label; when the origin serves `/.well-known/swarmfile.json`, the manifest carries that signed statement and `originAttested` reports it.

Neither mode proves the tree object or any block is served by default; `--verify-blocks` samples file manifest/chunk blocks and re-hashes them, which proves the sampled bytes (not the whole mirror). Treat the manifest as the portable attestation and the mirror as the copy.

Federated clones. A clone through a federating server (`swarmfile://<hub>/fed/<peer>/<org>/<repository>`) is verified the same way: write the manifest against the origin, then run `--provenance <manifest> --pin-key <origin-key> --origin <origin-url> --hub-url <fed-url> --verify-blocks N`.

The federator carries the origin's history and blocks unchanged, so the mirror head/tree and sampled blocks match; the manifest, not the federator's own `/.well-known`, is what attests the origin. A live `--check-host-identity` pointed at the fed URL would check the federator (its origin root), not the peer, so use the manifest for federation.

Checking a mirror. Pass the mirror's repository URL and ids with `--mirror-hub-url`, `--mirror-repository-id` and `--mirror-org-id` (plus `--mirror-header` for its credential). The verifier verifies the origin exactly as above, fetches the mirror's **confirmed** head, and requires the mirror's confirmed head commit to equal the origin's verified head commit.

The commit hash binds the tree CID, so a matching head cannot substitute a different tree; the verifier does not fetch the mirror's tree/blocks or require the head to be checkpointed. A `swarmfile-mirror` copy does not preserve the commit DAG — use a replica that does. Read this together with the checkpoint signature/pin result.

A mismatch — or a mirror branch with no confirmed head — fails the run and is reported in `--json` as `mirrorHeadMatches: false`. If the **origin** branch has no confirmed head, the comparison cannot run: the run fails with `mirrorHeadMatches: null` and a "not compared" note.

What it proves, and what it doesn't

- **It proves** that the history the service serves you is internally consistent — no edited commit, no broken link — and, with a pinned key, that the checkpoints on it were signed by the key you trust.
- **It doesn't, by itself, prove** that this is the same history you saw last month. The checks run against the checkpoints the service lists, so keep your own record of a commit hash you care about (a release, a delivery) — or of the checkpoints from an earlier run — and confirm it's still on the branch.
- **Provisional commits aren't covered** until they're confirmed. A branch with no confirmed commits yet reports that, rather than passing.
- **It needs the service to be reachable.** The verifier needs no Desktop App or mount, but it reads the commit objects and checkpoints from the service as it walks; it doesn't verify an exported bundle offline.
- **Checkpoints require checkpoint signing on the service.** Where the service doesn't have it enabled, the verifier reports verifiable history as unavailable — an informational result unless you pinned a key, in which case it's a failure.

Where you'll see it

- **Releases** of a public repository marked **commit-pinned** name the exact commit hash they were published from. The badge is the pointer; `swarmfile-verify-history` is how you check it.
- **`git clone`** of a repository uses the same confirmed commits: each repository commit in the clone carries a `Swarmfile-Commit:` trailer naming the Swarmfile commit it came from (the graft commit that anchors history at the cut, if any, has no trailer). See Cloning a Repository with Git.
- **Reproducible checkouts** — `swarmfile materialize hash:<commit-hash>` writes exactly that commit's tree, the most precise pin you can give a CI job (tags can be moved; hashes can't).

Git and Swarmfile

For an organization, Swarmfile is a version-control system in its own right — and for large, binary-heavy repositories a more flexible one than Git: there is nothing to clone or keep in sync (the repository streams from a mounted drive), access is per folder, and locking, review, rollback and releases are built in. Nothing here asks you to leave Git: the three paths below are where the two meet, so you can adopt Swarmfile self-serve through whichever fits and move more of your work over at your own pace.

Swarmfile isn't a full code-hosting forge (public repositories get only a light issue tracker, and code review happens through Swarmfile's own merge requests), but a repository with git access enabled serves a real git repository you can `git clone swarmfile://` and `git push` to. Plenty of teams that use it live in git too, so there are three separate ways the two fit together. They solve different problems, and you can use any combination:

	What it's for	Where your source lives
Swarmfile as your git-LFS server	Keep your repo on your git host; send the big binary files to Swarmfile.	Your git host
`git clone swarmfile://`	Get a real git repository of a Swarmfile repository, for git tooling, CI, or an offline archive, and push commits back.	Swarmfile (the mounted drive)
The `swarmfile` CLI's git-style commands	Do version control on the mounted drive with the verbs you already know.	Swarmfile (the mounted drive)

1. Swarmfile as your git-LFS server

Point git-LFS at a Swarmfile repository, and `git lfs push` / `pull` move your large files through Swarmfile while everything else stays in your existing git host. You get storage, deduplication and quota without bloating the host, pushes up to 256 GiB per object with the transfer agent, `git lfs lock` that is the same lock as a lock on the drive, and an exit with no lock-in: `git lfs fetch --all` with stock git-LFS pulls everything back. Pushed objects can also be reflected onto the mounted drive as ordinary files (through the transfer agent — a stock `git-lfs push` stores the object but doesn't place it on the drive).

On a managed repository, git-LFS uploads must use the Swarmfile transfer agent and are encrypted at rest; a stock direct push is refused with `lfs_agent_required`. On an unencrypted repository the direct paths remain and are plaintext by definition. Objects a managed repository accepted before the agent requirement are converted by an operator-run migration. git-LFS isn't available on end-to-end-encrypted repositories.

Start with Git-LFS.

2. `git clone` of a Swarmfile repository

Once an owner turns on git access for a repository, anyone who can read all of it can run:

```
git clone swarmfile://<org>/<repository>
```

and get every branch and tag as real git history, with large files as LFS pointers served by Swarmfile. `git fetch` brings it up to date, and `git push` sends new commits back as Swarmfile commits with their git ids unchanged. Pushes are fast-forward only, a push to a protected branch opens a merge request instead, an existing repository can be pushed whole into a new repository, and a pushed commit must look exactly like what a clone would produce (large files as LFS pointers, submodule gitlinks as mode `160000`; executable bits are kept) — the guide has the rules.

A pushed commit is *provisional* until a second, independent derivation confirms it. Swarmfile's hosted hub does that on its own as soon as the push's claim lands (normally within seconds); a community server confirms a member key's push from its own tree derivation at claim time (a claim it cannot reproduce waits for another key's claim); a teammate's fetch also confirms it, and `swarmfile git index` confirms a whole branch's unconfirmed commits in one run. A clone can opt out with `git config swarmfile.confirm false`. See How a pushed commit gets confirmed.

Figure omitted in print (animation): An owner enables a repository's git surface with one command; an ordinary git clone speaks swarmfile:// through the helper; a commit pushes fast-forward; and swarmfile-verify-history walks the chain it just created. - https://swarmfile.com/demo/oss-git.cast

Start with Cloning a Repository with Git.

3. The `swarmfile` CLI speaks git

On the mounted drive, the `swarmfile` CLI uses git's vocabulary where the meaning carries over — and where it doesn't, it tells you instead of failing.

Commands that work the way you expect: `commit` (including `--amend`, message-only), `log`, `show`, `diff`, `branch`, `switch`, `checkout`, `restore`, `revert`, `merge`, `tag`, `describe` and more. Your `~/.gitconfig` aliases work too (`swarmfile co`, `swarmfile st`). Coming from Git has the full cheat sheet.

Git habits that run the real thing:

You type	It runs
`swarmfile add`	Opens a changelist (if one isn't open), so your next saves are staged into it.
`swarmfile stash` / `stash push` / `stash save`	Parks the open changelist (`changelist park`).
`swarmfile stash list` / `stash show`	Lists parked work (`changelist parked`).
`swarmfile stash pop` / `stash apply`	Resumes parked work (`changelist resume`).
`swarmfile stash drop` / `stash clear`	Discards parked work (`changelist discard`), asking first.
`swarmfile reset --hard <rev>`	`swarmfile restore <rev>` — one undoable commit, not a history rewrite.
`swarmfile clone <repository-id> <path>`	Opens a mount of that repository at that path (`-b <branch>` pins a branch; `--checkout` writes a plain directory instead).

Git habits that print guidance instead: `push`, `pull`, `clone` (with anything other than a repository id and path), `rebase`, `cherry-pick`, `rm`, `mv`, `cp`, `mkdir`, `touch`, `worktree`, `grep`, `init`, `remote`, `clean`, `reflog`, `mergetool`, `difftool`, `shortlog`, `submodule`, `bisect`, `gc`, `prune`, `fsck`, `archive` and `notes` each print a one-line reason and what to do instead — `push` and `pull` explain that saves already sync, `worktree` points at `swarmfile mounts open`, `grep` at `swarmfile-search`, `mergetool` at `swarmfile conflicts resolve --tool`.

These work even when the Desktop App isn't running. The CLI reference has the details.

Compatibility at a glance

You want to...	Today
Keep source in your git host and store big files in Swarmfile	**Works** — git-LFS server
`git lfs lock` files, including files only in git	**Works**
Leave with all your LFS objects using stock git-LFS	**Works** — `git lfs fetch --all`
`git clone` a Swarmfile repository	**Works**, once git access is on for the repository — guide
`git fetch` / `git pull` new commits into that clone	**Works**, incrementally
`git clone --depth N` (shallow)	**Works**, once the repository's commits are indexed; `--shallow-since` / `--shallow-exclude` don't
`git push` to a Swarmfile repository	**Works**, fast-forward only, on managed and unencrypted repositories, and (through the desktop engine helper or the standalone/API-key helper with this device's user-key env) on E2E repositories — rules: deleting a branch archives it, lightweight and annotated tags become Swarmfile tags (an annotated tag keeps its exact object, message, signature and id; `--force` moves one and `:refs/tags/<tag>` deletes one, unless a release was published from it), a push to a protected branch opens a merge request, an existing repository can be pushed into a new repository in batches of up to 100 commits, a merge brings unpushed side history along on an auto-archived `merged/<sha>` branch, and a commit changing more than 5,000 paths goes through a chunked, resumable push session with rollback; force pushes to a branch are refused for an older clone (a helper-created clone translates them into a forward commit by default) — `swarmfile git force-forward` lands a force's end state as a forward commit instead
Import an existing git repository into a new Swarmfile repository	**Works** — `swarmfile git import` creates the repository (git access on, full history) and pushes every representable branch and tag with their upstream ids; annotated tags keep their exact object id by default (`--annotated-tags convert` imports them as lightweight, `skip` / `fail` to leave them out or stop); a default branch that uses Git-LFS, submodules or an octopus merge is refused, while the same problem on another branch skips and lists it; re-running the command syncs upstream changes into the repository it made (`--new` for a second repository)
Partial clone (`--filter`)	**Not supported through the `swarmfile://` helper** — large files arrive as LFS pointers; the read-only HTTPS gateway supports it for public plaintext repositories
Clone over a stock `https://…` git URL	**Not on the hosted hub** — use `swarmfile://`, or an operator-run read-only gateway for public plaintext repositories (clone/fetch, `--filter`, LFS downloads; no push/upload/lock)
Clone an end-to-end-encrypted repository	Supported by the **desktop engine helper** and the **standalone/API-key helper** with `SWARMFILE_USER_KEY_PRIVATE_B64` / `SWARMFILE_USER_KEY_ID` (all-inline; no LFS).
Fetch an arbitrary commit by id	**Not supported** — branches and tags only
Executable bit in a clone or push	**Works** — executable files are mode `100755` both ways (`swarmfile chmod` sets it on Windows)
Empty folders in a clone	**Not carried** — git can't store them
Git submodules	Gitlink push/clone is supported in the git view; the drive hides the submodule path. `git import` still skips a branch containing a submodule.
Rebase, cherry-pick, history rewriting	**Not supported** by design — Swarmfile never rewrites commits; `commit --amend` changes only the message

Which one should I use?

- **Your source code already lives in git and you're happy with that** → the git-LFS server on-ramp. Nothing about your workflow changes except where big files go — and when you want the drive, review and hosted CI for the whole repository, turn on git access and the same repository serves the code too.

- **Your repo lives in a git host and you want it *as* a Swarmfile repository** → `swarmfile git import <url>` brings the whole history, branches and tags over in one step (reference).
- **Your repository lives on the Swarmfile drive, and some tool needs a git repository** → `git clone swarmfile://`.
- **You're doing version control on the drive and your fingers type git** → the CLI, with Coming from Git open in a tab.
- **You run coding agents in parallel on one repository** → give each agent its own mounted workspace — a mount per agent and a shared dependency cache — instead of a clone or worktree per task.
- **CI that builds from a branch** → you may not need a clone at all: Hosted CI runs `.swarmfile/ci.yml` workflows on Swarmfile's containers against the branch directly, the self-hosted Runner (Headless CI) runs jobs on your own machine, and `swarmfile materialize` writes any ref to a plain directory.

Cloning a Repository with Git

A Swarmfile repository is normally something you **mount**, not clone: the drive is live, saves sync as you make them, and there's no working copy to keep up to date. But some tools want a real git repository — a build system, a code-review or static-analysis tool, a CI job, an archive you keep offline. For those, an owner can turn on **git access** for a repository, and anyone who can read the whole repository can then:

```
git clone swarmfile://<org>/<repository>
```

The result is an ordinary git repository: real commits, branches and tags, with large files as git-LFS pointers that Swarmfile serves. `git fetch` brings the clone up to date, and `git push` sends your commits back as Swarmfile commits (fast-forward only — see Pushing), and an existing repository can be pushed into a new repository. Starting from a repository that already exists elsewhere? `swarmfile git import` creates the repository from it in one command.

> New to how git and Swarmfile fit together? Git and Swarmfile compares this with the git-LFS server and the CLI's git-style commands.

Turning on git access

Git access is a per-repository setting that covers every branch and tag. A repository owner or org admin turns it on:

1. In the web dashboard, open **Repositories**, use the repository's menu, and choose **Git access…**.
2. Read the warning, tick **I understand this cannot be turned off or re-tuned**, and select **Enable git access**.

The desktop app has the same switch: with the repository mounted, open **Repository settings → Git access → Enable git access…**. Or from the terminal: run `swarmfile git enable`. Both freeze the same record and print what was frozen.

Git access is off until an owner enables it. A member who tries to clone, push, or index a repository without it gets the refusal, and the org's owners and admins get one notification asking them to enable it, so the request doesn't go silent.

It's a one-way switch. At the moment you enable it, Swarmfile fixes the rules it uses to turn the repository into git: the conversion version, the size above which a file becomes an LFS pointer (1 MiB by default), and where history starts. Those rules never change afterward, because changing any of them would change every commit id a clone has already seen. If a different large-file boundary matters for the repo, set it in the same breath as enabling — `swarmfile git enable --threshold-bytes <N>` freezes that value instead of the 1 MiB default.

Once enabled, the dialog shows the frozen settings instead of the switch, plus the exact `git clone` command to copy. That command names the repository by its id — the form that also works on a CI machine signing in with a repository API key.

Before you enable it, two things must be true, and the hub refuses the enable (and says which) if they aren't:

- **The repository keeps every saved version.** Git access needs the repository's full history to stay restorable. Turn it on with **Keep every saved version** in the repository's history settings, or `swarmfile repository set-keep-full-history true` (see Trash, History & Rollback).
- **Every large file has a recorded id.** A file at or above the LFS threshold reaches git as an LFS pointer, and a pointer needs the file's SHA-256. Files saved through the Desktop App get one as they're saved; older files may not. `swarmfile git status` shows how many are missing, and `swarmfile git enable` records any that are missing automatically before it freezes the view (run `swarmfile git backfill-oids` ahead of time if you want to see the work finish first).

The enable accepts an **end-to-end-encrypted** repository through the desktop engine helper: the helper unwraps this device's repository key, the view is all-inline (E2E has no LFS), and the hub never sees plaintext. The standalone/API-key helper supports it too when given this device's `SWARMFILE_USER_KEY_PRIVATE_B64` and `SWARMFILE_USER_KEY_ID`. The enable is also refused when a branch or tag head can't be read, and when the repository's history is too long to check in one pass.

Where history starts. When you enable git access, Swarmfile walks back from every branch and tag. If some older commit's content is no longer stored (for example, history from before the repository kept every saved version), history is cut there: that commit appears in git as a commit with no parent. Everything from that point forward is complete.

Want to see a clone without a Swarmfile account first? The Zero-Install Demo clones a public ~60 GiB LFS-backed repository with stock `git` over HTTPS.

Cloning

```
git clone swarmfile://acme/website          # by org and repository slug
git clone swarmfile://acme/website site     # into a directory of your choice
swarmfile git clone acme/website site       # the same, through the CLI
```

The forms the URL takes:

URL	Meaning
`swarmfile://<org>/<repository>`	The repository on the Swarmfile service your Desktop App is signed in to (or `SWARMFILE_HUB_URL`). Slugs or ids both work.
`swarmfile://<hub-host>/<org>/<repository>`	The same repository on a specific Swarmfile service. HTTPS, except that a loopback host (or a host an operator lists in `SWARMFILE_GIT_INSECURE_HOSTS`) may use HTTP. The org segment is required: `swarmfile://<hub-host>/<repository-id>` is read as a relative URL whose org is the host.
`swarmfile://<repository-id>`	A repository by its id, in your configured organization.
`swarmfile://<hub-host>/fed/<peer>/<org>/<repository>`	A **federated public read**: `<hub-host>` is a server that has allowlisted the origin `<peer>`, whose public plaintext repository is `<org>/<repository>`. The session is anonymous and read-only, and no local credential is sent. See Federated public reads.

`swarmfile git url <repository>` prints the URL for a repository, handy for `git remote add origin "$(swarmfile git url acme/website)"`. `<repository>` can be `<org>/<repository>`, a repository id, or a bare repository name — a bare name is qualified with your configured organization.

The printed URL names the hub (`swarmfile://hub.example.com/acme/website`) whenever your configured hub can be written that way, so a copied link means the same repository on every machine that can reach that host (and, for a plaintext host, that also allows it); a path-prefixed or non-allowlisted plaintext hub keeps the relative form. A relative URL still resolves against whichever hub the receiving machine is configured for.

In a URL, each segment is the org's or repository's **web address** (the lowercase slug, e.g. `yadc` for a repository named YADC) or its id. A slug is matched ignoring case; when no address matches, a display name that is unique among your orgs (for the org segment) or the org's repositories (for the repository segment) also resolves — a name matching more than one is refused with the candidates listed, rather than guessed.

Renaming a repository or an org keeps the old slug as a reserved alias: an old `swarmfile://` URL keeps resolving for a signed-in session and for public/anonymous clones. A repository API key resolves an old **public** repository slug through the public directory (which carries historical slugs); a **private** repository has no public listing, so a keyed private remote should use the org/repository **ids** (the web Git access panel copies that form).

What you need installed. Cloning uses a small git *remote helper*, `git-remote-swarmfile` , which every Desktop App installer ships (on macOS the installer also links it into `/usr/local/bin` ; on Linux it's in `/usr/bin`). No running drive is needed. If `git` says it can't find a helper for `swarmfile` , use `swarmfile git clone` , which finds the helper next to the `swarmfile` binary even when it isn't on your `PATH` . Git 2.11 or later is required.

Standalone helper. The open-source `swarmfile-git-remote` build (published with `swarmfile-server`) can serve the same `swarmfile://` URLs without the Desktop App. It deliberately does **not** read the app's `config.json` , so give it the hub and credential explicitly: `SWARMFILE_HUB_URL=https://hub.swarmfile.com` , an org/repository in the URL (or `SWARMFILE_ORG_ID` / `SWARMFILE_REPOSITORY_ID`), and `SWARMFILE_API_KEY` . Install it as `git-remote-swarmfile` on `PATH` ; its `--help` lists the modes.

There's no branch in the URL: a clone fetches every branch and tag, with the repository's default branch as HEAD — `main` unless its default branch is something else: a different initial branch at creation, or the branch an import brought with it. Use git's own options to narrow it, for example `git clone -b design-b --single-branch swarmfile://acme/website` .

Federated public reads

Two allowlisted servers can federate **public plaintext** repository data without copying it. The origin server exports a repository to a named peer; the federating server pins that peer's signed host identity and allowlists the repository. The hosted Swarmfile service can also be one side: staff pin/import the peer and, for a hosted origin, the repository owner enables **Repository settings → Public page → Hosted federation export** on the web (see Public repositories). A clone through the federator is anonymous and read-only:

```
git clone swarmfile://server-a.example/fed/server-b/acme/widget
```

- Only the origin's **public plaintext** data is served; managed/E2E repositories, ACLs, member data and keys are never federated, and no byte is copied or cached on the federating server.
- The origin's committed `.swarmfile/publicignore.yml` applies to federated reads too: if it hides a path in a served tree, the origin refuses the whole public surface with the same 404 rather than filtering the clone. See Self-hosting → Federating public repositories.
- The federator carries the origin's bytes, status and signed history unchanged — it does not re-sign anything. A portable provenance manifest written against the origin verifies against the federated clone (see Verifiable history).
- git-LFS **downloads** are carried through a federated read (the origin's object store is served through the federator); LFS **uploads and locking** are not federated. If a clone cannot fetch an LFS object, clone from the origin or use its release archive.
- A federated read cannot push. To contribute, clone from the origin with a credential and push there.
- The origin URL is unlisted, but the repository is public: exporting it makes its public reads answerable to anyone who knows the URL. An operator can additionally require a signed request from the federating peer (`admin federation repository allow --export --require-actor`); then only the federator's signed reads work and a direct public link is refused. Operators set this up with `swarmfile-server admin federation peer add` and `admin federation repository allow` ; see Self-hosting.

On a protected repository, your **current** access governs what git can materialize: a teammate added to the repository can clone its whole history, including commits made before they joined — the derived ids depend on tree content, and the hub checks permissions as they are now. The hub separately remembers what access existed at each commit's time, which is what keeps point-in-time history views faithful; that replay isn't what git reads. If your access is revoked, git reads stop too, just like the file browser.

Figure omitted in print (animation): An owner enables a repository's git surface with one command; an ordinary git clone speaks swarmfile:// through the helper; a commit pushes fast-forward; and swarmfile-verify-history walks the chain it just created. - https://swarmfile.com/demo/oss-git.cast

Signing in

The helper looks for a credential in this order:

1. **`SWARMFILE_API_KEY` .** A repository API key in the environment wins over everything else, including a Desktop App session, and the helper says so when both are present.
2. **Your Desktop App sign-in.** On a machine where you're signed in to the Desktop App and no explicit key is set, `git clone` just works.
3. **A repository API key from git's credential store.** On a machine with no Desktop App session (a CI box, a container), store a repository API key (`sf_key_…`) as the password for the Swarmfile host with `git credential approve` , and the helper picks it up when no explicit key or session is present.

A key is repository-scoped, so for a **private** repository use its **id** in the URL — `swarmfile://<org>/<repository-id>` , or the host-qualified `swarmfile://<hub-host>/<org>/<repository-id>` . A repository key also cannot resolve a private org's **slug**, so use the org **id** for that segment in a CI URL (`swarmfile://<org-id>/<repository-id>`); the web Git access panel copies exactly that form. Resolving slugs needs a session; the refusal says which id to use. A stored credential (option 3) is never silently preferred over an explicit `SWARMFILE_API_KEY` .

A **branch-scoped** repository key can clone and push only its branch and needs the client's `SWARMFILE_BRANCH` pin (the helper does not infer it yet); a **path-scoped** key cannot use git at all (it is engine/API-only). A keyed clone still follows the access you hold today.

Credentials stay with their authority. A Desktop App sign-in is only used for the hub it was made against; a host-qualified URL that names another server is cloned anonymously (public repositories) or falls back to a credential stored for that host. An explicit `SWARMFILE_API_KEY` is global, so it is sent only to the configured hub

unless you set `SWARMFILE_ALLOW_CROSS_HUB_KEY=1` — and with **no** configured hub, a host-qualified URL also requires that opt-in (the URL alone does not authorize the global key; both the standalone and engine helpers enforce this). Unset it before cloning a link to an authority you do not control.

On the hosted hub a **repository key cannot list the org's repositories** (by design), so resolving a renamed **repository slug** by name needs a signed-in session or the anonymous public clone; the helper also asks the hub's one-request `/public/<org>/_resolve/<repository>` route, so a public repository past the first directory page resolves. A repository **id** always works.

For a scripted sign-in, `SWARMFILE_OIDC_REFRESH_TOKEN` works and is **not** one-shot: the IdP rotates refresh tokens on use, but the live successor is persisted under the Swarmfile cache directory, and later commands — the push after a clone, a restarted engine, any other git command — reuse that stored successor instead of the spent environment value. Supplying a *different* token later still wins. A stored Desktop App session or a repository API key remains the simplest CI credential (no cache-dir file to carry between steps).

A clone needs to read the **whole** of every commit it fetches, and the hub checks permissions against the access you hold **today**. If a folder is restricted for you now, it can't hand those commits over complete, and a clone or fetch that has to walk one is refused with a message saying so rather than producing a repository with silent holes — grant read and the same clone succeeds, including commits made while you were restricted.

(The hub also records what access existed when each commit was made; that replay answers explicit point-in-time history views, not git.) The mounted drive and `swarmfile materialize` follow your access today in the same way. Use an owner's or a fully-permitted member's credential, or a repository API key, for clones of restricted content.

What a clone contains

- **Branches and tags.** Every non-empty branch becomes a git branch; tags become git tags. A lightweight tag points at its commit; an annotated tag is re-emitted as the exact tag object that was pushed, so its message, tagger, signature and id survive. An empty repository clones as an empty repository.
- **Commits.** Each Swarmfile commit becomes a git commit, with your message and a `Swarmfile-Commit:` trailer naming the Swarmfile commit it came from. A commit with no message reads `(no message)`. Authors are recorded by account id; the helper writes a local mailmap so `git log` shows the names of people in your organization.
- **Unsaved-to-a-commit work.** If a branch has saves since its last commit (autosaves that haven't been rolled into one yet), the clone ends that branch with one extra commit, authored by `swarmfile` with the message "swarmfile snapshot of the live tree", so what you clone matches what's on the drive.
- **Large files as git-LFS pointers.** Files of 1 MiB or more come through as LFS pointers, with a generated `.gitattributes` in each folder that has them (added after any `.gitattributes` you already had). The clone is configured to fetch LFS content from Swarmfile automatically — see Git-LFS for the transfer agent and performance notes.
- **Symlinks** stay symlinks. A symlink's target must stay inside the repository; the drive and the UI validate it at the link's real depth. A `git push` create has no mount depth at upload time, so it uses the stricter rule and refuses an absolute target or one with a leading `..` (an internal `../` link the drive would accept at depth can be refused there; a moved link is checked at its destination depth, and a moved directory's descendants at the strict rule). **Empty folders** aren't carried (git can't store them). **Executable files** (marked with `chmod +x` on the macOS or Linux drive, or `swarmfile chmod +x`) arrive as mode `100755`; every other file is `100644`.
- **On Windows**, a path Windows can't create stops the clone with a message naming it.

Commit ids are stable: two people who clone the same repository get the same ids, and a later fetch never rewrites what you already have. A clone's `git log` shows git's own SHA-1, not this Swarmfile id; the web **Commits** page shows the reported git SHA-1 on a second line under the Swarmfile id (marked *unconfirmed* until a second principal's report confirms it).

Staying up to date

Run `git fetch` (or `git pull`) in the clone. Only new commits are converted and downloaded, so a fetch after a day's work is quick. New branches and tags appear like they would from any remote.

If a branch is being saved to while you fetch, the fetch may ask you to retry shortly — it won't build a snapshot of a tree that's still moving.

When a branch's newest commit can't be served yet. A commit enters the git view once it has been *derived* (the Desktop App does this after each commit, or run `swarmfile git index`).

A provisional commit (derived but not yet confirmed) is served like any other to a signed-in clone — an anonymous clone serves confirmed history only, so a head that is still provisional is left out there with its reason until it confirms (normally within seconds of the push's claim on the hosted hub; a community server also confirms a member key's push, by rebuilding the tree from its own rows when the helper claims it (a claim the server cannot reproduce still waits for another key) — see How a pushed commit gets confirmed).

A head that has never been derived is derived and claimed on the spot when the credential running git may write — a signed-in app or a key heals the branch, and the command then works (any clone, fetch, push or `ls-remote` heals it; a clone with `swarmfile.confirm=false` has opted out).

When it can't (an anonymous or read-only clone, an opted-out clone, or a chain longer than the helper derives in one command), or when the claim is disputed, the branch is left out of the branch listing with that reason printed, rather than serving a stale pointer. Clones and fetches of the repository's other branches still work, and asking for that branch by name fails. The lasting fix for those is `swarmfile git index` from a machine with the Desktop App; it derives the selected drive's branch.

A disputed claim

A disputed claim means two derivations of the same commit disagree. Swarmfile then derives the commit again on the server from its own records and keeps whichever claim matches, so a dispute usually resolves on its own within a minute or two and the branch serves again.

Settling a rare dispute

The steps below are for the rare dispute where the server's derivation matches neither claim. Running `swarmfile git index` shows which hash your machine reproduces without submitting anything, and a repository owner or admin can read both recorded values with `swarmfile git disputes <commit number>` or in the **Commit disputes** section of the repository's Commits page on the web.

They then reset the commit with `swarmfile git clear-claim <commit number>` or the **Clear** button on that page, using the number the error names. The dispute record is kept after the clear.

Swarmfile then derives the commit again on its own (running `swarmfile git index` does the same, sooner), except when Swarmfile's own derivation was part of the dispute: then the commit waits for `swarmfile git index` from a member's machine.

Owners and admins are notified when a dispute is settled. A settlement that left the commit provisional can be overridden with the same clear; a settlement that confirmed the commit is final. Each settlement and each clear is recorded, with the values involved, in the **Recent changes** panel of the repository's Permissions page (with the whole repository selected).

Pushing

`git push` sends commits from the clone back to the repository. Each git commit becomes one Swarmfile commit, and Swarmfile keeps the commit's exact author, committer, dates, message and any signature, so its id doesn't change: anyone who clones or fetches afterwards gets the same commit ids you have. Pushing works on managed, unencrypted and end-to-end-encrypted repositories — the latter through the desktop engine helper or the standalone/API-key helper with this device's user-key environment.

```
git push origin main                # fast-forward an existing branch
git push origin my-feature          # a new branch is created on Swarmfile
git push origin :my-feature         # archive a branch
git push origin v1.0                # a lightweight tag becomes a Swarmfile tag
git push -f origin v1.0             # move the tag to the commit you tagged
git push origin :refs/tags/v1.0     # delete the tag
git push -v origin main             # also print each commit's planned changes
git push --dry-run origin main      # run every check, write nothing
```

On a hub that predates the targeted protection read, `git push --dry-run` of a new branch reports its protection as **unknown**, not unprotected: the hub cannot evaluate a glob rule for a name with no branch row yet. The real push still applies the hub's refusal if a rule matches.

A successful push prints a link to the repository's commits in the web app (or to the merge request, below).

What a push accepts:

- **Fast-forwards only.** Your commits must build on the branch as Swarmfile has it, one first parent after another. A force push whose history shares no commit with Swarmfile — a rewritten root — is refused, and a helper-created clone translates an ordinary force push (a rebase plus force included) into a forward commit by default (see below); an older clone can opt into that translation with `swarmfile.force-forward`. A `git merge` of the remote into a diverged local branch is also refused — a merge commit must sit *on top of* the remote head, so fetch and `git pull --rebase` instead. If someone saved to the branch since you last fetched, git reports `fetch first`: run `git pull --rebase`, then push again. Saves on the drive that haven't been rolled into a commit count too, since a fetch turns them into a snapshot commit. If you need the branch's *tree* to be yours anyway, without rewriting history, run `swarmfile git force-forward` in the clone: it pushes a forward commit whose tree is your tip's (your commits keep riding the auto-archived `merged/<sha>` side branch, so nothing is lost and teammates fast-forward) and prints the `git fetch <remote> && git reset --hard <remote>/<branch>` that re-aligns your clone. A rewritten **root** commit (an amended initial commit) cannot land this way — the replaced history shares no commit with Swarmfile, so the push refuses naming the unrelated root; push that as a new branch instead.
- **New branches.** Pushing a branch Swarmfile doesn't have creates it, starting from the newest commit in your history that Swarmfile already holds.
- **Merge commits** with two parents. If the history you merged isn't on Swarmfile yet, the push sends it first, to a branch named `merged/<first 12 characters of its newest commit>` that starts where that history left the main line (an existing one is reused). The push lists each branch it creates. Once the merge itself lands, the push archives the branch; its commits stay reachable through the merge, so the history doesn't change. If those commits aren't confirmed yet (below), the branch stays until a teammate's fetch has confirmed them, and your next fetch or push archives it then. A `merged/*` branch someone has added commits to is left alone. This works for merges nested inside merged history too. History that shares no commit with the repository at all (a merge of an unrelated repository) is refused, naming the commit. An octopus merge (three or more parents) is accepted when every parent after the second is already on Swarmfile — push those branches first; an extra parent that isn't on the hub is refused with that instruction.
- **Protected branches** don't move. Your commits go to a branch named `push/<branch>/<first 12 characters of your commit>`, started at the protected branch's head, and the push opens a merge request into the protected branch, or reuses the one already open from that branch (also when another push opened it a moment earlier: the push links that merge request instead of failing). Git reports the ref as not updated, with `opened merge request #<n> (push/<branch>/<commit>)`; the link takes you to it. The same rules apply to `merged/*` branches: if one is protected, the push is refused.
- **Root creates need root authority.** On a protected repository, a commit that adds a new name directly at the repository root — or moves one there — is refused unless you're an org owner, the repository's creator, or hold a repository-wide `write` (or higher) grant; a folder-scoped grant doesn't qualify, and overwriting an existing root file doesn't need it. The push names the remedy: move the files inside a folder and push again, or ask an owner for repository-wide write access. See Permissions.
- **Deleting a branch** (`git push origin :<branch>`) archives it, and it can be restored from the web app. `main` and protected branches can't be deleted this way.
- **Tags.** A lightweight or annotated tag on a commit Swarmfile holds becomes a Swarmfile tag on that commit; an annotated tag's object is stored and re-emitted verbatim, so `git tag -a` and `git tag -s` keep their message, tagger, signature and id. To move an existing tag, retag and push with `--force` (`git push -f origin <tag>`); the commit must already be on Swarmfile, and replacing an annotated tag with a different object or a lightweight tag also needs `--force`. `git push origin :refs/tags/<tag>` deletes a tag. A tag that a release was published from can be neither moved nor deleted.
- **Large commits.** A commit that changes more than 5,000 paths is sent in parts and applied as one commit, up to 2,000,000 paths. The branch is locked while it applies. Above 100,000 paths Swarmfile applies the commit in stages after the upload, and the push shows its progress. Anyone reading the branch during that time can see the files that have been applied so far. If the push is interrupted after the parts were sent, push again: it picks up where it left off. If someone else's change gets onto the branch while a staged commit is applying, Swarmfile stops the commit. The push then reports how far it got, and the changes already applied stay on the branch as uncommitted changes for you to review. To undo them, use **Roll back** on the notice the web app's Commits page shows for that branch: every file the push touched goes back to how it was before the push, as one new commit you can revert. The person who pushed and repository owners can roll back, and the rollback is refused if any of those files has changed again since the push stopped.
- **A commit already on another branch** (for example a fast-forward of `main` to your feature branch's tip) is recorded again on the target branch with the same commit id, so both branches show the same history.

If your history diverged. Two shapes are the same problem: the branch moved while you worked, and a `git merge` of the remote into your line cannot be pushed — a push must continue the remote head along its first parents:

```
A---B---C        the branch on Swarmfile (C is the head)
     \
      X---M      your clone: X, then M = merge(X, C)     <- refused
```

`git pull --rebase` is the supported way back: your work replays on top of `C` as `A---B---C---X'`. When you need your merged *tree* as it stands, `swarmfile git force-forward` lands it as a new commit `F` on top of `C` instead — nothing is rewritten, `C` and `X` stay reachable, and everyone else fast-forwards:

```
A---B---C---------------F     F's tree = M's tree
     \                 /
      X-------------M
```

A clone created through the helper also makes plain `git push --force` (and `--force-with-lease`) take the same path by default (`swarmfile.force-forward=true`); an older clone can opt in with `git config swarmfile.force-forward true`, or set it false to keep the refusal. Because git's own bookkeeping then points your remote-tracking ref at the tip you sent rather than the commit that landed, follow the note the helper prints — `git fetch`, then reset your branch to the fetched one — before pushing again.

Push an existing repo into a new repository. A repository that has never been on Swarmfile goes into a repository where the branch you're pushing to (`main` in this example) has no commits and no files yet, with its whole history:

```
git remote add origin swarmfile://<org>/<repository>
git fetch origin                    # configures git-LFS (and its credential) for Swarmfile
git lfs push --all origin main      # if the repository uses git-LFS
git push origin main
```

The first commit lands with no parent, merges bring their merged history along as described above, and a fresh clone afterwards has the same commit ids. The push sends consecutive commits together, up to 100 commits or 5,000 changed paths per request, within the plan's request budget. Uploading file content and indexing each commit take most of the time. When the budget does run out, the push waits for it and carries on. A `main` that already has files but no commit can't take a root commit: fetch first, and the push builds on the snapshot.

What a pushed commit may contain. Swarmfile accepts a commit only if it is exactly what a clone of the result would produce, so `git fetch` never rewrites it. The push checks every commit, and asks the hub to check the first one, before uploading anything, and names the file and the fix when one doesn't fit:

- **Submodules.** A gitlink (mode `160000`) is pushed and cloned with its exact target commit id; the mount hides the submodule path (it has no bytes in this repository) and recursive submodule clones are not rewritten (the submodule URL is external). An executable file (`100755`) is fine: it lands as an executable Swarmfile file, and a commit that only flips the bit (`git update-index --chmod=+x|-x <file>`) is a change like any other.
- **Large files must be LFS pointers.** A file of 1 MiB or more (the repository's frozen threshold) must be committed as a git-LFS pointer, and its content must already be on Swarmfile. The clone is configured to send LFS content to Swarmfile with the same credential the clone uses (your signed-in Desktop App session, a stored `sf_key_` key, or `SWARMFILE_API_KEY`; nothing to set up), so git-LFS's own pre-push step normally takes care of that; if a push says an LFS object is missing, run `git lfs push origin <branch>` and push again. A file under the threshold must be committed as a regular file, not as a pointer.
- **Each LFS file is covered in its folder's `.gitattributes`**, either by a `git lfs track` pattern in that folder's own `.gitattributes` (for example `*.psd filter=lfs diff=lfs merge=lfs -text`), or by the line a clone writes for a file the patterns don't cover: `/<name> filter=lfs diff=lfs merge=lfs -text`, one per file, sorted by name, at the end of the file after the line `# swarmfile: generated LFS attributes — view v1`. Only patterns in the file's own folder count, not ones in parent folders, and a pattern with a `/` in it covers nothing. That matters for `git lfs track`: run at the repository root as `git lfs track "media/*.psd"`, it writes a pattern with a `/` into the root `.gitattributes`, which covers nothing here. Put the pattern in the folder's own file instead, for example `printf '*.psd filter=lfs diff=lfs merge=lfs -text\n' >> media/.gitattributes`. A `lockable` attribute on the line (what `git lfs track --lockable` adds) is fine. When a folder's lines don't match, the push prints the lines it expects.

What happens to the files on the drive:

- Changed, new and deleted files become the same changes on the drive. A file renamed without changes keeps its Swarmfile history, and so does a whole folder renamed or moved without changes: it moves as one folder.
- Folders you add are created, and folders your commit empties are removed. Empty folders that exist only on the drive (git can't show them) are left alone.
- A push of several commits sends them in batches, and each commit is applied as a separate Swarmfile commit, in order. If one is refused, the commits before it have already landed, and the branch stays at the last of them. Fix the refused commit and push again to continue from there.
- Commits you push are *provisional* until a second, independent derivation confirms them. Swarmfile's hosted hub queues its own confirming derivation the moment the push's claim lands, so a pushed commit normally confirms within seconds; a community server confirms a member key's push from its own tree derivation at claim time (a claim it cannot reproduce waits for another key's claim, and on a community server an **owner key's own first claim** self-attests directly — there is no self-confirmation setting there; a teammate's fetch also confirms). On the hosted hub, a repository owner with self-confirmation turned on likewise confirms their own first claim. A push whose helper died before claiming is caught by a couple-of-minutes backstop, which derives the commit but still needs a client's matching derivation to confirm it. Provisional commits are cloned and fetched like any other. While one is provisional, its row on the repository's **Commits** page shows no commit hash — hovering the dash explains why — and the hash appears on the page's next refresh once confirmation lands (the page keeps re-checking for a couple of minutes after a push).

How a pushed commit gets confirmed

A commit is confirmed when a second person works out its file tree independently and gets the same result as the person who made it. You don't have to do anything for that to happen:

- **Swarmfile's server confirms what a signed-in app already claimed.** As soon as a `git push`'s claim (or a snapshot mint by the Desktop App) lands, the hub queues its own derivation and confirms the commit when the result matches — no teammate's fetch needed. A commit no app ever claimed (a web upload, a revert, a merge) is derived by the server but still needs a client's matching derivation (the routes below) to confirm.

- **A teammate's `git fetch` or `git clone` confirms what it brings in.** After the fetch, the helper recomputes each new provisional commit that someone else made, from the branch's history as of that commit, without reusing their results, and sends the result to Swarmfile. Parents go before their children, so a whole chain of commits is confirmed in one fetch. One fetch confirms up to 200 commits, oldest first, and the next fetch continues. A fetch never confirms your own commits, a shallow clone only confirms the commits it fetched, and if confirming fails (for example because of a rate limit), the fetch still succeeds and the next one tries again.
- **`swarmfile git index` confirms a whole branch at once** on a machine with the Desktop App. With no argument, it confirms the branch head and every unconfirmed commit behind it, oldest first, back to the nearest commit that is already confirmed (or one you made yourself). It handles up to 1,000 commits in one run and lists what happened to each commit. `swarmfile git index <seq>` handles just that one commit.

If the two results don't match, the commit can't be confirmed. Swarmfile records both results for the repository owner to review. A fetch prints a warning and still succeeds. `swarmfile git index` names the commit and exits with code `2`. Commits that come after it in the same run aren't confirmed.

To keep a clone from confirming commits (a read-only mirror, say), turn it off in that clone:

```
git config swarmfile.confirm false
```

or clone with `git clone -c swarmfile.confirm=false swarmfile://<org>/<repository>`.

The same setting also turns off the on-demand derivation of a branch head that has never been derived: a git command in that clone then leaves the branch out of the listing with the `swarmfile git index` guidance instead of claiming it.

When a clone fails

The helper always says why: its message prints above git's `fatal: remote helper 'swarmfile' aborted session` line, and the same text is written to `git-remote-last-error.log` in the Swarmfile cache directory (`~/.cache/swarmfile` on macOS and Linux, `%LOCALAPPDATA%\Swarmfile` on Windows) — include it if you contact support.

The common causes: git access isn't turned on for the repository (Turning on git access); the repository is private and the signed-in account isn't a member of its organization (a **public** repository with anonymous cloning on doesn't fail this way — a credential the hub refuses is named in a one-line notice and the clone continues anonymously, see below); or the repository name in the URL doesn't resolve (use `swarmfile git url <repository>` to print a URL that does).

If a clone gets into a bad state

A clone is a local, derived view: every commit id is recomputed from the repository's branch state, so a clone you have wrecked — bad refs, a half-finished rewrite, a corrupted object store — can always be replaced. Only work that exists nowhere else is worth rescuing first.

- **Save unpushed work**: `git bundle create rescue.bundle --all` (one file you can move anywhere), or `git format-patch origin/<branch>..HEAD -o ../patches`. To land it on Swarmfile instead, push it to a throwaway branch — `git push origin HEAD:rescue/<name>` creates a new branch forked at the newest commit Swarmfile already holds.
- **Re-clone**: `rm -rf <clone> && git clone swarmfile://<org>/<repository>` gives a pristine view with identical commit ids. Generated clone config (git-LFS wiring, `swarmfile.confirm`) is recreated.
- **Repair in place**: `git fetch origin --prune`, then `git checkout -B <branch> origin/<branch>` (or `git reset --hard origin/<branch>`) and `git clean -fdx`; `git fetch --unshallow` if the clone is shallow; `git replace -d <ref>` followed by `git reflog expire --expire=now --all && git gc --prune=now` drops local rewrites.

If a bad state reached Swarmfile itself (junk branches, a moved tag, a bad commit), a local reset won't undo it: archive the junk branch (`git push origin :<branch>`, restorable from the web app), push the tag back (`git push -f origin <tag>`, unless a release used it), commit a revert forward, or ask an owner to clear a stuck commit claim with `swarmfile git clear-claim <seq>`.

Limits and caveats

An octopus merge whose extra parents are not on Swarmfile	Push the side branches first, then the merge. A merge of three or more branches whose parents are all on the hub pushes and clones with its exact parent order.
A force push with a rewritten root	A helper-created clone translates an ordinary force push (a rebase plus force included) into a forward commit by default; a force push whose history shares no commit with Swarmfile (a rewritten root) is refused, and an older clone refuses force pushes until `swarmfile.force-forward` is enabled. Swarmfile history is append-only; push new commits instead. To land a force push's end state — the branch's tree is yours — as a forward commit, run `swarmfile git force-forward` in the clone, or set `git config swarmfile.force-forward true` on an older clone (set it false to keep the refusal) so ordinary force pushes are translated (see Pushing). Tags can be moved with `--force` and deleted, unless a release was published from them. See Pushing for what a push accepts.
Pushing to an end-to-end-encrypted repository	Supported by the desktop engine helper and by the standalone/API-key helper with this device's user-key environment (the view is all-inline; LFS stays refused).
Shallow clones	`--depth`, `--deepen` and `--unshallow` work once the commits at the cut have been indexed (`swarmfile git index`). A depth fetch also needs those boundary commit ids *confirmed*; on a repository only one person derives, set `SWARMFILE_GIT_ACCEPT_PROVISIONAL=1` where you run the clone (or let a second person confirm them: a teammate's fetch, or `swarmfile git index` — see How a pushed commit gets confirmed). `--shallow-since` and `--shallow-exclude` aren't supported.
Partial clones	`--filter` is not supported through the `swarmfile://` helper; large files already arrive as LFS pointers. The read-only HTTPS gateway supports `--filter=blob:none` for public plaintext repositories.
Anonymous clones	Read-only, and **on by default** for a public, git-enabled repository. An owner can turn it off from **Repository settings → Public page** (web and desktop), or `swarmfile git anonymous-clone off`. While on, a signed-out visitor can clone, fetch, `git lfs pull`, and read — but never push. Anonymous clone is unavailable while any active branch's committed tree carries a `.swarmfile/publicignore.yml` that hides a path in it, any tag's committed tree carries one that hides a path, or a ready (published) release hid paths; an already-published older release keeps the contents it was built with. The check reads both committed trees a branch can advertise (its confirmed head changeset tree and its live head tree — they differ for a fork taken from a dirty source) and every tag's committed changeset tree. An invalid or unreadable policy, or a boundary read/walk error, refuses anonymous clone (fail closed). There is no filtered clone, because hiding paths at the same commit ids would contradict the public history (a change takes effect within about a minute; a path that exists only in an older commit is not retroactively hidden). A clone also contains the commit and annotated-tag messages and the committed `.swarmfile/publicignore.yml` itself verbatim — filtering either would change commit ids — so never put a secret in a public commit message. The web commit list and the federation activity/commit feeds redact path names the policy hides; a git clone does not. The boundary is anonymous-only — member clones are governed by the ACLs described above, so a restricted folder can still refuse one. Publishing a release that would hide paths requires anonymous cloning off; once the release is ready, turning it back on does not re-expose the paths while the ready release or an active branch's committed tree still hides them. A signed-in account the hub refuses for the repository (not a member of its organization, a revoked or expired key, or a key scoped elsewhere) gets the same anonymous read: the helper prints one line naming why (the refusal, or the lookup failure for a repository the account can't see yet), then continues read-only.
Fetching a commit by id	Only branches and tags can be fetched, not an arbitrary commit id.
A stock `https://…` git URL	The hosted hub does not serve one. An operator can run the read-only `swarmfile-git-gateway` (public plaintext repositories; stock clone/fetch including `--filter` and sparse checkout, a download-only LFS proxy, no upload/lock/push, TLS at a reverse proxy) and clone `https://<gateway>/<org>/<repository>.git`; otherwise clone through the `swarmfile://` helper (which connects to a host-qualified hub over HTTPS, or HTTP for loopback/an allowlisted host).
End-to-end-encrypted repositories	Supported by the desktop engine helper and by the standalone/API-key helper with this device's user-key environment (the view is all-inline; LFS stays refused).
Branch or tag names git can't represent	Refused with a message naming them.

Two kinds of problem tell you how to fix them: a large file that has no recorded id (enable records missing ids automatically, or run `swarmfile git backfill-oids` yourself), and a clone made against a different git view version must be re-cloned (the helper says so).

See also

- Clone and push a Swarmfile repository with git — the announcement post, with the end-to-end walkthrough.
- Git and Swarmfile — every way git and Swarmfile meet, side by side.
- Coming from Git — the command cheat sheet for day-to-day work on the drive.
- Runner (Headless CI) — CI that runs on a Swarmfile branch without a clone at all.

The Zero-Install Demo

A public Swarmfile repository (about 60 GiB of LFS payload) that answers the question "how big can a git clone feel small?" with stock git over HTTPS — no Swarmfile client, no account, no API key.

```
git clone --filter=blob:none --no-checkout https://git-demo.swarmfile.com/swarmfile/large-demo.git
cd large-demo
git sparse-checkout set sample
git checkout main
git lfs pull --include='sample/**'
```

The repository has a small `sample/` subtree (README, generated thumbnails, a point sample and one LFS-tracked preview) and `large/` — tens of GiB of synthetic tensor parts, every one of them a git-LFS pointer in the git history.

What just happened

- The clone asked for **history and trees only** (`--filter=blob:none`), so it did not download the `large/` file bytes or even the sample file contents.
- `git sparse-checkout set sample` limited the working tree to `sample/`, and `git checkout` fetched just those files.
- `git lfs pull --include='sample/**'` fetched the sample preview's LFS object through the same gateway. Fetch one large part the same way:

```
git lfs pull --include='large/part-000.bin'
```

The 50–100 GiB of LFS payload is served by the origin and proxied per request; the gateway's mirror holds only the git objects (the pointers). That is why the clone is small and the bytes you ask for are the only bytes you wait for.

Without git-lfs

The partial clone and sparse checkout work with plain git; the large files stay pointers and the sample's inline files arrive:

```
git clone --filter=blob:none --sparse https://git-demo.swarmfile.com/swarmfile/large-demo.git
cd large-demo
git sparse-checkout set sample
git checkout main
```

Install git-LFS when you want the preview or a large part.

Measured

Step	Measured
First byte (`info/refs`, before the clone)	—
Clone (`--filter=blob:none --no-checkout`, before LFS)	—
Sparse checkout (`sample/`)	—
`git lfs pull --include='sample/**'`	—
Large-object LFS batch (`large/part-000.bin`)	—
Measurement date / runner region	—

The numbers are produced from a clean runner by the same check that guards the demo:

```
node scripts/ops/git-demo-canary.mjs \
  --url https://git-demo.swarmfile.com/swarmfile/large-demo.git \
  --region <runner-region> --json
```

The first byte is measured BEFORE the clone (so the clone does not warm the gateway) and the clone's bytes BEFORE any LFS object is fetched; the LFS pull is timed separately. The launch measurement records its date and runner region with the numbers. Until that run is published the table is intentionally empty rather than estimated.

What it is not

- It is **read-only**: push is not served over HTTPS (push stays the `swarmfile://` helper).
- It is **public plaintext**: the demo path is anonymous and the gateway only mirrors allowlisted public repositories. It is not a private, managed or end-to-end-encrypted repository and cannot be used as one.
- The data is **synthetic and clearly labeled** (deterministic generated bytes, no third-party dataset). It demonstrates throughput, not a model or a media collection.
- The demo is rate-limited and may be paused for maintenance. A temporary `429` / `503` is expected under heavy load or during a redeploy; retry shortly. A normal `git clone` failure is the honest signal.

See Cloning a Repository with Git for the `swarmfile://` clone and push paths, and Git-LFS for using Swarmfile as an LFS server in a repository hosted elsewhere.

Git-LFS

Swarmfile can act as a **git-LFS server**. Keep your source in the git host you already use — GitHub, GitLab, or self-hosted — and route only the large binary assets to Swarmfile, where they get real storage, deduplication, encryption at rest on a managed repository, and quota instead of bloating your git remote. LFS objects count as storage on your plan — deduplicated like drive content — and downloads are free.

This is the lowest-friction way to start: nothing about your VCS or your team's workflow changes. You keep `git push`; `git lfs push` / `pull` / `checkout` move the big files through Swarmfile.

> The fuller destination is making Swarmfile the home for the *whole* repository — native version control on a drive you mount, with no clone at all (see Coming from Git and Version Control). The LFS bridge is where you start; you can move the rest at your own pace.

Which repositories it works on

git-LFS works on **managed-key** and **unencrypted** repositories. It is **not** available on end-to-end-encrypted repositories: a stock `git-lfs` client can't hold a per-user encryption key, and the hub structurally can't read end-to-end content, so there is no one to encrypt or decrypt for. An LFS request against an E2E repository is refused up front with a `422` and a clear message rather than failing silently per object. See End-to-End Encryption for what that tier trades off.

On a managed repository, uploads must use the Swarmfile transfer agent. The agent uploads the object as small content-addressed blocks, which the hub encrypts at rest (the Desktop App's built-in agent seals them client-side; the standalone `swarmfile-lfs` agent's blocks are encrypted by the hub on write). The stock direct paths — a plain `git-lfs push`, a hand-crafted direct object PUT — are refused with a `422` and the typed error `lfs_agent_required`, because they would store the object as one unencrypted object. Install the agent once per machine (step 3 or Pushing large files) and retry.

Repository	Stock `git-lfs push` (direct)	Transfer agent (chunked)
Managed-key	**Refused** (`lfs_agent_required`)	**Encrypted at rest**, any size
Unencrypted (keyless / Free / public-visibility)	Allowed — plaintext by definition	Allowed — plaintext by definition
End-to-end	Refused (`422` ; LFS is unsupported)	Refused (`422` ; LFS is unsupported)

An **unencrypted** repository — every Free-plan repository and every public repository on any plan — is plaintext on every path by definition. Objects pushed through the chunked agent path also deduplicate against the same content on your mounted drive.

> **Older objects.** The agent requirement applies to new uploads. A managed repository that took stock pushes before this rule may still contain objects written the old, plaintext way until its operator-run migration reports clean — the hub converts each one to encrypted storage and verifies the bytes before removing the plaintext copy, and the migration is only clean when no plaintext candidate (including an object too large for its default per-object cap) remains. Until that migration reports clean for the repository, don't assume every existing LFS object is encrypted. Uploads started after the rule took effect are encrypted; a presigned single PUT admitted before the rule landed can still complete as an older plaintext object, but a multipart session created before the rule cannot complete — its part and complete requests answer the same `lfs_agent_required` 409 and abort the session. (End-to-end repositories never supported LFS at all.)

Importing an existing repository that uses git-LFS isn't supported: `swarmfile git import` refuses a default branch that carries LFS pointers and skips a side branch that does (see Known Limitations). Keep the repo on its current host and use the LFS redirect below instead.

Setup

1. Generate a git-LFS credential

In the dashboard, open your repository → the **git-LFS** tab → **Generate git-LFS credential**. This mints a repository-scoped API key (an `sf_key_…` string). Treat it like a password; it grants access to this one repository's files. It counts against your plan's headless-seat allowance (Free 1; Starter 2 per seat, minimum 5; Pro 5 per seat, minimum 10; Studio 10 per seat, minimum 20) — if the mint is refused, revoke an unused key or add a headless seat under Settings → Billing. One key per repository is the natural pattern: reuse it for that repository's LFS, CI, and scripts rather than minting a fresh one each time.

2. Commit `.lfsconfig`

The panel shows an `.lfsconfig` block. Commit it to the root of your repo so every clone points its LFS traffic at Swarmfile:

```
[lfs]
  url = https://hub.swarmfile.com/orgs/<org>/lfs/<repository>
```

The URL encodes your org and repository, so there's no coupling to your git remote — the git remote stays exactly where it is.

Keep it to just the `url`: `git-lfs` only accepts a small safe allowlist of keys in a repo-committed `.lfsconfig` and prints a `warning: These unsafe '.lfsconfig' keys were ignored` line for anything else — per-machine settings (the large-download retry window, credentials) go in `git config` in the next step instead.

3. Store the credential

`git-lfs` authenticates over **HTTP Basic**. Run the setup once per clone: register git-lfs's filters (without this, `git add` commits the raw bytes instead of an LFS pointer), store the `sf_key_…` in your git credential store as the **password** (the username can be anything), tell git-lfs to use basic auth directly so it skips a 401-probe round trip on every call, and widen the large-download retry window (see Large files):

```
git lfs install

# Store the credential (username arbitrary, password = your sf_key_…)
git credential approve <<EOF
protocol=https
host=hub.swarmfile.com
username=swarmfile
password=sf_key_…
EOF

# Use basic auth for this endpoint without the 401 probe
git config lfs.https://hub.swarmfile.com/orgs/<org>/lfs/<repository>.access basic

# Per-machine: how many times a large download retries while it's prepared.
# (.lfsconfig can't carry this — git-lfs ignores non-allowlisted keys there.)
git config lfs.transfer.maxretries 240

# Required on a managed repository: wire git-lfs to the Swarmfile transfer
# agent (bundled with the Desktop App; otherwise install it once — see
# "Pushing large files"). Optional on an unencrypted repository.
swarmfile-lfs install
```

The "Generate git-LFS credential" panel emits this snippet with your real org, repository, and key filled in — copy it from there; it has a **Windows (PowerShell)** tab for machines where bash is not available. A `~/.netrc` entry (`machine hub.swarmfile.com login swarmfile password sf_key_…`) works too if you prefer.

On Windows, run the same setup from PowerShell instead — the credential step uses a here-string, since `<<EOF` is bash-only:

```
git lfs install

@'
protocol=https
host=hub.swarmfile.com
username=swarmfile
password=sf_key_…
'@ | git credential approve

git config lfs.https://hub.swarmfile.com/orgs/<org>/lfs/<repository>.access basic
git config lfs.transfer.maxretries 240

# Required on a managed repository (the Desktop App bundles it; otherwise
# install it once — see "Pushing large files").
swarmfile-lfs install
```

4. Track and push

From here it's the usual git-LFS commands; on a managed repository the transfer agent you installed handles the upload:

```
git lfs track "*.psd" "*.exr" "*.uasset"   # writes .gitattributes
git add .gitattributes
git add my-big-file.psd
git commit -m "Add art source"
git push           # source objects to your git host
git lfs push       # big files to Swarmfile
```

Teammates who clone the repo get the committed `.lfsconfig` , add their own credential once (step 3), and `git lfs pull` fetches the big files from Swarmfile.

Figure omitted in print (animation): git-LFS pointed at Swarmfile through the helper and the swarmfile-lfs agent: a 25 MB level file goes in as a pointer, and a second clone returns it byte-identical — the two SHA-256 lines are the evidence. - https://swarmfile.com/demo/oss-git-lfs.cast

Or let the GitHub App do it

Everything above works with any git host and any git-lfs client. If your remote is GitHub, the repository's git-LFS tab can do part of it for you (the git-LFS tab installs the Swarmfile GitHub App and opens the setup pull request; the manual steps above always work). If you belong to more than one organization, the install page asks which one to link the installation to — one installation maps to one Swarmfile organization:

- **Install the GitHub App** — click the button in the tab, pick the repositories to let Swarmfile see, and come back. One installation per Swarmfile organization.
- **Route a repository** — in the same tab, choose the repository and click **Route this repo to** `<repository>` . That links it to the repository (each repo gets its own `.lfsconfig` , because the URL embeds the repository id).
- **Open setup PR** — Swarmfile opens a small pull request that adds `.lfsconfig` to the repository. Merge it (or close it and use the manual steps above — nothing is forced).
- **Mint git-LFS credential** — the tab mints a repository-scoped `sf_key_…` for you and shows the same per-clone commands as step 3, prefilled.

- **Filter or bulk-link** — an installation with many repositories gets a filter box and a **Link all shown** action, so you don't link them one at a time.
- **Re-run reflect** — if a push's automatic reflect misses (a very large push, or an object that was still being verified), the repository row's **Re-run reflect** runs it again from the branch head.
- **Mirror releases** — the repository row's checkbox mirrors this repository's future ready releases (tag, notes, attached assets) to the GitHub repo as GitHub Releases. Off by default, one-way, idempotent; unpublishing a release here deletes its mirrored GitHub release. A CI-driven team can instead paste a workflow that calls the mirror endpoint with a repository API key on `release: published` — the same idempotent mirror, so the two never conflict:

  ```
  name: Mirror releases to GitHub
  on:
    release:
      types: [published]
  jobs:
    mirror:
      runs-on: ubuntu-latest
      steps:
        - name: Ask Swarmfile to mirror this release
          run: |
            curl -fsS -X POST \
              -H "Authorization: Bearer ${{ secrets.SWARMFILE_API_KEY }}" \
              "https://hub.swarmfile.com/orgs/${{ vars.SWARMFILE_ORG_ID }}/repositories/${{ vars.SWARMFILE_REPOSITORY_ID }}/releases/${{
  github.event.release.tag_name }}/mirror" \
              && echo mirrored
  ```

  The endpoint accepts the release **tag** (as above) or its release id. The API key must belong to an organization owner or admin — the mirror route is owner/admin-gated. (The scheduled sweep is the belt-and-braces: a missing or failing workflow cannot leave the mirror behind.)

After that, `git lfs push` / `pull` work exactly as described above, and **agent-pushed objects show up on the mounted drive automatically** — a push webhook reflects the pushed branch's objects onto the repository's matching Swarmfile branch, so you don't need to run `swarmfile-lfs reflect` by hand.

On the machine that pushed, the file appears on the drive within moments as an optimistic preview; the hub's reflect then confirms it, or withdraws the preview if it refuses the mapping (a path conflict, or a folder you can't write to). Teammates see the file once the hub reflect lands. (Objects pushed by a *stock* `git-lfs` client are pull-able but not reflected — see the note under Deduplication and the mounted drive.)

The App is provisioning-only: it never sees or proxies your LFS objects, and the credential stays the `sf_key_…` you minted — uninstalling the App doesn't break clones that already have their credential.

Deduplication and the mounted drive

Swarmfile stores content once. If two people push the same bytes — or the same object already exists — the second upload is skipped, exactly as LFS dedup is meant to work. One exception: when the bytes so far exist only as a file saved on the drive, the first push uploads them anyway, and Swarmfile checks the drive copy in the background; once it's confirmed, later pushes of the same bytes are skipped.

It goes one step further: a file saved to the **mounted Swarmfile drive** through the desktop app is `git lfs pull`-able by its object id, with **zero duplicate storage**. (So is a file added through the web dashboard's uploader. A file saved before object ids were recorded can be given one with `swarmfile git backfill-oids`.) The LFS object *is* the native content — the same bytes the drive already holds, addressed two ways. So a file your team produced on the drive can be pulled by a git-LFS client that only knows its `oid`, without storing it twice.

That holds for any version the repository still keeps: a drive version that has aged out of your history retention is gone for LFS too, and a pull of it reports the object as missing. On the default **Keep every saved version** repositories a drive version doesn't age out, so its LFS object stays pull-able while the version is retained.

Pushing or re-pushing an object with `git lfs push` makes it an LFS object in its own right, kept independently of drive retention.

It doesn't apply to end-to-end encrypted repositories (no LFS at all), nor to content encrypted with a static key you supply yourself (`SWARMFILE_ENCRYPTION_KEY`): in both cases the hub holds no key, so Swarmfile never sends it a file's SHA-256 object id.

The reverse direction works too, with one manual step. An object pushed with `git lfs push` can be **projected onto the mounted drive** as a first-class file at its repo path — reusing the blocks the push already stored (again, zero duplicate storage). Because the LFS protocol only ever sends an object's *id*, never its path, the file's path comes from your git working tree, so you run one command after pushing:

```
git lfs push origin main
swarmfile-lfs reflect           # repositories the just-pushed objects onto the drive
```

`reflect` reads `git lfs ls-files`, sends the oid→path map to Swarmfile, and the hub creates the drive entries. It mirrors your current git branch onto a same-named Swarmfile branch (auto-created; override with `--branch`), and it **never overwrites** a file already on the drive — a path already occupied by different content is reported as a conflict and left alone. Re-running it is safe (already-projected files report as unchanged). (If you run the desktop app rather than the standalone binary, `swarmfile-engine reflect` is the identical command.) To make it automatic, wire it into a `git push` alias:

```
git config --global alias.pushr '!git push "$@" && swarmfile-lfs reflect'
# then: git pushr origin main
```

After reflection, that path has a real drive entry — so it's reachable through *both* surfaces (drive + LFS), participates in mount worksharing locks, and appears for teammates on the drive.

> **Reflection needs the transfer agent.** `reflect` maps an object onto the drive only when Swarmfile stored it as content it can address — i.e. the object was pushed through the transfer agent (above) or the desktop app. On an unencrypted repository, a plain **stock `git-lfs push`** takes the direct-upload path: the

> object is stored and `git lfs pull`-able, but it isn't a drive manifest, so `reflect` reports it as `missing`. (On a managed repository a stock push is refused; use the agent, whose objects reflect normally.) If a file must appear on the mounted drive, install the agent (see Pushing large files) and push through it, or write the file to the drive directly. The GitHub App's automatic reflect has the same dependency.

Reflection is **additive**: it creates or leaves-alone, and never deletes. Removing or renaming a file in git won't remove or move its drive entry — do that on the drive (or via the dashboard). It enforces the same **write** permission a drive write does: on a repository with per-folder access rules, reflecting a file into a folder you can't write to is refused (reported as `denied`) and skipped.

A very large push reflects a **bounded number of objects per run** (a whole-tree scan is capped for cost). The rest land on the next push — or immediately, via another `swarmfile-lfs reflect` (or the tab's **Re-run reflect**). The cap is noted in the row's status when it applies.

Large files

Downloading large files

Small and mid-sized objects download immediately. A **very large** object (roughly a gigabyte or more) is prepared on its **first** download: Swarmfile assembles it into a single downloadable copy in the background, and until that finishes `git lfs pull` waits and retries automatically — you'll see it pause, then complete.

This is why step 3 sets `git config lfs.transfer.maxretries 240`: it gives the client enough patience to wait through preparation. Once prepared, that copy is reused, so subsequent pulls of the same object are immediate.

Under the hood, an attempt while preparation is still running gets `429 Too Many Requests` with a `Retry-After` interval — that pause is git-lfs honoring the retry — and a preparation that fails for good gets `502` naming the reason rather than looping the client on 429.

If a pull of a very large object ever gives up with a "being prepared, retry" message, just run `git lfs pull` again — preparation continues in the background and the retry will pick up the finished copy. A `502` is different: preparation failed permanently and the message says why, so retrying won't change the outcome until the underlying cause clears. (Raising `lfs.transfer.maxretries` further with `git config` widens the wait window if you routinely pull multi-gigabyte objects. It has to be `git config`, not `.lfsconfig` — git-lfs ignores that key in a committed config file.)

Pushing large files

On an **unencrypted** repository a plain `git lfs push` sends each object in a single request, supporting objects up to about 5 GiB. On a **managed** repository the transfer agent is required for every upload; it also lifts the size boundary, sending chunked, resumable blocks (up to 256 GiB per object).

Install the **Swarmfile transfer agent**: a small (~2 MB), standalone, open-source binary (`swarmfile-lfs`) that speaks git-lfs's documented custom-transfer protocol. Its FastCDC chunker and the block/manifest wire format are open source too, published with the agent at swarmfile-lfs.

No desktop app required — but if you already run the desktop app, it's already installed (every desktop installer bundles it on `PATH`), so skip the download and go straight to `swarmfile-lfs install`.

```
# 1. If the desktop app is installed, the agent is already on PATH — skip to
#    step 2. Standalone install:
cargo install swarmfile-lfs-transfer    # any Rust toolchain; installs the swarmfile-lfs binary
curl -fsSL https://get.swarmfile.com/lfs | sh             # macOS / Linux
#    Windows (PowerShell):  irm https://get.swarmfile.com/lfs.ps1 | iex
# …or download the binary for your platform from the public release:
#    https://swarmfile.com/public/swarmfile/swarmfile-lfs
#    (No Homebrew formula; use cargo or the installer above.)

# 2. Wire git-lfs to it, once per machine:
swarmfile-lfs install
```

> Those standalone channels ship separately from the app, so the dashboard's install command is the authoritative one for your account. If you already run the desktop app, `swarmfile-lfs` is installed on `PATH` with it, and the app also bundles an equivalent built-in agent (`swarmfile-engine lfs-transfer`; on a managed repository its large-object fast path is unavailable, so it falls back to the same encrypted block upload).

`swarmfile-lfs install` just writes the git-lfs custom-transfer adapter config for you (add `--local` to scope it to the current repo instead of your global config; `swarmfile-lfs uninstall` undoes it in the same scope).

With it in place, `git lfs push` transparently uploads large files as small content-addressed blocks — encrypted at rest on a managed repository (the built-in agent seals blocks client-side; the standalone agent's blocks are encrypted by the hub on write) — up to **256 GiB per object** (and your repository quota), deduped against the same content on your Swarmfile drive.

Nothing else changes: `git lfs push` / `pull` work unchanged, the agent only kicks in for uploads, and it authenticates with the same credential you stored in step 3 (so make sure that's done on each machine that pushes large files).

On an unencrypted repository, stock `git lfs push` still works up to about 5 GiB per object without the agent. On a managed repository the agent is required — a stock push is refused with `lfs_agent_required` — and any file is always available by writing it to the mounted drive instead.

If you already run the **Swarmfile desktop app**, `swarmfile-lfs` is already on `PATH` — `swarmfile-lfs install` is all you need.

The engine also exposes an equivalent built-in agent as `swarmfile-engine lfs-transfer`, if you'd rather point git-lfs at that (`git config lfs.customtransfer.swarmfile-chunked.path swarmfile-engine` / `.args lfs-transfer`). (Both agents encrypt at rest on a managed repository; they differ only in where the encryption happens — client-side in the built-in agent, on the hub for the standalone agent's blocks.)

An interrupted large push is safe to re-run. Just run `git lfs push` again: content-addressed blocks the agent already uploaded are deduplicated, and the desktop app's built-in agent additionally resumes a ≥ 64 MiB multipart upload from the parts that already landed instead of starting over. Nothing is left behind either way — an upload nobody resumes is reaped server-side after 7 days on Swarmfile-managed storage.

What's supported

- The LFS **Batch API** (`upload` and `download`) and the **basic transfer** adapter — the operations `git lfs push`, `pull`, `checkout`, and `fetch` use.
- **Verify**, so a truncated upload is caught rather than silently stored.
- **Locking** (`git lfs lock` / `git lfs locks` / `git lfs unlock`), for every tracked path — including files that live **only** in your git host and have never been written to the Swarmfile drive. When a path *does* also exist on the drive, its LFS lock and its desktop/mount worksharing lock are the **same** server-side lock, so locking via `git-lfs` and locking on the drive mutually exclude, across machines and across both surfaces. The path resolves on the branch `git-lfs` reports (its `ref` — the checked-out branch) whenever Swarmfile has that branch; otherwise it falls back to the repository's default branch, so an LFS-only repo whose git feature branches have no Swarmfile counterpart keeps working as before. A path the resolved branch's tree doesn't have — a file that exists only in git, or one missing from that branch — takes the branch-less advisory lock below. Lock listing and push verification scope entry locks the same way — a drive lock taken on a feature branch doesn't block a push to `main` — while the branch-less advisory lock applies to every branch. The advisory lock is what other `git-lfs` clients see; unlocking is owner-gated (use `--force` to break another user's lock). Locks hold for 30 days and nothing renews a git-side lock, so a lapsed one releases itself rather than staying stuck — take it again if the work outlives that. See Working with Files.

Performance

Upload throughput is bounded by *your* connection to Swarmfile, so no headline number is meaningful. Two properties hold regardless of uplink:

- **The chunked agent is faster for large objects**, because it sends many ~1 MiB blocks in parallel instead of one large request.
- **Re-pushing an unchanged object transfers nothing** — git-lfs sees the server already has the object and skips it, so a re-push costs one batch round-trip (well under a second) no matter the file's size.

Continuous integration (CI)

Two things make CI painless, and they split by direction:

- **Pulling assets needs no agent for almost everything.** `git lfs pull` / `fetch` / `checkout` works with the stock git-lfs that's already on every runner; large objects materialize and retry (that's what the `lfs.transfer.maxretries` setting below is for). One boundary: a very large object's first *stock* download may be paced — the hub answers with a "wait, preparing" and the retry setting carries the client through. Uploads are the direction that needs the agent — on a managed repository every upload does, and for an unencrypted repository it lifts the ~5 GiB boundary. A build or test job needs only credentials.
- **Pushing objects needs the agent on a managed repository**, and needs it for objects over ~5 GiB on an unencrypted one. A release or asset-generating job adds one install step.

In both cases, store the repository's `sf_key_…` as a CI secret (`SWARMFILE_API_KEY`) and feed it to git through a credential helper. Never bake the key into `.lfsconfig` or a command line.

GitHub Actions

Pull-only job (build/test — no agent):

```
env:
  SWARMFILE_API_KEY: ${{ secrets.SWARMFILE_API_KEY }}
steps:
  - uses: actions/checkout@v4
    with: { lfs: false }            # skip the auto-smudge; we pull explicitly below
  - name: Configure Swarmfile LFS credentials
    run: |
      git config --global credential.helper \
        '!f() { test "$1" = get && printf "username=lfs\npassword=%s\n" "$SWARMFILE_API_KEY"; }; f'
      git config --global lfs.transfer.maxretries 240   # wait through large-object preparation
  - run: git lfs pull               # any size the hub can assemble; a very large object may ask the client to wait while it prepares
```

Push job (release/asset generation — installs the agent). If you can't use the `get.swarmfile.com` installer (a restricted network, or a policy against piping a download to a shell), download `swarmfile-lfs` from the downloads page or copy it off a machine with the desktop app, and substitute that path for the `curl` step:

```
env:
  SWARMFILE_API_KEY: ${{ secrets.SWARMFILE_API_KEY }}
  SWARMFILE_LFS_URL: https://hub.swarmfile.com/orgs/<ORG>/lfs/<REPOSITORY>
steps:
  - uses: actions/checkout@v4
  - name: Install the Swarmfile transfer agent
    run: |
      curl -fsSL https://get.swarmfile.com/lfs | SWARMFILE_LFS_BINDIR="$HOME/.local/bin" sh
      echo "$HOME/.local/bin" >> "$GITHUB_PATH"
      swarmfile-lfs install         # writes the git-lfs custom-transfer config (global)
  - name: Configure Swarmfile LFS credentials
    run: |
      git lfs install                # register the clean/smudge filters on this runner
      git config --global credential.helper \
        '!f() { test "$1" = get && printf "username=lfs\npassword=%s\n" "$SWARMFILE_API_KEY"; }; f'
      git config --global "lfs.${SWARMFILE_LFS_URL}.access" basic
      git config --global lfs.transfer.maxretries 240
  - run: git lfs push origin HEAD   # all objects go through the agent (required on managed)
```

GitLab CI

```
push-assets:
  image: ubuntu:22.04
  variables:
    SWARMFILE_LFS_URL: https://hub.swarmfile.com/orgs/<ORG>/lfs/<REPOSITORY>
  before_script:
    - apt-get update && apt-get install -y git git-lfs curl ca-certificates && git lfs install
    - curl -fsSL https://get.swarmfile.com/lfs | SWARMFILE_LFS_BINDIR=/usr/local/bin sh
    - swarmfile-lfs install
    - git config --global credential.helper
        '!f() { test "$1" = get && printf "username=lfs\npassword=%s\n" "$SWARMFILE_API_KEY"; }; f'
    - git config --global "lfs.${SWARMFILE_LFS_URL}.access" basic
    - git config --global lfs.transfer.maxretries 240
  script:
    - git lfs push origin HEAD
  # Set SWARMFILE_API_KEY as a *masked, protected* CI/CD variable — not in this file.
```

The credential helper reads `SWARMFILE_API_KEY` at the moment git invokes it, so the secret never lands on a command line or in the log. For a pull-only job you can drop the agent install and the `access basic` line entirely.

Leaving Swarmfile

Your source lives in your own git host and never touches Swarmfile — so leaving is only ever about pulling your binary objects back out, and that works with **stock git-lfs and no Swarmfile software installed at all**. There is no export API to learn and no proprietary format to unwind.

```
# In a clone that points at Swarmfile, pull every version of every object
# referenced by any branch or tag into your local .git/lfs/objects store:
git lfs fetch --all

# Confirm you hold complete, uncorrupted copies (each object hashes to its oid):
git lfs fsck
```

After `git lfs fetch --all`, every object is on your disk. To move them to another LFS server, point the repo at the new endpoint and push:

```
git config -f .lfsconfig lfs.url https://your-new-lfs-server/…   # GitHub, GitLab, S3-backed, self-hosted, anything
git lfs push --all <remote>
```

This is deliberately the *standard* git-lfs export path — the custom-transfer agent is only an optional accelerator for large **uploads**; nothing about your objects is locked to Swarmfile software. You can walk away with a single command, which is exactly the point: your binaries are as portable as your git history.

Troubleshooting

A `warning: These unsafe '.lfsconfig' keys were ignored` line. Something other than `lfs.url` is in the committed `.lfsconfig` (often `lfs.transfer.maxretries`). git-lfs only accepts a safe allowlist in a repo-committed config file and ignores the rest by design; move per-machine settings to `git config` (step 3 shows the retry one). The warning is about the ignored key, not a failed transfer.

A `401` on every LFS call. Credential helpers disagree about whether the key belongs in the username or the password field; Swarmfile accepts the `sf_key_…` in **either**, but the safest placement is the password. Re-run the `git credential approve` step above, and confirm you set `…access basic` so git-lfs sends the header on the first request.

A `422` for every object (`lfs_agent_required`). The repository is managed/encrypted, and the push is using the stock direct path. Run `swarmfile-lfs install` (the Desktop App bundles the agent), then retry. On an unencrypted repository this error does not apply.

A `422` for every object (not `lfs_agent_required`). The repository is end-to-end encrypted, which git-LFS can't support (see above). Use a managed or unencrypted repository, or move the source-plus-assets to a mounted drive instead.

`git lfs pull` fetches nothing / objects are missing. Confirm `.lfsconfig` is committed and its `url` names the right org and repository, and that the credential you stored is for that same repository.

`git lfs unlock` fails on a lock someone else holds. Path locks are owner-gated: only the user who took a lock can release it normally. Pass `--force` (or `--id <id> --force`) to break another user's lock, or ask them to unlock it. `git lfs locks` shows the current holder of every live lock.

Coding agents (mounted workspaces)

A Swarmfile repository mounts as a real filesystem path. That makes it a good workspace for a coding agent: the agent reads and writes ordinary files, and every change is versioned, lockable, and shared — no `git clone`, no `git worktree` per task, and no full re-install of dependencies for every parallel checkout.

The shape this guide builds:

- one **engine process** per machine (or per agent when you want process isolation — see below),
- one **repository API key** (optionally branch-scoped, or path-scoped when the agent should not touch git) so the engine runs headless (no browser sign-in),
- one **mount per agent**, each with a label and (optionally) its own branch,
- one **shared scratch cache** per repository per machine, so a second mount reuses the first one's downloaded packages,
- an optional **installed dependency tree**, so a new branch or a second workspace starts without reinstalling (see §3),
- and the **durable create queue**, which is what makes a burst of file creations from an agent safe.

The same primitives appear in the Desktop App — this guide is the scripted, headless form.

What each agent gets, and what it must not be given

Give an agent:

- **The mount path.** It's a normal POSIX/Windows path (`~/agents/agent-1/…`, `X:\…`). Everything the agent does happens there; there is no separate "sync" step.
- **The scratch dir** (below) as the place for tool caches.
- **Exit codes** (below) so its automation can tell "you need to do something" (2) from "something failed" (1).
- **One `SWARMFILE_*` environment set** per engine — never two engines sharing a cache directory.

Do **not** hand an agent:

- A second mount point that points at the *same directory* another agent uses — mount points are exclusive to one engine process.
- A personal refresh token copied off a workstation. Use a repository API key: it's revocable, it's walled off to one repository, it can be branch- or path-scoped (see §7.1), and it can't act as the human who minted it.

Platform matrix: how many mounts per engine

Platform	Several mounts in one engine?	Recommended shape for N agents
Windows (WinFsp)	Yes — `mounts open` takes a drive letter (`Y:`, `Z:`, …), `auto` for the next free one, or an **empty** folder (a folder with anything in it is refused)	One engine, N mounts, one API key
Linux (FUSE)	Yes — `mounts open` takes directory paths (same engine path as Windows; macOS/Linux use a folder, not a drive letter)	One engine, N mounts, one API key
macOS (FUSE-T)	Yes — `mounts open` takes directory paths; FUSE-T provides 18 mount-helper slots per Mac, shared with every FUSE-T app (not a macOS mount limit — see below)	One engine, N mounts, one API key

An engine can serve every agent's mount: auth is one API key, and the shared scratch cache (below) is per machine, so the dependency cache is still paid for once. Running one engine per agent remains a valid isolation choice (a wedged mount helper then can't affect siblings), but it isn't required on macOS. FUSE-T provides 18 mount-helper slots per Mac, shared by every FUSE-T app on it.

An `open` at capacity fails with `fuse_t_helper_ports_exhausted`: close a drive or quit the app holding a slot, then retry; `swarmfile doctor` names the holders. A boot mount that never attached can retry in place with `swarmfile mounts retry`; one that attached earlier needs a restart.

1. Headless engine with a repository API key

The commands in this guide use the `swarmfile` CLI, which ships with every platform installer (Linux `.deb`, the Windows MSI, the macOS `.pkg`). A headless box needs only that package — no GUI sign-in — and the CLI resolves the running engine's control socket on its own. See CLI: swarmfile for the per-platform install locations.

Mint a key as an org admin or owner, from any running engine:

```
swarmfile api-key create agent-node-1                # this mount's repository
swarmfile api-key create agent-node-2 --repository proj_… # or an explicit repository id
```

`--repository` takes a **repository id**, not a slug (a slug is refused with `repository_not_found`, and the error names the id to use). The secret is printed **once** (`sf_key_…`). List and revoke keys with `swarmfile api-key list` and `swarmfile api-key revoke <id>`. A key is walled off to exactly one repository, so a compromised agent node can't reach anything else in the org.

Keys count against your plan's **headless-seat allowance** (Free 1, Starter 2 per seat with a minimum of 5, Pro 5 per seat with a minimum of 10, Studio 10 per seat with a minimum of 20), and an additional headless seat is $2/month on any paid plan (one repository-scoped key each). One key per machine/fleet is the recommended pattern (revocation radius and its own rate-limit bucket); a `402` with `code: keys_cap` means the org is at its cap, so revoke an unused key or add a seat under Settings → Billing. Agents running through a workstation's existing mount need no key at all.

To give each agent its own engine (process isolation; otherwise one engine can host every agent's mount, as the matrix above recommends), point each at its own cache dir and mount point:

```
export SWARMFILE_API_KEY=sf_key_…
export SWARMFILE_ORG_ID=org_…
export SWARMFILE_REPOSITORY_ID=proj_…
export SWARMFILE_CACHE_DIR="$HOME/.swarmfile/agent-1"   # one per engine
export SWARMFILE_MOUNT_POINT="$HOME/agents/agent-1"     # one per engine
export SWARMFILE_MACHINE_ID=agent-node-1
swarmfile-engine
```

With `SWARMFILE_API_KEY` set, the engine skips sign-in entirely — no browser, no refresh token. (If both an API key and a refresh token are set, the key wins and the engine says so in its log.) See the `api-key` CLI reference for the full flags, and Deployment Topologies for putting these on dedicated boxes.

2. One mount per agent (and targeting it)

The engine's first mount is the **boot mount**, id `default`. Additional mounts are full, symmetric mounts — each with its own changelists, metadata, and lock manager:

```
swarmfile mounts list
swarmfile mounts open Y: --label agent-2 --branch feature-x
swarmfile mounts open Z: --label agent-3 --repository proj_other
swarmfile mounts status agent-2-id
swarmfile mounts current agent-2-id
```

> The examples use Windows drive letters; on macOS and Linux the mount point is a directory path instead (e.g. `swarmfile mounts open /Volumes/agent-2 --label agent-2`). Everything else — labels, ids, `--branch`, `--repository`, `status`, `current` — is the same on every platform.

- `--conflict-mode hard` makes the mount tell an agent the truth about a lost write: when a save would overwrite a version another writer changed first, the saving `fsync` (and on Linux the close-time `flush`; on Windows `FlushFileBuffers`) waits for the server and fails with a stale-version error (`ESTALE`; Windows: the file-was-externally-altered status) instead of accepting the save locally and surfacing the conflict later.

  Agents should `fsync` before treating a save as committed — on macOS a `close()` cannot carry the error (the NFS client would retry it forever).

  Without hard mode (neither this flag nor a repository's `conflictMode` = hard in repository settings), a save that lost the race parks quietly — fine for a human at the tray, wrong for a loop that will never look.

  Agents should treat the error as "re-read the file, re-apply your change, retry"; `EAGAIN` from the same call means the server is just slow (the save stays queued — fsync again later).

  A checked-in `.swarmfile/conflict-policy` can mark regenerable paths (lockfiles, build output, generated code) as newest-wins. A stale save to a matching path is resolved in the newest writer's favor through the hub's conflict record instead of parking read-only; the replaced version stays on the conflict record.

  It is opt-in per path, gitignore syntax with `!` exceptions, and `swarmfile mounts status <id>` reports how many rules the mount found. Unmatched paths keep the behavior above. A fresh write against a held editing lock is still refused; a stale replay that already conflicted is not re-checked against the lock at resolve time today.

Figure omitted in print (animation): Two agents work one branch from their own mounts: both read index.html, the first save lands, and the second agent's save — made against the version it read, over a range that does not even overlap — is refused. The agent takes the teammate's version, re-reads, re-applies, and both writes land. - https://swarmfile.com/demo/oss-agents.cast

- `--label` is for humans; the id (a long hex string, or `default` for the boot mount) is what scripts use — take it from `mounts list --json`.
- `--branch <name>` pins that mount to a branch. It opens (or reuses) a divergent `(repository, branch)` scope, so the mount works that branch while every other mount stays on its own — and can be switched later with `mounts branch`. A unique directory root per agent (e.g. `/agents/<name>/…`) is still good hygiene, but not required: the engine creates missing ancestor directories on a directory create, so a fresh branch is safe to write into anywhere.
- `--repository <id>` opens a mount on a different repository in the same org.
- `--sparse <glob>` (repeatable) opens a **sparse** mount: only the matching paths are visible (and writable) on that drive, so an agent working one package never sees the rest of the repo. `--exclude <glob>` is sugar for the `!` negations ("everything except"); an excluded path has no listing entry, refuses an open and a write-create, and the mount's offline (`hydrate`) set matches. Scope is per mount, so branches mount differently.
- `--exclusive` opts that mount into **exclusive-write mode**: while it has a file open for writing, another mount opened with the same flag on the same repository+branch has its write-open refused (the app sees `EAGAIN`, or "file in use" on Windows) — a rename, delete, or atomic-save replace of that file is refused too — instead of both editing the file and the later save becoming a conflict to resolve.

  Off by default, fixed at open, and shown as a lock badge in the drive list. Open **every** agent mount with it when several agents share one tree, and pair it with `--conflict-mode hard`: two mounts of one engine are a single identity to the hub, so ordinary locks do not separate them.
- Every command that isn't under `mounts` targets the current mount unless you pass the global flag:

```
swarmfile --mount agent-2-id status
swarmfile --mount agent-2-id commit -m "agent 2 results"
```

Close a mount with `swarmfile mounts close <id>`. It refuses while a changelist is open (`--force` abandons it) and refuses the boot mount outright (`exit 2`, `is_default_mount`) — closing that one means shutting down the engine.

3. Shared dependency cache

Every mount of a repository on a machine can share one **scratch directory** for tool caches (a pnpm store, a cargo registry, a pip wheel cache). Declare what `run` should redirect, in a committed file at the repository root:

```
# .swarmfile/cache.yml
caches:
  - name: pnpm
    env: PNPM_HOME
    subdir: pnpm
  - name: cargo-registry
    env: CARGO_HOME
    subdir: cargo
```

`swarmfile scratch init` writes a commented starter file. Then run installs and builds through `run`, so the tool sees the shared cache:

```
swarmfile mounts scratch-dir default     # the shared dir + declared redirects
swarmfile run -- pnpm install            # PNPM_HOME -> shared scratch
swarmfile run -- cargo build --release   # CARGO_HOME -> shared scratch
```

`run` passes stdio and the exit code through, and it never refuses to run your command over a cache problem — with no config, a bad entry, or an unreachable engine it warns and runs the command unmodified.

Maintenance:

```
swarmfile scratch list
swarmfile scratch gc --older-than-days 14 --dry-run
swarmfile scratch clear <name> --force
```

One scratch dir per repository per machine, shared regardless of branch. Only redirect **read-mostly, content-addressed caches** — not build output directories, which would contend between concurrent mounts. Nothing in a scratch cache is unrecoverable repository data; clearing one only costs a slower next build. Full reference: `scratch` and `run`.

Installed dependency trees

The scratch cache removes the downloads, not the extract: a fresh mount still runs the installer over its packages (about 22 s for 69 packages). Two opt-in forms keep the installed tree itself:

- **Hub-backed, for agents on different machines.** `swarmfile workspace warm --share-deps -- npm install` installs inside `.swarmfile/deps/<OS-arch-Node-ABI>-<libc>-<lockhash>/tree`, stages the packages copy as `tree/.packages.partial`, renames it to `tree/packages` with a completion marker, and leaves a relative `node_modules` symlink. A branch forked with `swarmfile mounts new --from <warming-branch>` inherits it.

  Measured: **126 s** one-time install+copy, then a 1,246-file fork ran `require("express")` with no install, including from a fresh engine cache.

  It is ordinary repository content — synced and storage-billed; `.npmrc` credentials are never copied; pnpm's nested layout and a WinFsp mount are not supported.

- **Machine-local, for agents on one machine.** `swarmfile workspace warm --local -- npm install` installs in the engine cache and imports the tree into the mount as local-only metadata/blocks (no hub row, no absolute link). A new workspace on the same machine attaches it with no install — measured **14.6 s** first install+import (2 s npm, warm cache) and **12.6 s** for the second workspace.

  It is Unix-only, must run from the mount root, and refuses workspace/monorepo installs; the import is capped at 20,000 entries and 512 MiB per file. Executable bits are preserved (so `.bin` tools run), a strict E2E/static mount is refused (the layer is plaintext), and a failed import is rolled back rather than left as a truncated tree.

  `workspace deps gc` lists/removes old layers; `workspace deps materialize <key> <checkout>` copies one into a plain (unmounted) checkout on the same machine.

Both forms key on the repository's manifests, and `--local` also keys on the install command; a cached `--local` hit does not run the command (the CLI says so, and `--json` reports `commandRan: false`). Remove the old `node_modules` once after a lockfile change. Old layers are manual cleanup with the repository's `swarmfile workspace deps gc --kind local` (a live layer is kept by its repository marker; `--keep` / `--force` handle the exceptions). Full flags: `workspace warm`; the limits are in Known Limitations.

4. Create bursts and the pending-create queue

File creation returns immediately on the drive. On the mounted drive, a create (or `mkdir`, or a symlink on macOS/Linux — Windows refuses symlink creates) returns at once under a locally minted id; a background worker confirms it with the hub (8 in flight by default) and peers on the LAN see the name by gossip before the hub does.

Creates that are due together go to the hub together, up to 50 in one request, and a new file's first save travels with its create instead of in requests of its own. This is what makes an agent's scaffolding burst safe and fast: on Pro's default limits, 5,000 new files reach the hub in under a minute (lower plans drain the same burst more slowly, and nothing is lost), and a create made while the hub is unreachable simply queues.

What to check, and what an agent will see:

```
swarmfile status                # "new files: N waiting to reach the hub (oldest …)" while queued
swarmfile --json status | jq .pendingCreates
```

pendingCreates field	Meaning
queued	Creates this machine made that the hub hasn't confirmed. 0 = drained.
cancelled	Deleted locally before the hub heard of them; no hub call needed.
stuck	Queued and failed 6+ times — hub refusing or unreachable for these.
oldestAgeSecs	Age of the oldest queued create. Growing with queued unchanged means nothing is draining.
peerUnconfirmed	Entries peers gossiped as created that this machine hasn't seen confirmed.
items	The oldest 20 queued creates (path, type, attempts, age).

Rules of the road while something is pending:

- **A name collision on the hub renames, it doesn't overwrite.** If the hub already has a different file at that path when the queued create lands, your copy is kept as `name (conflict-xxxxxxxx).ext` and the error ring says so.

  The exception is a name held only by this machine's own queued delete or move of the old file — an editor's safe save, which moves the original aside and deletes it — where the create waits (up to two minutes) for that to reach the hub and then keeps its name; it shows as queued meanwhile. A **folder** with the same name is merged into the hub's folder (like `mkdir -p`) — files you created inside it land there, and the hub's existing contents appear after a few seconds.

- **Operations on a just-created file wait, then answer `409`.** Locking, ACL reads, share-link creation and comment writes on a pending file wait up to 8 seconds for the create to confirm and then proceed. A change (a lock, share, ACL or comment write) sends the file's create at once, even while its first save is still uploading, so it doesn't wait behind that save. If the hub can't be reached in that time they answer `409` with `code: "pending_create"` — retry shortly. For an entry *another* machine created and the hub hasn't confirmed, the refusal is immediate.

- **A create the hub refuses for good is retracted** from peers and reported once in the error ring (`create failed for "…"`). An empty file, a symlink, or a folder is removed. A file you had already written content into is **kept on this computer only** (never uploaded) so the work isn't lost — the message says so. Content in files under a refused folder is discarded, and the count is in the message.

- **Deleting or renaming a just-created file** is passed to other machines immediately. A file a peer created that never reaches the hub is removed after 6 hours with a message.

- **A new file can reach the hub up to a minute after it appears locally.** While a new file's first save is still uploading, its create waits for it (up to 60 seconds) so the two go to the hub as one request. Other machines on the LAN see the file at once by gossip, and the web dashboard's Files view shows "N files arriving from …" until the hub has them.

Tunables, rarely needed:

Variable	Default	Effect
SWARMFILE_CREATE_DISPATCH_CONCURRENCY	8 (1–64)	Creates sent to the hub at once when they go one by one. Lower it if the hub keeps answering 429; raise it on an unthrottled hub.
SWARMFILE_PENDING_CREATE_WAIT_SECS	8 (0–120)	How long lock/ACL/share/comment writes wait for a pending create before 409 pending_create. 0 refuses at once.
SWARMFILE_CREATE_BATCH	on	Send due creates in batches of up to 50 per request. 0 sends each on its own.
SWARMFILE_CREATE_COALESCE	on	Send a new file's first save with its create (one request). 0 sends the create, then locks and commits separately.
SWARMFILE_CREATE_HOLD_UPLOADING_SECS	60 (0–600)	How long a create waits for its first save while that save is uploading. The upload finishing sends it at once, so this only caps a stall.
SWARMFILE_CREATE_HOLD_SECS	2 (0–10)	How long a create waits for a file that is open for writing but not saved yet. 0 turns off all waiting.
SWARMFILE_ARRIVING_REPORTS	on	Tell the repository how many new files are on their way (at most once every 30 s), for the dashboard's "files arriving" note. 0 / false / off turns it off.

These are client-side. The hub's own ceilings are per-org and per-user (plan defaults, owner/admin configurable). Each active API key gets its own window; a per-key limit (`swarmfile api-key rate-limit`) changes that key's bucket.

The shared org window is the larger of the plan's per-org default and active keys × the per-key default, capped by the plan org max and by the plan's key allowance (seats plus packs): give each machine or fleet its own key to add a bucket and scale the org window, not one shared key with a high limit. `swarmfile admin rate-limits usage` reports this minute's usage against them; see Organizations, Repositories & Members.

How fast can one user create files?

The per-user ceiling is a number of requests per minute, and it depends on the plan. Defaults:

Plan	Per-user default	Owner/admin can raise it to	Per-org default
Free	100 / min	fixed	300 / min
Starter	300 / min	500 / min	1,000 / min
Pro	500 / min	1,000 / min	1,000 / min
Enterprise	500 / min	5,000 / min	10,000 / min

The per-org column is the default; active keys scale it to the larger of that default and (active keys × the per-key default), up to the plan org maximum (Enterprise auto-scaling stops at 50,000/min). The plan key allowance caps how many active keys count.

What counts against it is requests, not bytes, and block uploads are not counted at all:

Kind of new file	What it costs
A new file with content (saved when it is created)	one create that carries its first save: **1 request**, or **a tenth of one** when it goes out in a batch
An empty file or folder	**1 request**, or **a tenth of one** in a batch
A file created empty and written later	create, take the write lock, then one commit that also gives the lock back: **3 requests**

Batching is paced to the plan's per-minute allowance: on Pro's default ceiling (500 requests a minute), a 5,000-file burst drains in well under a minute — batched creates cost about a tenth of a unit per file, so a full 50-file batch is 5 units even though it is a single HTTP request. Nothing is lost when a limit is reached; the queue simply drains more slowly.

If a fleet needs more, give it its own key(s): each active key adds its own bucket and one per-key allowance to the org window, up to the plan org max; a per-key ceiling (`swarmfile api-key rate-limit`) only limits that key's share. An owner/admin org override replaces the scaled window (time-boxed with `--for 6h`).

5. Parking and switching

An agent's scope can move without losing queued or staged work:

- **Branch switch (hot).** `swarmfile branch switch <name>` moves the boot mount; `swarmfile mounts branch <id> <branch> [--wait]` moves one independent mount. Both refuse (exit 2) while a changelist is open. Creates queued before the switch land on the branch they were made under — the scope is captured when the create is enqueued, not when it drains.
- **Repository/org switch.** `swarmfile workspace switch --org <id> [--repository <id>]` moves the whole engine. A same-org switch usually takes a hot path; the fallback restarts the engine process (the OS supervisor relaunches it). Queued creates and their corrective renames/deletes stay in the repository captured at enqueue.
- **Park.** `swarmfile changelist park` sets staged work aside so a branch or repository can be left; `swarmfile changelist parked` lists it (age, files) and `swarmfile changelist resume` restores it. Parked work is a *set* per branch, and it's kept alive on the hub, so it survives the switch. `changelist discard --yes` abandons it.
- **Upstream refresh.** A branch is a fork: it does not see later work on `main` . There is no main→branch merge exposed — `swarmfile merge` and `mr` both go *into* the parent — so plan agent branches to be short-lived, or recreate the branch from current `main` when it needs the latest. A merge-down surface is a known gap.
- On a TTY, `workspace switch` prompts before abandoning staged/queued work and offers to park first; `--yes` / `--json` skip the prompt for scripts.

The tray mirrors all of this: the mount switcher lists every mount with its branch and per-row switch — and the tray menu's **Open Another Drive...** opens the same **Open a drive** dialog (its **Prevent another drive from editing a file this one has open** checkbox, under **Advanced**, is the same `--exclusive` opt-in), so a second mount is one menu click away — while a **Parked changes** panel lists, resumes (all at once or one at a time), and discards parked work.

The dialog puts the drive at the next free drive letter on Windows and in a folder next to your main drive on macOS and Linux, unless you choose otherwise under **Advanced**. If a mount can't open, the form says why straight away — for a drive letter that's taken it offers the next free one — and the failed mount never becomes the current one.

6. What to hand an agent (a copy-paste preamble)

```
You are working in a Swarmfile mount, not a git checkout.
- Workspace path: <mount path>          (writes sync automatically; there is no commit-required step)
- Tool caches:    run builds via `swarmfile run -- <cmd>` from the mount
- Status:         `swarmfile status` / `swarmfile --json status` (jq .pendingCreates)
- Exit codes:     0 success, 1 failure, 2 a refusal you should act on,
                  3 a long operation detached (`swarmfile op status` follows it)
- If an operation answers 409 with code "pending_create", the file is still
  syncing: wait a moment and retry. Do not delete and recreate it.
- Don't run two engines against the same cache dir or mount point.
- Session-scoped secrets are rendered into the session environment and are
  redacted from tool output; a value SHORTER THAN 4 CHARACTERS is not redacted
  (use a longer value for anything that must never appear in output).
```

7. One workspace per session (harness hooks)

The manual shape above works, but the harness can do it for you. One command installs Swarmfile's hooks into the harness's own settings — repository-level by default, so the whole team gets it in one committed file. Repository-scope hooks call `swarmfile` from PATH and use the default cache, so every teammate needs the CLI installed; `--user` pins the absolute binary and cache instead.

```
swarmfile hooks install claude-code          # .claude/settings.json + skill
swarmfile hooks install cursor               # .cursor/hooks.json + rule
swarmfile hooks install codex                # .codex/hooks.json + skill
swarmfile hooks install copilot              # .github/agents profile (no hooks)
swarmfile hooks install claude-code --user   # ~/.claude/settings.json
```

At `--user` scope, Cursor's hooks file is written but its rules are UI-managed, so add the Swarmfile rule in Cursor Settings; repository-scope rules are written automatically.

Cursor and Codex expose lifecycle hooks (Codex asks you to review/trust new hooks via `/hooks` , and caps `SessionEnd` at 3 seconds; Cursor does not run session hooks in its cloud agents). Copilot custom agents are prompt profiles, not lifecycle hooks, so that adapter writes the agent profile only; use `swarmfile hooks run --copilot` for a mounted workspace.

Each session then behaves like this:

- **Session start** creates an ephemeral `agent/<label>-<6 hex>` branch and opens a mount for it, waits until the drive is attached, and gives the agent the workspace path as context. Nothing is cloned.
- **After every mutating tool call** (`PostToolUse` for Write, Edit, MultiEdit, NotebookEdit, Bash and PowerShell; Codex also `apply_patch`), the workspace's open changeset is committed as a checkpoint named after the step (`step 12: Bash` , or `step 13: Edit: src/cart.ts`). `Stop` is a turn-end backstop. On an `agent/*` branch the engine also checkpoints after 30 seconds of quiet (`SWARMFILE_AGENT_CHECKPOINT_IDLE_SECS`), so agents without hooks still get per-turn history.
- **Session end** waits briefly for in-flight saves, checkpoints, then closes the mount and archives the branch — its commits stay in history. With `--mr` , it opens a draft merge request instead and leaves the branch for review. **Codex is the exception:** its 3-second SessionEnd cannot run the archive, so it closes the mount and leaves the branch to the TTL sweeper; use `swarmfile hooks run --mr -- codex` for a draft MR.

The hooks call `swarmfile` from PATH; nothing else is added to the agent's environment. The installer also writes the harness's instructions file (`.claude/skills/...` , `.cursor/rules/...` , `.agents/skills/...` for Codex, or the Copilot agent profile) if absent, with the command cheat-sheet: checkpoint, `blame --lines` , restore, conflicts, the shared cache, and the "never clone" rule. An edited skill is left alone. Re-running the install replaces Swarmfile's own hooks and leaves any others in the file alone.

A hook cannot change the agent's working directory, so the safest launch is the wrapper:

```
swarmfile hooks run -- claude
```

It mounts first, starts the command with its cwd **inside** the workspace, and finishes the workspace when the command exits. It prints how long the workspace took to become ready.

`swarmfile workspace warm -- <install or test command>` runs one dependency warm-up through the same shared cache (the first workspace on a machine pays; later ones reuse it). The `hooks run` wrapper exports the `.swarmfile/cache.yml` redirects to the agent process, so its own installs/builds use the shared cache too; the hook path (an agent launched without the wrapper) should use `swarmfile run --` explicitly.

Add the global `--timings` to see elapsed and phase timings on stderr — supported on `commit` , `checkpoint` , `changelist submit` , `mounts new/open` (with `--wait`), `materialize` and the hook internals; `--json --timings` also adds a `timings` object (`elapsedMs` plus named phases). `mounts new --wait` blocks until the drive is attached instead of returning a `pending` mount. The installed hooks see `SWARMFILE_AGENT_MANAGED=1` and stand down, so the two paths never double-mount.

If the engine is not running, FUSE is unavailable (a container), or the mount is refused, the hook says so on stderr and exits 0; `SessionStart` waits up to 60 seconds for an attach, and `hooks run` exits 2 if it cannot mount at all. A crashed session's workspace is covered by its TTL (7 days by default, `--ttl-days` 1–30) and by the normal workspace sweeper.

MCP tools

The same workspace is available as Model Context Protocol tools with the published `swarmfile-mcp` server (Node 20+; the engine and `swarmfile` CLI must be running/on `PATH` , and the `search` tool also needs the standalone `swarmfile-search` on `PATH`):

```
claude mcp add swarmfile -- npx -y swarmfile-mcp
```

Claude Desktop, Cursor, Codex and VS Code configs are in the `swarmfile-mcp` repository. The tools cover workspaces, checkpoints, restore, blame/diff, conflicts, edit leases, agent sessions and search. The default policy is safe for a prompt-injected agent: `restore` , `conflicts_resolve` , `claim` and `finish_agent_session` need `SWARMFILE_MCP_ALLOW_DANGEROUS=1` , and `SWARMFILE_MCP_READ_ONLY=1` exposes only reads. `SWARMFILE_MCP_POLICY_ROOT` points `create_workspace` at a non-writable base checkout, so an agent cannot rewrite the `.swarmfile/agents.yml` that governs it.

The recorded two-agent run is on the demo page (`oss-agents`); the eight-agent harness is `scripts/demo/coding-agents.sh` . The npm listing is the source of truth for the MCP package; the public project's source snapshot refreshes with the next release.

Rewinding an agent

Every checkpoint is a normal commit, so the existing restore works unchanged:

```
swarmfile steps                              # the agent's step timeline
swarmfile checkpoint -m "before the refactor" # close the current batch now
swarmfile log --limit 10                     # find the one you want
swarmfile restore --to 482 --path src/       # back to that state (a new commit)
```

The web History page has the same view behind **Steps only**.

`--to` is an alias for the positional ref. Restoring never rewrites history: it creates a new commit that matches the old state, so the undo is itself undoable. Checkpoints exist only on the workspace's own branch; a person's `main` keeps the normal auto-commit cadence.

Repository policy: `.swarmfile/agents.yml`

Commit a flat policy file to the repository so every agent session and the GitHub Action use the same limits:

```
max_concurrent: 20
branch_prefix: agents
ttl_days: 2
checkpoint_idle_secs: 30
conflict_mode: hard
image: node:24-bookworm
paths_deny:
  - secrets/**
merge_diff3: true
```

The GitHub Action (`swarmfile-agent-action`) reads `ttl_days` , `conflict_mode` , `image` , `branch_prefix` , `local_checks` (running its `local-check-command`), `upstream_push` (`never` or `manual`) and `upstream_ci` (`pr_only` , `on_push` or `none` ; the caller's workflow mirrors that with `on: pull_request` / `on: push`). The engine reads `max_concurrent` , `checkpoint_idle_secs` and `paths_deny` on `mounts new` ; the MCP server (`swarmfile-mcp`) reads `max_concurrent` , `ttl_days` and `conflict_mode` for `create_workspace` , and passes its operator-owned `SWARMFILE_MCP_POLICY_ROOT` to the engine.

`merge_diff3` is the engine merge fallback when `.swarmfile/merge.yml` is absent. An explicit action input still overrides the policy.

The action installs the Linux `.deb` from the hub's anonymous release assets by default. A runner image may opt into its own Swarmfile toolchain with `allow-preinstalled: true` ; a preinstalled copy receives the repository key, so enable it only on an image you control.

7.1 Branch- and path-scoped keys, and read-only delegations

A **branch-scoped** key is a repository API key bound to one non-default branch. It keeps git, history and merge requests on that branch; a write to another branch is refused. A **path-scoped** key is a branch-scoped key that is also limited to a set of path globs (`src/**` , `docs/**`). It can list, read and (unless it is a read-only delegation) write only inside those paths, plus the branch's fork base where projected files live; anything outside is refused, including a copied CID whose manifest is not in scope.

Mint one from the CLI:

```
swarmfile api-key create agent --branch-scope agent/x \
  --path-glob 'src/**' --path-glob 'docs/**' --deny-glob 'secrets/**'
```

or from the dashboard's API keys page / the tray's API keys panel. The API is `POST /orgs/:orgId/api-keys` with `branchScope` , `pathGlobs` and `denyGlobs` (the request body is published in the hub OpenAPI). The scope is **set at mint time** — to change it, revoke the key and mint a new one. Each list holds at most 64 globs of at most 1,024 characters; a blank branch or glob is refused. Minting does not check that the branch exists; the hub refuses a dead or default branch on first use, and the UI/CLI warn that the branch must already exist and be non-default. A branch-only key may use git; adding path globs makes it engine/API-only.

Two path-key limits are worth knowing:

- **No git, history or commit walk.** Path keys are engine/mount keys: no changesets, LFS, merge requests, branch history, commit messages or project jobs list. `swarmfile log` /history against one is refused.
- **New content must be verifiable.** A path key's upload is recorded only after the hub verifies it (ciphertext is decrypted and hashed); a presigned upload is declined and falls back to the Worker upload. On an E2E repository (client-only keys) a path key cannot introduce new content, and a manifest with more than 8,192 blocks is refused. In-scope edits of existing files and copies of in-scope content still work. E2E git with a scoped key is a documented non-goal for now — use an owner credential. On a community server scoped keys are not supported (hosted-only until P1).

A delegation can be minted **read-only** (`repositoryRole: "read"`); every write through it is refused, so use it for an agent that only reviews. Agent session secrets are redacted from tool output, but a value shorter than 4 characters is not — use a longer value.

8. Copy-paste install prompt

Paste the block below into a fresh session in Claude Code, Cursor, Codex, or GitHub Copilot. It pins the CLI version this page documents (**0.3.17**): the agent checks `swarmfile --version` first and stops if the installed CLI does not match, rather than half-working against a different flag set. The prompt installs that harness's hooks, keeps the generated instructions file, and gives the agent the mounted-workspace commands.

```
# Swarmfile setup for this agent session

You are setting up Swarmfile for this coding-agent session. Work through the
steps in order, and report what you ran and what it printed. Do not install or
upgrade Swarmfile: this prompt assumes the CLI, engine, and repository are already
available.

1. Verify the pinned CLI.
   Run: swarmfile --version
   It must print: swarmfile 0.3.17
   If it prints a different version, stop and report the version. Do not
   install or upgrade anything.

2. Get onto a mounted workspace FIRST.
   This is not a clone or a checkout. If the directory you are in is already a
   Swarmfile mount, work there. If not, run:
     swarmfile mounts new --label <short-label> --wait
   It creates an `agent/<label>-<6 hex>` branch and waits until the drive is
   attached; cd into the mount path it reports. Do not open a second mount on
   the same directory.

3. Install this harness's hooks from inside that mount (or user-wide).
   Pick your harness:
   - Claude Code: claude-code
   - Cursor: cursor
   - Codex: codex
   - GitHub Copilot: copilot
   Recommended when the agent will create fresh agent/* workspaces: a
   user-wide install from anywhere:
     swarmfile hooks install <harness> --user
   (Copilot has no --user; use repository scope on the base branch instead.)
   Repository scope, run from the mount root:
     swarmfile hooks install <harness>
   It writes settings/instructions into the repository, but a later workspace
   inherits them only if they are committed on the branch it forks from. Do
   not install repository-scope hooks only on an ephemeral agent/* branch and
   expect the base branch to see them.
   Then run
     swarmfile hooks doctor <harness>
   (add --user if you installed there) and re-run the install only if it
   reports missing pieces. Keep the generated instructions file: Claude Code
   and Codex get a SKILL.md, Cursor a rule, Copilot an agent profile. Do not
   edit, move, or delete it. Codex asks you to review and trust new hooks with
   /hooks; Cursor does not run session hooks in its cloud agents.
   Harnesses load hooks at session start: restart the harness after installing
   (or use `swarmfile hooks run -- <command>` for this session).
   If you installed at repository scope, land the generated settings and
   instructions files on the branch the team uses (a checkpoint or
   `swarmfile commit`) so teammates and later workspaces see them.

4. Run commands in the workspace.
   To start a command in a fresh mounted workspace (including starting the
   harness itself), use:
     swarmfile hooks run -- <command>
   It mounts first, runs the command with its cwd inside the workspace, and
   checkpoints/finishes the workspace when the command exits. Inside an
   existing workspace, run installs and builds as:
     swarmfile run -- <command>
   so the repository's shared dependency cache is used.

5. Checkpoint after meaningful changes.
     swarmfile checkpoint --if-changed
   This is a no-op (exit 0, no output) when nothing changed. With hooks
   installed, mutating tool calls and turn end already checkpoint; an explicit
   checkpoint at a natural boundary is still fine.

6. Verify without writing to the hub.
     swarmfile --version                # must be 0.3.17
     swarmfile hooks doctor <harness>   # hooks and instructions present
     swarmfile status --json            # mounted repository, branch, pending creates
     swarmfile checkpoint --if-changed  # silent no-op when nothing changed
   swarmfile restore --to <seq-or-changeset> is non-destructive (it creates a
   new commit that matches the old state), but it is still a write: on a
   read-only repository, do not run it and do not attempt edits; report the
```

```
   repository as read-only.

What this setup does NOT do:
- It does not install the Desktop App, the engine, or the CLI.
- It does not work without a mounted repository: there is no clone, and a session
  that cannot mount says so on stderr and continues in its original directory.
- It does not capture non-mutating calls: only mutating tool calls are
  checkpointed (Write, Edit, MultiEdit, NotebookEdit, Bash, PowerShell; Codex
  also apply_patch). Reads and other non-mutating calls are not.
```

The pin is deliberate: a prompt that assumes flags from a newer CLI is worse than one that refuses to run. At repository scope the hooks call `swarmfile` from PATH and use the default cache; `--user` pins the absolute binary and cache. The Copilot adapter is a repository-scoped agent profile, not lifecycle hooks, so `--user` is refused for it.

9. Agent spend attribution

Coding-agent sessions can report their token usage to the org's **Agent spend** card without sending prompts or file contents. Collection is opt-in: an an owner enables it in Settings → Usage; a member's engine then posts token counts and model names (never transcript text) at most every five minutes. Checkpointed sessions are split into **landed**, **exploration**, **in review** and **unknown** cost by the hub's default-branch history and merged-MR chains; unknown is never counted as exploration.

Scan a transcript locally, with no upload and no running engine:

```
swarmfile usage agents --local
swarmfile usage agents --local --session claude-abc123 --json
```

The local scan keeps the transcript path locally, discards each line's content after reading its id/model/timestamp/token counts, and reports an unknown model as `unpriced` rather than guessing a price. The org ledger is the default (no `--local`) and needs a running engine; it reads only your own sessions as a member, the org's as an owner/admin. See `swarmfile usage` for the flags and JSON shape.

Budgets are a reported circuit breaker for new workspaces, not an invoice: a configured budget refuses work when the day's spend reaches it, and when a model has no price it refuses with `agent_budget_unpriced` rather than counting the unpriced tokens as zero. The price table is versioned and dated; treat the total as an estimate.

10. Service-minted agent tokens

A **Starter+** hosted org can approve a service delegation for an `agent/` workspace. An owner/admin — or a member with the repository root write (the repository creator or a whole-repository grant), for themselves — creates it for a service user (the account the delegation acts as), with an optional path-glob scope and a token TTL ceiling of at most one hour (900 seconds by default). The service user's PAT then mints a token for an exact `agent/<label>-<6 hex>` branch:

```
curl -sS -X POST \
  -H "Authorization: Bearer $SERVICE_USER_PAT" \
  -H 'content-type: application/json' \
  -d '{"branch":"agent/hydrate-abc123","ttlSeconds":900}' \
  "https://hub.swarmfile.com/agents/v1/orgs/$ORG/agents/$AGENT/delegations/$DELEGATION/tokens"
```

The response's `key` (`sf_key_…`) is a branch-scoped key valid for the requested TTL (15 minutes by default; the delegation's ceiling and the platform's 1-hour cap both apply). It narrows rather than replaces the PAT: path globs can only narrow it, and revoking the token, the delegation or the PAT (lineage) ends it. In an SSO-enforced org the mint needs an owner or an interactive session — a non-owner service user's PAT is refused by the same policy as any PAT. Hosted only: the community server has no hosted agent-delegation or service-minted agent-token surface. (Its short-lived TTL member keys are a separate, narrower credential, not a delegation or an agent token.)

A **glob-free** delegation's token can also call the branch archive in §11. A path-scoped token is engine/API-only and the archive refuses it.

11. Hydrating without a mount: the branch archive

When a tool cannot mount a drive, the hub can stream the branch as one tar:

```
# a live branch; paths are relative to the branch root
curl -sS -H "Authorization: Bearer $TOKEN" \
  "https://hub.swarmfile.com/orgs/$ORG/repositories/$REPOSITORY/files/archive?branch=main" \
  -o main.tar

# sparse + gzip (paths is repeatable; prefix or glob, e.g. src/**/*.ts)
curl -sS -H "Authorization: Bearer $TOKEN" \
  "https://hub.swarmfile.com/orgs/$ORG/repositories/$REPOSITORY/files/archive?branch=main&paths=src&paths=README.md&format=tar.gz" \
  -o main.tar.gz

# the manifest primitive (JSON); If-None-Match with a prior ETag -> 304
curl -sS -H "Authorization: Bearer $TOKEN" \
  "https://hub.swarmfile.com/orgs/$ORG/repositories/$REPOSITORY/files/archive/manifest?branch=main&paths=src" \
  -o manifest.json

# a folder; paths are relative to that folder (v2 filters/format/ETag)
curl -sS -H "Authorization: Bearer $TOKEN" \
  "https://hub.swarmfile.com/orgs/$ORG/repositories/$REPOSITORY/files/$FOLDER_ENTRY_ID/archive" \
  -o folder.tar

# LFS-hydrated (pointer files remain the default)
curl -sS -H "Authorization: Bearer $TOKEN" \
  "https://hub.swarmfile.com/orgs/$ORG/repositories/$REPOSITORY/files/archive?branch=main&lfs=1" \
  -o main-lfs.tar
```

The token is a member credential, a branch-scoped repository key, or a glob-free service-minted token. A path-scoped key is refused, and the folder form is member-token only (service callers are refused). The branch must be live; the archive is the caller's readable tree, not a repository-wide dump.

Limits: at most 5,000 **emitted** files, 480 walked directories and 1 GiB per request on hosted (256 MiB on a self-hosted community server). The walk (and the two route-level reads that precede it) scans at most 20,000 rows and 500 tree pages — those are scan limits, not an emitted file count, and hosted and self-hosted share them. At most 64 `paths` values of 1,024 characters each. On the community server archive starts are limited to 30 per minute **per repository** (shared by all callers); a 429 carries `Retry-After`. A cap or scan failure is a typed 413; an unsafe or ustar-unrepresentable path is a 422 (`archive_unsafe_path` / `archive_path_unrepresentable`); a malformed filter or unsupported format is a 400; E2E is 409; an unreadable **projected** leaf manifest is `503 archive_content_unavailable` (a materialized row can still fail mid-stream).

The branch response is `tar` by default; `format=tar.gz` is opt-in gzip (chunked, no `Content-Length`); zstd is not supported (`format=tar.zst` is a typed 400). A strong `ETag` is a generation hash (branch head/sequence, ACL/LFS generations, caller role/principals/mode and the selection); tar, gzip and manifest responses each have their own representation-specific ETag, so send the ETag from the response you are revalidating. A match returns 304 before the walk.

The manifest route returns the same filtered list plus `headSeq` (main included: it reads the live `repository_head_ref`), so a sandbox can keep its prior manifest, fetch only changed paths with `paths=`, and delete old paths absent from the new list. The manifest's `headSeq` — repeated as `X-Swarmfile-Archive-Seq` on both 200 and 304, so a cached manifest can refresh a stale base — is the branch sequence to pass to the batch checkpoint's `baseSeq`. There is no server-stored delta, resume or checksum.

Empty directories and zero-byte files are emitted; non-empty directories are implied by their members. Safe relative symlinks are emitted (absolute or root-escaping targets are a 422); a submodule/gitlink is an empty directory placeholder, not hydrated. An LFS pointer is archived as the pointer by default; `lfs=1` replaces it with the native content-map object (decrypted on managed tiers) or a standalone plaintext `lfs/<repository>/<oid>` object, and the hydrated size drives `Content-Length`/ETag. A missing or managed-encrypted standalone object is `503 archive_content_unavailable`; LFS object upload is not part of the archive.

Windows-hostile names such as `CON` or a trailing dot are allowed but a Windows extractor may reject them; traversal and absolute/drive/backslash/NUL paths are refused. Successful responses carry `X-Swarmfile-Archive-Branch`, `-Head`, `-Repository` and `-Format` (plus `ETag`). On the community server `-Head` can be a member's **provisional** claim rather than a confirmed commit hash; the `X-Swarmfile-Archive-Claim-Status` header says `confirmed`, `provisional` or `unknown`, and only `confirmed` is repository history. Hosted and self-hosted; both scan at most 20,000 rows, and a self-hosted community server cannot serve managed/E2E tiers.

12. Committing a sandbox's changes in one request

`POST /orgs/{org}/changesets/batch?branch=<branch>&machineId=<id>` stages and commits a branch's side-effects as **one atomic Mode-B changeset**:

```
curl -sS -X POST   -H "Authorization: Bearer $TOKEN"   -H 'content-type: application/json'   -d '{
    "repositoryId": "<repository id>",
    "message": "run tests + format",
    "baseSeq": 42,
    "changes": [
      {"path": "src/app.ts", "rootCid": "...", "size": 1234},
      {"path": "docs/new.md", "rootCid": "...", "size": 88},
      {"path": "src/old.ts", "delete": true}
    ]
  }' \
  "https://hub.swarmfile.com/orgs/$ORG/changesets/batch?branch=agent/task-123"
```

- Content is **content-addressed**: upload blocks/manifests first (the engine chunker or your client), then name each `rootCid` / `size`. The hub does not apply a unified patch. On a **plaintext `agent/*` branch**, a small file may instead carry `contentBase64`: the hub chunks one block server-side (4 MiB/ file, 16 MiB/batch), runs the same subscription, grace, community-email and storage gates as a block upload, and commits the real CID. The hub's inline form is refused on

managed/E2E repositories. The sandbox sync client is stricter: it writes content only to plaintext repositories and refuses a managed/E2E create or update with `content_requires_plaintext` before any upload (deletes remain allowed).

- A large non-LFS file is uploaded as BLAKE3 blocks with a flat manifest, or a segmented v2 manifest once one block would overflow. On a **plaintext** repository the sandbox hydrates a changed file above `directReadThresholdBytes` (default 256 MiB; `--direct-read-threshold-bytes` / `SWARMFILE_SANDBOX_DIRECT_READ_THRESHOLD_BYTES`) by fetching its root, segment and chunk blocks directly and verifying every BLAKE3 CID, so there is no archive-size ceiling for plaintext files. On a **managed** repository the same threshold routes the file through the hub's `/content` range read (decrypted server-side), so existing managed files above the archive cap also hydrate read-only. Sandbox content writes to managed remain refused; use the engine for managed writes. **E2E** has no server-side read path: sandbox sync is unsupported at every size (the archive is `409` and the client reports `e2e_unsupported`), not merely above a 1 GiB cap. A path-scoped key cannot direct-hydrate a large file; use a branch-scoped key.
- An existing path is updated; a missing path is created (missing parent directories too); `delete` removes a file. Directory deletes and renames are not supported.
- One changeset, one head advance. `baseSeq` refuses a stale base with 409; `id` is the client idempotency key (a retry while open finishes it; a committed id returns 409 `changeset_not_open`). At most 500 changes.
- Use a member token, a branch-scoped service key, or a **path-scoped** key for content-addressed changes.

  A path-scoped key is admitted only when every change path is inside its globs and every `rootCid` is this key's upload, the entry's previous root, or an in-scope branch entry's root; inline `contentBase64` is refused for it. `?branch` is required.

  The response is `{changeset, entries, changes:[{path,entryId,action}]}`; errors are `path_scope_forbidden`, `path_scope_content_not_owned`, `path_scope_provenance_too_large`, `batch_base_mismatch`, `batch_path_not_found`, `batch_path_is_directory`, `batch_path_not_file`, `invalid_batch_change`, `batch_path_not_directory`, `batch_projection_changed`, `batch_path_conflict`, `batch_path_reserved`, `batch_path_forbidden`, `too_many_entries`, `inline_content_requires_plaintext`, `inline_content_requires_agent_branch`, `commit_mode_locked`, `locked` / `protected_branch`, `archive_overlay_unavailable`.
- A symlink change is `{path, linkTarget}` (create/retarget) or `{path, delete:true}`. `linkTarget` must be relative and non-escaping; it cannot be mixed with content. The archive emits symlinks. Directory rename/delete is not part of the batch (use the entry rename/delete API).
- **More than 500 changes:** create an async batch with `POST /changesets/batch/async` (first <=500 content-addressed changes; reuse `jobId` / `chunkId` on retries; the changeset id is server-generated), append <=500-change chunks at `/changesets/batch/async/:jobId/changes` (4,000 total), then `/seal`.

  Poll `GET /jobs/:jobId` or cancel with `POST /jobs/:jobId/cancel`. The job pins the branch/base and re-checks ACL, path scope and provenance at execution; inline content is refused.

  The SDK exposes `createBatchCheckpointAsync`, `appendBatchCheckpointAsync`, `sealBatchCheckpointAsync`, `getRepositoryJob` and `cancelRepositoryJob`. A collecting job not sealed within 24 hours fails.
- Hosted and self-hosted. The self-hosted server uses its flat key model (no path scopes or hosted agent-delegation/service-minted agent tokens; it does mint short-lived TTL member keys), and a member key's batch is a provisional claim rather than a confirmed-head advance. It rebuilds the whole live branch tree per commit, so the base is capped at 100,000 entries by default (`SWARMFILE_SERVER_MAX_BATCH_TREE_ENTRIES`, up to 10,000,000); an over-cap branch is `413 batch_tree_too_large`. Batch starts are limited to 60/minute per repository there. The sandbox flow is: hydrate with §11, run/tests, upload changed content, then this one request. The agent SDK exposes `batchCheckpoint` (including `dryRun`) and `planSandboxSync`.

VS Code extension (no mount)

The Swarmfile VS Code extension opens a repository without mounting a drive. It hydrates the branch into a normal folder through the bundled sandbox-sync client, checkpoints your edits, and shows the agent token spend available to you beside the code.

Use it when you want a repository in VS Code without installing the desktop engine, or when a remote/SSH window cannot mount a drive. For a drive-backed workspace with native file locks, use the desktop app and the coding-agent workspace guide instead.

Install and connect

The 0.0.2 release is being prepared for the VS Code Marketplace and the direct downloads page. The Marketplace listing is the recommended path and handles Marketplace signing/publisher verification; the direct page will offer the same VSIX for manual installation and a signed Windows MSI that installs it through your VS Code CLI. The direct VSIX is a ZIP and is not itself Authenticode-signed; only the Windows MSI is. The links go live with the release. After installing, run **Swarmfile: Get Started**.

The wizard offers **Sign in with browser** (the site's app-link handoff; no token to copy) or a personal access token for a self-hosted server. Browser sign-in is supported in VS Code, VS Code Insiders and VSCodium, whose callback schemes the site accepts. Other editor builds the site does not recognize, and remote/SSH setups where the callback cannot reach a window, need a personal access token instead. It then lists your organizations, repositories and branches and asks for an empty folder to hydrate. The browser session is stored in the VS Code secret store and refreshed automatically; changing hubs requires signing in again.

The repository is addressed by its UUID; the picker fills that in for you. The hub, organization, capture consent and scan roots are application-scoped, so a workspace cannot redirect your token or turn on employer-visible telemetry. The token is bound to the hub it was saved for; changing hubs requires entering the token again.

Daily use

- **Sync & Checkpoint** hydrates remote changes and commits local changes in one batch. The Source Control view also exposes a **Checkpoint** input.
- **Switch Branch (New Folder)** hydrates another branch into a fresh folder; it does not rewrite the current working tree.
- **Open Mounted Repository** opens the installed drive root (`~/Swarmfile`, a namespaced `~/Swarmfile-<namespace>`, or `S:\`; set `swarmfile.mountRoot` for a non-default location). **Checkpoint Mounted Repository** delegates to an absolute, trusted installed CLI path. **Repair or Install Desktop App** opens the signed downloads page; the extension never runs a system installer, a PATH-resolved tool, or driver installation itself.
- **Show Agent Spend** reads the agent-spend summary available to you: estimated cost, uncached input, cache reads and writes, output, landed/in-review/exploration state, plugin and session tables, budgets, a custom window and CSV export. A plain member sees their own attribution; an owner or admin sees the organization-wide summary.

Agent usage and attribution

Capture is **off by default**. Enabling **Swarmfile: Capture Agent Usage Now** asks for consent, then reads local Claude Code transcripts and Cline/Kilo/Roo task stores for the hydrated workspace. It posts model, session id, harness, plugin, repository, branch, timestamps, message/token counts and the workspace folder name to the organization ledger. Prompts, tool arguments, file content and paths are never posted. A workspace can opt itself out, and the folder label can be omitted.

The extension skips sessions the desktop engine already records. That check reads the engine's default cache directory (`~/.cache/swarmfile`, or `%LOCALAPPDATA%\Swarmfile` on Windows) and a live namespaced directory beside it. If the engine's data directory is elsewhere or is not detected (for example it was started with its own `SWARMFILE_CACHE_DIR`), set the same `SWARMFILE_CACHE_DIR` for the VS Code extension host or point `swarmfile.agentUsage.engineCacheDir` at the engine's cache directory, or the same session can be counted twice. A server whose capability profile declares `agentUsage: false` is shown as unavailable.

Checkpoints can carry `Agent:` and `Swarmfile-Agent-Session` trailers. An explicit association (or the opt-in auto-association setting) claims the hub-verified commit. `Install Agent Commit Hook` adds a `prepare-commit-msg` hook for raw Git commits in a Git repository; the hub claim still needs a Swarmfile changeset, so the hook adds the display trailers only.

Limitations

- **Copilot Chat and Cursor** do not expose token usage through the public VS Code API, so their sessions remain `unreported`. Codex usage stays on the desktop engine path until a plugin adapter ships.
- **Other agent plugins** can call the extension's accounting-only `swarmfile.reportAgentUsage` command only after the user enables capture, the application-scoped `swarmfile.agentUsage.allowPluginReports` setting, stores an adapter token with **Swarmfile: Set Plugin Report Token**, supplies it as `bridgeToken` (per-adapter tokens are supported; blank sets the default), and approves the per-report confirmation. The bridge refuses the Cline/Kilo/Roo/Claude ids that are read natively; Continue and other adapters are not bundled.
- **No auto-install:** the extension does not download or run a system installer. Use the signed desktop installer for the drive.
- **The extension has a focused SCM surface:** staged/conflict groups with VS Code diffs and quick diff, file history, file-level selective commit, branch/MR actions and conflict resolution. Line-hunk partial commit is provided through the engine's commit-plan barrier for one regular, non-LFS staged file up to 8 MiB (the command saves the buffer first and never writes a partial version to disk); a native merge editor is not provided.
- **Authentication:** browser sign-in uses the site's app-link handoff and is supported in VS Code, VS Code Insiders and VSCodium; other editor builds the site does not recognize, and remote/SSH setups whose callback cannot reach a window, need a personal access token. The first-party IdP does not offer an authorization-code/PKCE grant, so a self-hosted or automation setup also uses a token.

Related

- Coding agents for mounted, headless workspaces.
- Git-LFS for source in git and assets in Swarmfile.
- Known limitations.

Runner (Headless CI)

A **runner** is a headless Swarmfile engine that watches one branch and executes a job when a commit or tag matches a rule — the same shape as GitHub Actions or a self-hosted CI runner, but built on the engine's existing content-addressed blocks instead of a full `git clone`.

Each rule match materializes the branch into a plain checkout directory (not a live mount) before running the job. By default that is every file in the tree — the checkout directory persists between runs, so a re-checkout only re-fetches blocks that actually changed since last time (an unchanged file is a fast local copy, not a network fetch), but the first run against a large repository, or one where most files change often, still pays for the full tree.

A top-level `sparse:` list narrows the checkout to the paths a job needs, so a huge repo only materializes the part it uses.

What it is not

This self-hosted runner has no job queue, no matrix builds, and no artifact storage — it is the trigger→execute→report loop only. For those, use Swarmfile's separate Hosted CI, which runs jobs on Swarmfile-managed containers, or a registered Self-Hosted CI runner, which claims queued `.swarmfile/ci.yml` jobs on hardware your organization controls. This runner is a different tool: it watches its own branch with `.swarmfile/runner.yml` and never participates in the hub's CI job queue. Run history is inspectable via the CLI, the Desktop App, and the web dashboard (below).

A runner is also deliberately **not a mounted engine**: it serves no control socket and never presents a drive, so the `swarmfile` CLI cannot talk to it. It is configured entirely by environment variables plus the `.swarmfile/runner.yml` in the branch it watches, and a job starts only when a matching commit or tag lands — this runner has no external "run this now" API.

On a machine that *does* run a mounted engine, `swarmfile tail` streams this repository's activity in real time; the org's outbound webhooks are dispatched when the event is recorded (typically within seconds, with the hourly tick as a backstop), but they are not a real-time build trigger; treat them as a reaction/integration channel.

The watched branch must be protected. `.swarmfile/runner.yml` decides what runs on the runner host, so a runner executes it only on a branch with effective branch protection — otherwise anyone who can push to that branch could run commands on the machine. `SWARMFILE_RUNNER_ALLOW_UNPROTECTED=1` overrides that and is not a production option — use it only on a trusted or throwaway runner.

For a job that isn't trigger-driven, a machine running a mounted engine can use `swarmfile materialize <ref> --out <dir>` — the same checkout, written to a plain directory with no FUSE, pinned to whichever ref you name.

Where rules live

Rules are **not** configured on the hub, in the Desktop App, or via the CLI — there is no `swarmfile runner rules` command, because there is nothing on the hub to create. A rule is a plain YAML file, committed into the branch it governs, at:

```
.swarmfile/runner.yml
```

This is deliberate: a rule change is reviewable in the same diff as the code it gates, exactly like a GitHub Actions workflow file. Whoever can commit to the branch controls what runs — the same trust boundary as any other file on that branch, not a new one.

```yaml
# Optional, top-level: check out only these paths (see "Checkout scope").
sparse:
  - "src/**"
  - "*.md"
rules:
  - name: build-and-test
    on: commit          # commit | tag — default: commit
    branches:            # glob patterns; omit to match every branch
      - main
      - "release/*"
    paths:                # optional — matches the full relative path
      - "*.dwg"
    run: "scripts/ci.sh"  # shell command, run with the checkout as CWD
    timeout: 1800          # seconds — default 1800 (30 min); on expiry the command's whole process tree is killed
```

A repository can define several rules; each is evaluated independently against every commit/tag on the watched branch.

`paths:` matches each changed file's full relative path. `*.dwg` matches `plan.dwg` wherever it lives in the tree; `assets/**` anchors to that folder. A leading `!` negates (gitignore-style, last match wins), so `paths: ["src/**", "!src/docs/**"]` skips doc-only changes; the same applies to `branches:`. Leave `paths:` off to run on every commit to a matching branch.

`on: tag` rules ignore `paths:`. A tag names a single commit; there's no "files changed since the last tag" the way there is for a fresh commit.

Checkout scope (sparse)

By default a matched run checks out the whole branch. For a large repo, add a top-level `sparse:` list to materialize only the paths a job needs:

```
sparse:
  - "src/**"
  - "assets/models/*.bin"
  - "*.md"
rules:
  - name: build
    run: "make -C src"
```

A path is checked out when it matches a glob, or when a directory in its ancestry matches.

- A glob with no `/` matches by name at any depth (`*.md` finds `docs/notes.md`).
- A glob containing `/` is anchored to the repo root (`assets/models/*.bin` does **not** match `vendor/assets/models/x.bin`).
- A leading `!` excludes a path an earlier glob included (gitignore-style, last match wins), so `["src/**", "!src/**/*.test.ts"]` checks out `src/` without its tests; the top-level `exclude:` key is sugar for those `!` negations, and with no `sparse:` it means "everything except".
- Paths that match nothing are neither written nor fetched: a directory the filter cannot reach is skipped without being walked (and a frozen commit tree is not even fetched), and a directory holding no matching file is not created.
- Only **anchored** globs enable that pruning — a bare name like `*.md` can match at any depth, so the walk cannot skip anything; anchor with a `/` (`docs/*.md`) when you can.

`sparse:` is **top-level, not per-rule**: a runner has one checkout directory and materializes it once per event for every rule that matched, so a per-rule scope could not be honored. Omit it (or leave it empty) for a full checkout. Changing the list between runs converges the directory: paths that drop out of scope are removed, the newly included ones are fetched. At most 64 globs of 1024 characters each, and at least one non-`!` glob; an invalid list fails the affected runs with the reason (visible in `swarmfile runner runs`) rather than silently checking out everything.

Running a runner

A runner is a normal `swarmfile-engine` process started with `SWARMFILE_RUNNER_MODE=true` and a repository API key (see CLI: swarmfile for `swarmfile api-key create` — the key counts against your plan's headless-seat allowance; see Billing & Plans). Like seed mode, it never mounts a drive and never prompts for interactive sign-in; unlike seed mode, it materializes a real checkout directory (under its cache dir) because a job needs actual files on disk to run against — not a FUSE/WinFsp mount, which would require a kernel driver on a CI box that may not have one.

```
export SWARMFILE_API_KEY=sf_key_...
export SWARMFILE_ORG_ID=org_abc123
export SWARMFILE_REPOSITORY_ID=proj_xyz789
export SWARMFILE_BRANCH=main
export SWARMFILE_RUNNER_MODE=true
swarmfile-engine
```

With a **repository API key** (`sf_key_…`) the runner is zero-config: it asks the hub which org and repository the key belongs to and fills both in, so `SWARMFILE_ORG_ID` and `SWARMFILE_REPOSITORY_ID` may be omitted.

With a personal access token, `SWARMFILE_REPOSITORY_ID` is still needed, and `SWARMFILE_ORG_ID` is needed unless the hub URL already ends in `/orgs/<id>`; both are needed against a hub old enough not to serve the key-lookup route. Every request is routed per org (`/orgs/<id>/repositories/…`), so a bare hub URL with no key context has no org scope and the hub answers `404`; a hub URL that already ends in `/orgs/<id>` supplies the org too.

The key must be a **repository API key**, not a personal access token: Swarmfile's hosted hub accepts only a repository API key as the writer of runner run records, so the runner **refuses to start** with a PAT there unless `SWARMFILE_RUNNER_ALLOW_PAT=1` is set (jobs then run, but every report is refused with `403 runner_run_requires_service_caller` and the Runs tab stays empty). A community server has no runner-runs route at all, so a PAT is a warning there.

The lookup runs in the runner daemon; **job steps run with Swarmfile credentials stripped** (a repo-controlled `run:` must not be able to exfiltrate the runner host's key), so a step that needs the repository uses the materialized checkout, and a `swarmfile://` URL carries its own org/repository. The standalone git helper does not read the app config; see Cloning a Repository with Git.

`swarmfile-runner` is the same thing as its own binary: it sets `SWARMFILE_RUNNER_MODE=true` (when unset) and re-execs the sibling `swarmfile-engine`, so the CI role reads as its own tool without an env-var prefix. It ships with the installers — `/usr/bin/swarmfile-runner` via the Linux `.deb`, inside the macOS app bundle (with a `/usr/local/bin` symlink), and next to `swarmfile-engine.exe` from the Windows MSI. On any engine binary the env-var form above is equivalent. See `swarmfile-runner` for its flags and acquisition. It takes the same env and flags:

```
SWARMFILE_API_KEY=sf_key_... SWARMFILE_REPOSITORY_ID=proj_xyz789 swarmfile-runner
```

`SWARMFILE_BRANCH` is the same setting a normal branch-pinned mount already uses (see Branches & Merging) — "which branch does this runner watch" needs no setting of its own.

On startup the runner replays any commits it missed while it was offline (via the branch's commit log), then listens live. A crash or restart never silently skips a commit.

Before checking anything out, a triggered run waits up to 120 seconds for a file that is still uploading on the branch — a mid-change tree would be the wrong thing to build. If the upload is still live at the deadline, the run is not started: it is recorded as a `failure` whose log says it was skipped for a live upload (visible in `swarmfile runner runs`), rather than building from a partial tree. There is no separate `skipped` status. A shutdown during the wait ends it immediately, treated the same way.

Checking out a ref in your own CI

If you already have a CI system, you don't need runner mode to get a checkout: `swarmfile materialize <ref> --out <dir>` writes a ref's tree to a plain directory with **no mount and no FUSE**, so it works on a container with no kernel driver. Pin by **hash** — a tag is a name that can be repointed by delete + recreate, and a branch

moves; the 64-hex hash is the only immutable pin (a commit's hash is shown in the web commit log, MR header, and release page, and on `swarmfile log` / `show`):

```
swarmfile materialize hash:$SWARMFILE_COMMIT_HASH --out "$PWD/checkout"
```

`--json` echoes the resolved `commitHash`, `kind`, and `seq` — record them in your build metadata so an artifact names the exact tree it was built from.

Add `--sparse <glob>` (repeatable) to check out only the paths a job needs — the same scoping as the runner's top-level `sparse:`, with the same glob rules:

```
swarmfile materialize hash:$SWARMFILE_COMMIT_HASH --out "$PWD/checkout" --sparse "src/**"
```

A few things to know before relying on it:

- **The executable bit is preserved.** A file marked executable (with `chmod +x` on the drive, `swarmfile chmod +x`, or a git push of mode `100755`) checks out executable on macOS and Linux, so `./scripts/ci.sh` runs as-is. Windows has no executable bit.
- **The marker owns the directory.** A new or empty destination is materialized fresh; one this tool already wrote is updated in place, pruning only the paths in its marker, so your job's own build outputs and caches survive a re-checkout. A non-empty destination with no marker, or one owned by a different repository, is refused rather than merged or deleted — there is no `--prune` flag.
- **One destination, one writer.** The marker's written-paths list is read and rewritten per run with no cross-process lock. Give each job its own `--out` directory; two concurrent materializations into the same one can interleave.
- **Symlinks are written, not copied.** On Windows that needs Developer Mode (or elevation); a symlink the OS refuses is reported in the run's per-file `failures` rather than silently skipped, like any other entry that can't be written.
- **Not every commit has a hash.** Provisional commits (not yet confirmed) and archived commits have none; address those by seq or tag. An unknown hash is the typed `hash_not_found`, not a silent fallback. On the hosted hub a pushed commit is confirmed by the claim-triggered derivation within seconds; a community server confirms a member key's push from its own tree derivation at claim time (a claim it cannot reproduce waits for another key's claim).

`swarmfile clone <repository-id> <path> --checkout -b <ref>` is the git-familiar spelling for the same operation, cross-repository included: `-b` takes the full ref vocabulary (branch, tag, hash, seq), and `--sparse <glob>` (repeatable) narrows the checkout exactly as it does for `materialize`. The mount form of `clone` keeps `-b` branch-only, because a mount works one branch and a tag or hash has no branch to live on; it refuses `--sparse`, since a mount always shows the whole branch.

`materialize` still needs a running engine (the CLI is control-socket-only), so on a driverless box it's usually a mounted engine elsewhere doing the checkout, or the runner above doing the whole loop. The `runner-runs` check-reporting API below needs no engine at all.

Inspecting run history

```
swarmfile runner runs                  # every branch in the repository
swarmfile runner runs --branch main    # narrowed to one branch
```

Each row carries the matched rule's name, the triggering commit, status (`running` / `success` / `failure` / `timed_out`), exit code, and a truncated log. There is no separate `swarmfile runner runs show <id>` command — the list output already includes each run's log inline, so there's nothing further to drill into.

The same history is browsable without the CLI: in the Desktop App, via the repository's **CI runs** page on the activity bar; on the web dashboard, under the repository's **Runs** tab (open to every repository member, like **Files** and **History** - guests are refused).

Both surfaces also offer **Run workflow** and **Cancel** for a repository's CI runs — hosted or on a registered self-hosted runner — never on this branch-watching runner, whose runs are only ever created and finished by the runner engine. The adjacent **CI** tab, which holds hosted-CI settings, secrets and runner sizes, is the admin/owner-only one (like **Commits**, **RFIs** and **Permissions**).

Reporting checks from your own CI system

You don't have to run Swarmfile's own runner to gate a protected branch's `--require-checks-to-pass` — any CI system that can make two authenticated HTTP calls can report a check result the same way a runner does. The `runner-runs` API is deliberately not tied to the runner binary: it only requires a repository API key, the same credential a runner engine uses (see `swarmfile api-key create`).

The two calls are the whole contract. They're repository-scoped routes under your org and repository, so the examples assume `SWARMFILE_HUB_URL` (`https://hub.swarmfile.com`), `SWARMFILE_ORG_ID`, `SWARMFILE_REPOSITORY_ID` and `SWARMFILE_API_KEY` are set, plus three variables your job sets itself: `BRANCH_ID`, `BRANCH_NAME` and `COMMIT_SEQ`, the branch id, branch name and commit number your job ran against (where to get each is explained below the example; the engine doesn't set them):

```
RUNS="$SWARMFILE_HUB_URL/orgs/$SWARMFILE_ORG_ID/repositories/$SWARMFILE_REPOSITORY_ID/runner-runs"

# Start of your job — creates a `running` row so a crash reports as failure,
# not silence.
RUN_ID=$(curl -sf -X POST "$RUNS" \
  -H "Authorization: Bearer $SWARMFILE_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "repositoryId": "'"$SWARMFILE_REPOSITORY_ID"'",
    "branchId": "'"$BRANCH_ID"'",
    "branchName": "'"$BRANCH_NAME"'",
    "changesetSeq": '"$COMMIT_SEQ"',
    "ruleName": "external-lint"
  }' | jq -r '.run.id')

# ...your actual CI job runs here...

# End of your job — reports the real outcome.
curl -sf -X PATCH "$RUNS/$RUN_ID" \
  -H "Authorization: Bearer $SWARMFILE_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{"status": "success"}'   # or "failure" / "timed_out"
```

What each field means for gating, exactly as a Swarmfile-runner-produced row would be read:

- `ruleName` is an arbitrary label you choose — it's what shows up as one named check.
- `requireChecksToPass` first narrows to only the runs reported for the *exact* commit being merged, then requires the *latest* run of *every distinct* `ruleName` *reported at that commit* to be `success`. A rule that reported at an earlier commit but hasn't reported again at the new head isn't carried forward as still-required — it simply isn't part of the gate for this head, the same as a rule that's never run at all; a branch with zero matching runs at its head fails closed rather than passing vacuously.
- `changesetSeq` must be the commit number of the exact commit your job ran against (`swarmfile status` / `mr status` locally, or resolved from whatever your CI already checked out) — a check reported against an older commit doesn't satisfy a merge request whose branch has since moved past it.
- `branchId` is the branch's real id, not its name; resolve it once via `swarmfile branch list` or `GET /orgs/<org>/repositories/<repository>/branches`.

A GitHub Actions job reporting into this looks like (with `SWARMFILE_HUB_URL`, `SWARMFILE_ORG_ID` and `SWARMFILE_REPOSITORY_ID` set in the job's `env:`):

```
- name: Report check start
  run: |
    RUN_ID=$(curl -sf -X POST "$SWARMFILE_HUB_URL/orgs/$SWARMFILE_ORG_ID/repositories/$SWARMFILE_REPOSITORY_ID/runner-runs" \
      -H "Authorization: Bearer ${{ secrets.SWARMFILE_API_KEY }}" \
      -H "Content-Type: application/json" \
      -d '{"repositoryId":"...","branchId":"...","branchName":"...","changesetSeq":...,"ruleName":"gh-actions-build"}' \
      | jq -r '.run.id')
    echo "RUN_ID=$RUN_ID" >> "$GITHUB_ENV"

# ...your build/test steps...

- name: Report check result
  if: always()
  run: |
    STATUS=${{ job.status == 'success' && 'success' || 'failure' }}
    curl -sf -X PATCH "$SWARMFILE_HUB_URL/orgs/$SWARMFILE_ORG_ID/repositories/$SWARMFILE_REPOSITORY_ID/runner-runs/$RUN_ID" \
      -H "Authorization: Bearer ${{ secrets.SWARMFILE_API_KEY }}" \
      -H "Content-Type: application/json" \
      -d "{\"status\": \"$STATUS\"}"
```

This is exactly what Swarmfile's own runner does internally — there's no separate, more-privileged path only the runner binary can use. A repository API key is the whole trust boundary, same as everywhere else in the API (see Security below).

Security

A rule's `run:` command runs with the permissions of that machine's user — the same trust model as a self-hosted GitHub Actions runner. The runner's API key itself is narrowly scoped: it can create and update its own run records, plus everything any other repository-scoped API key can already do (read metadata, download blocks, read/write changesets and branches). It cannot post comments, touch ACLs, or reach admin routes — the same restriction every API key has.

Hosted CI

Hosted CI runs the workflows in `.swarmfile/ci.yml` on Swarmfile-managed containers. There is nothing to install on a build machine and nothing to keep patched: a commit (or a manual dispatch) starts a container, the container streams the job's log back as it runs, and the result lands on the branch as a normal check.

Hosted CI is off for a repository until an owner enables it, and a workflow only runs automatically from a branch with **effective protection** unless the repository explicitly opts into unprotected branches (see Security). Every job is metered per second while it runs.

The reason to choose it is the data plane, not GitHub-compatibility: a job streams the paths it declares from the repository's block store instead of cloning the whole repository, and a self-hosted runner can claim the same job on hardware that already holds the bytes. The dialect is native and the importer translates the run-centric subset of an existing GitHub workflow; a workflow that depends on a marketplace action is refused rather than silently changed, except a small pinned set of `setup-*` steps that become native toolchain checks. Do not read it as a drop-in Actions replacement.

For a headless runner you host yourself, see Runner as CI; for a runner that claims hosted-CI jobs on your own hardware, see Self-Hosted CI Runners. The hosted service below shares the check and status vocabulary but not the machine.

Enable it

1. Open the repository in the web app → **CI** tab (or the tray's **Repository settings → Hosted CI**, or run `swarmfile ci enable`).
2. Toggle **Enable hosted CI for this repository**. This is the cost guard: no workflow runs until an owner turns it on.
3. Commit a `.swarmfile/ci.yml` to the branch you want to run from.

The **CI** tab also holds the repository's secrets and variables and shows the runner-size catalog (every built-in and the org's custom sizes, each with its per-second rate, read-only). Custom sizes are org-wide and are managed in **Settings → Hosted CI** (or `swarmfile ci size`).

The **Runs** tab lists every hosted run on the branch, with a per-job log viewer and a **Cancel** button; like **Files** and **History** it is member-only — external collaborators (guests) are refused.

A finished run can be re-run from its page (`swarmfile ci rerun <run-id>`), and a single failed job re-queued (`swarmfile ci retry <job-id>`). A run that ends in failure — including a timeout, a cancel, a queue expiry, a blocked start or an infrastructure fault — notifies its starter, and on a protected branch that branch's recent committers too, by bell/tray and the personal webhook, not email (see Notifications).

A first workflow

```
version: 1
name: Build and test

on:
  commit:
    branches: [main]
  manual: true

jobs:
  test:
    size: medium
    timeout: 20m
    steps:
      - name: Install
        run: npm ci
      - name: Test
        run: npm test
```

Commit that to a protected `main` and a run starts on the commit. The **Runs** tab shows it live; `swarmfile ci logs <job-id>` streams the same text in a terminal, the run view's **Download** button saves the whole job log as a `.log` file, and its **Search** box finds lines in the job's log server-side (case-insensitive, 50 matches by default).

Triggers

Trigger	Runs when	Filters
`on.commit`	A commit lands	`branches` (globs, `!` negates), `paths` (changed paths; empty match means no run)
`on.tag`	A tag is created	`pattern` (globs)
`on.manual`	Someone dispatches it — the Runs tab's **Run workflow**, the tray, or `swarmfile ci run`	`inputs` (optional; see below)
`on.merge_request`	An MR opens, its head moves, it reopens or becomes ready for review	`types`, `branches` (base-branch globs), `paths`
`on.schedule`	A 5-field UTC cron expression fires	`crons`, `branches` (default branch if omitted)
`on.metadata`	A commit changes folder-profile field values (edits made by API keys or CI don't trigger it)	`branches`

A `paths` filter on a commit or an MR compares against the changed entries; if the change set is unknown the filter matches nothing (a run is never started on a guess). A schedule fires at most once per missed occurrence: a hub that was asleep catches up the **latest** due firing, not every one.

Manual inputs

A manual trigger can declare typed inputs. The dispatch form (web or tray) renders them automatically, and the CLI passes them with `swarmfile ci run --input name=value` (repeatable; values travel as strings and coerce to the declared type):

```
on:
  manual:
    inputs:
      environment:
        description: Where to deploy
        type: choice
        options: [staging, production]
        default: staging
      dry_run:
        type: boolean
        default: false
      retries:
        type: number
        required: true
```

Types are `string` (default), `boolean`, `number` and `choice` (with `options`). `required: true` with no default must be supplied; an optional input with no default gets the type's zero (`""`, `false`, `0`). Values are read as `${{ inputs.<name> }}` anywhere an expression is allowed — including a job `if:`, which is evaluated when the run is created, so a job can be skipped statically. An unknown input name, a missing required one, or a value that does not match the declared type (or choice options) refuses the dispatch with a visible `ci_input_invalid` blocked run. Input names use letters, digits and underscores.

Jobs and steps

```
jobs:
  build:
    name: Build
    size: large            # small | medium (default) | large | xlarge, or a custom size
    # runs-on: [self-hosted, linux, x64]   # self-hosted selector (1–8 unique
    #                                      # lower-case labels, self-hosted first)
    image: swarmfile-node  # optional; default swarmfile-standard (see Runner images)
    environment: production # optional; binds the job to an org environment (see Environments)
    approval: true         # optional; force an approval even if the environment names none
    timeout: 30m           # default 30m; 120m maximum (Enterprise up to 360m)
    needs: [setup]         # waits for every listed job
    continue-on-error: true
    if: success()
    env:
      NODE_ENV: production
    secrets: [NPM_TOKEN]   # ONLY these secrets are readable by this job
    outputs: [artifact]
    sparse: ["src/**"]     # optional — check out only these paths
    steps:
      - id: prep
        name: Prepare
        run: echo "sha=$(git rev-parse HEAD)" >> "$CI_OUTPUT"
      - name: Publish
        if: success()
        working-directory: packages/app
        shell: bash
        env:
          TOKEN: ${{ secrets.NPM_TOKEN }}
        run: npm publish --provenance
```

- Every job runs in its own fresh container with a clean checkout of the triggering commit.
- `run:` is a shell script; the default shell is `bash -e -u -o pipefail`, overridable per step.
- `if:` on a job or a step accepts `success()`, `failure()`, `always()`, `canceled()`, comparisons, `!`, `&&`, `||`, parentheses, and the functions `contains`, `startsWith`, `endsWith`, `format`, `join`, `toJSON`, `fromJSON`. A condition the runner cannot evaluate is treated as **false** — a job never runs on a condition no one can explain.
- `needs` jobs that did not succeed cause their dependents to be skipped (`ci_needs_failed`) — unless the dependent carries an explicit `if:` other than a literal `success()`, which is admitted and evaluated with `failure()` true for the failed need. A job with `continue-on-error: true` counts as success for `needs`.
- `sparse:` checks out only the listed paths, so a huge repo only materializes what the job uses — see Partial checkout.

Partial checkout

Every job gets a clean checkout of the triggering commit. For a large repo, `sparse:` narrows it to the paths a job needs:

```
jobs:
  test:
    sparse:
      - "src/**"
      - "*.md"
    steps:
      - run: npm test
```

A path is checked out when it matches a glob, or when a directory in its ancestry matches. A glob with no `/` matches by name at any depth (`*.md` finds `docs/notes.md`); one containing `/` is anchored to the repo root (`assets/models/*.bin` does **not** match `vendor/assets/models/x.bin`). Everything else is neither fetched nor written, and a directory holding no matching file is not created.

A leading `!` excludes a path that an earlier glob included (gitignore-style, last match wins; at least one non-`!` glob is required), and `#` comments are not supported. `exclude: [glob]` is sugar for those `!` negations — and with no `sparse:` it means "everything except" (check out the whole tree, then drop the exclusions).

At most 64 globs of 1024 characters each. Omit `sparse:` / `exclude:` (or leave them empty) for a full checkout. The same globs work for the self-hosted runner and `swarmfile materialize --sparse`. Importing a GitHub workflow? An `actions/checkout` step's `sparse-checkout` becomes this field.

Read-only workspace

`mount: read-only` runs the job on a live read-only FUSE mount of the job's commit tree. The mount serves the exact pinned commit (a stored git tree, or the commit's entry listing when nothing has derived a git tree yet — a dashboard/API or manual run), so a step reads files without a local copy and a write fails with a read-only filesystem error. The step and the engine share a uid, so this is a guard against silent divergence, not a security boundary. A mount failure fails the job before any step runs. (On a runner image that predates the live mount, the workspace falls back to the older materialized read-only tree.)

```
jobs:
  audit:
    mount: read-only
    steps:
      - run: ./scripts/verify-release.sh
```

Hosted runners are **Linux amd64** only; Windows and macOS jobs run on self-hosted runners (see Self-hosted runners). A matrix `os:` value is a variable you use in steps — it does not select a runner image.

`sparse:` / `exclude:`, `downloads` and `cache` are refused with `mount: read-only` (the live mount is full-tree; extraction/cache writes would contradict it). Artifact uploads are reads and remain available. `mount: read-write` is refused; a writable job uses `mount: write` with `write: [commit]` (below).

Writable workspace and CI commits

`mount: write` is the read-only mount's writable counterpart: the job's workspace is the run's pinned commit tree, and when every step succeeds the engine commits the step-made changes back to the run's branch as one commit authored by the CI identity (`ci/<orgId>/<repositoryId>@<runId>`). The workspace is a **live writable mount**: the pinned tree is mounted read-only and overlaid locally (a userspace overlay), so a step edits the mount directly — there is no separate checkout copy — and writes stay local until the job-end commit. A failed or canceled job commits nothing.

A self-hosted runner provides the same live mount on Linux (the host needs `fuse-overlayfs`) and macOS (the runner serves the workspace through FUSE-T). It requires `write: [commit]` (and the scope requires the mount).

The branch must be **unprotected** — a protected branch refuses a direct CI commit exactly as it refuses a human direct write; use `write: [mr]` there — and the repository Open-ACL (a CI identity cannot be an entry principal). The branch head must still be the run's pinned commit (`ci_commit_base_mismatch` refuses a moved head), and a target carrying an explicit ACL deny is refused.

```
jobs:
  format:
    mount: write
    write: [commit]
    steps:
      - run: ./scripts/format.sh
```

The diff is exactly the workspace: created, updated and deleted paths, at most 500 per commit. `sparse:`, `downloads`, `cache` and `snapshot:` are refused with `mount: write` (they would add files the diff would commit or hide files it would read as deletes), and a symlink or special file left in the workspace refuses the commit by name. A refused commit fails the job, exactly like a refused tag or merge request. A non-Open-ACL repository or a protected branch is refused; use `write: [mr]` and merge the request there.

Matrix jobs

```
jobs:
  test:
    matrix:
      os: [linux, windows]
      node: ["18", "20"]
    steps:
      - run: echo "${{ matrix.os }} on node ${{ matrix.node }}"
```

Each combination becomes its own job (`test (os=linux, node=18)`); a job that `needs` a matrix waits for all of its rows. `include:` adds or augments combinations and `exclude:` removes them, with GitHub's semantics. At most 64 combinations per job, 200 declared jobs per workflow and 256 expanded jobs per run — past the combination cap the workflow is refused `ci_matrix_too_large` ; past a job cap, `ci_too_many_jobs` (declared) or `ci_jobs_too_many` (expanded).

GitHub's `strategy:` wrapper is accepted too: `strategy.matrix` works exactly like `matrix:` , and `strategy.fail-fast` and `strategy.max-parallel` are accepted for compatibility but ignored (concurrency follows your plan). Any other `strategy` key is refused `ci_unsupported_syntax` .

Expressions and environment

`${{ … }}` works in `env:` , `run:` , `working-directory` and `if:` . **Never interpolate an expression directly into `run:` for a value a teammate can influence** (`${{ inputs.* }}` , `ci.branch` , `ci.actor` , `vars.*`): like GitHub, the value is substituted into the shell command verbatim, so `run: ./deploy.sh ${{ inputs.environment }}` lets a dispatcher inject shell syntax.

Pass it through `env:` instead — `env: { TARGET: "${{ inputs.environment }}" }` then `run: ./deploy.sh "$TARGET"` . Only `steps.*` outputs are substituted whole at run time; an expression that *combines* a step output with other values inside `run:` or `env:` is not evaluated (the literal text reaches the shell) — compute it in a step and reference the output.

The context is:

Context	Contents
`ci.*`	`branch` , `sha` , `tag` , `actor` , `event` , `run_id` , `run_number` , `status`
`env.*`	job `env:` values in a job `if:` ; a step's own `env:` is not visible to that step's `if:` (declare the value at the job level to branch on it)
`vars.*`	repository variables, then org variables (repository wins)
`secrets.*`	the job's declared secrets only
`needs.<job>.outputs.*`	a completed dependency's `$CI_OUTPUT` values
`steps.<id>.outputs.*`	an earlier step's `$CI_OUTPUT` values
`matrix.*`	the job's combination values
`inputs.*`	the manual run's resolved `workflow_dispatch` inputs

The runner exports `CI=true` , `CI_BRANCH` , `CI_SHA` , `CI_TAG` , `CI_ACTOR` , `CI_RUN_ID` , `CI_RUN_NUMBER` , `CI_JOB_KEY` , `CI_JOB_ID` , `CI_WORKSPACE` , `CI_REPOSITORY_ID` , `CI_ORG_ID` , `CI_EVENT` and `CI_OUTPUT` (the `name=value` file for job outputs). Print `::add-mask::<value>` and the runner masks that value (and the job token) out of the log from that point on.

Composite actions

Shared logic can live in a local composite action: a directory under `.swarmfile/actions/<name>/` containing an `action.yml` with `inputs:` and `runs.steps` (run-only steps; no nested `uses` , no `timeout`). A job step uses it with a repo-relative path and passes inputs with `with:` :

```
# .swarmfile/actions/setup/action.yml
name: Setup
inputs:
  node:
    default: "20"
runs:
  using: composite
  steps:
    - run: echo "using node ${{ inputs.node }}"
```

```
# .swarmfile/ci.yml (jobs.<id>.steps)
- uses: ./.swarmfile/actions/setup
  with:
    node: "22"          # a scalar, or one whole ${{ … }} expression
- uses: ./.swarmfile/actions/setup
  with:
    node: ${{ matrix.node }}   # the outer expression survives
```

The action's steps are expanded into the job at compile time: every `${{ inputs.<name> }}` becomes the caller's `with` value (or the declared default; a required input with neither refuses the workflow). The caller's `env:` , `if:` and `continue-on-error:` apply to the expanded steps (the action's own step `env` wins on a name collision). A missing action file refuses the workflow `ci_action_not_found` ; remote/marketplace `uses:` stays refused in the native dialect (the GitHub importer translates a few pinned `setup-*` actions — see Migrating from GitHub Actions).

Because the whole expansion happens before the run is created, an action change takes effect on the next run that reads the branch — and `swarmfile ci run` (manual dispatch) recompiles the branch head like any other run.

Reusable workflows

Shared job graphs can live in a local workflow file. A job calls one with a repo-relative path; the callee's jobs are expanded into the caller at that call (keys `<caller>--<job>`), so needs, artifacts, caches and secrets behave like ordinary jobs:

```
# .swarmfile/workflows/node.yml
on:
  workflow_call:
    inputs:
      node:
        type: string
        default: "20"
jobs:
  test:
    env:
      NODE_VERSION: ${{ inputs.node }}
    steps:
      - run: node --version
```

```
# .swarmfile/ci.yml
jobs:
  lint:
    steps:
      - run: npm run lint
  test:
    uses: ./.swarmfile/workflows/node.yml
    needs: lint          # the caller's needs apply to the callee's root jobs
    with:
      node: "22"
```

The callee is read from the **same commit** as the caller (a manual run recompiles the branch head). `workflow_call.inputs` support `string`, `boolean`, `number` and `choice`; a missing required input, an unknown `with:` key or a path outside `.swarmfile/workflows/` refuses the workflow. v1 limits: local `.swarmfile/workflows/<name>.yml` only, one level (a callee cannot call another reusable workflow), no `secrets:` inheritance and no `outputs` — the callee's jobs read repository secrets directly. A `uses:` job may set only `uses`, `with`, `needs` and `name`; at most 10 calls per workflow.

Why a queued job is waiting

A queued job now shows a non-terminal reason in the run detail and in `swarmfile ci status`:

Reason	Meaning
`self_hosted_runner`	a self-hosted job has no matching live runner
`ci_admission_deferred`	compute quota/usage state is being refreshed
`ci_paused`	hosted CI is paused by the operator
`ci_concurrency`	the plan's concurrency budget is full

The reason clears when the job starts; `waitingSince` preserves the original wait start across repeated reports.

Writing a tag from a job (P0)

A job can create **one declarative tag at its own pinned commit** after every step succeeds:

```
jobs:
  release:
    write: tag
    tag: v1.${{ ci.run_number }}
    steps:
      - run: ./build.sh
      - run: ./publish.sh
```

Rules:

- Both keys are required: `write: tag` without `tag`, or `tag` without `write: tag`, fails compilation.
- The tag is created by the CI identity (`ci_job:<job>`) at the run's exact commit and branch; a request cannot name another commit.
- The job's branch must be **effectively protected**. The repository's `allowUnprotectedCi` setting does **not** widen writes.
- No force or move: an existing tag at another commit is refused; the same commit is idempotent. A CI-created tag does **not** evaluate tag triggers (no self-trigger loop).
- `tag` must resolve when the job is prepared (`ci.*`, `env`, `vars`, `matrix`). Step outputs are not supported in P0.
- Writes are opt-in per job **and** per deployment: a deployment that has not enabled CI write tokens fails the job at preparation (`ci_write_disabled`) rather than silently skipping the tag.

`write: commit` is described above (writable mount and CI commits). Artifact attestation is **not** built yet.

Cloud OIDC and merge requests

Two more write scopes are available:

```
jobs:
  deploy:
    write: [oidc, mr]
    environment: production
    steps:
      - run: ./build.sh
      - run: ./deploy.sh "$(swarmfile ci oidc --audience sts.amazonaws.com)"
```

- `write: [oidc]` lets the job mint a short-lived (10 minute) RS256 JWT with `sub = ci:<orgId>:<repositoryId>:<jobId>` and the repository/ref/commit/run claims. The audience must be one an owner allowed for the organization; an empty allowlist disables OIDC. Use `swarmfile ci oidc --audience …`; the job token itself never reaches a step. Mints are capped at 10 per job. `--aws-role <arn>` also writes the AWS web-identity env (`AWS_ROLE_ARN`, a 0600 `AWS_WEB_IDENTITY_TOKEN_FILE`, `AWS_ROLE_SESSION_NAME`) into `$GITHUB_ENV` (or `--export-env FILE`), so the AWS SDK/CLI assumes the role itself and the token is never printed. `--gcp <provider-resource>` writes a Google external-account credentials file and `GOOGLE_APPLICATION_CREDENTIALS`; `--azure --azure-client-id … --azure-tenant-id …` writes the Azure workload-identity env. A job that resolved an environment also carries the additive `environment` and `approval` claims, so a provider can require a production, approved gate.
- `write: [mr]` opens or updates a merge request from the job's pinned branch when every step succeeds. It does not bypass protected-branch rules, force a head, or author as the triggering person; the CI identity is the author and the run is provenance.
- `write: [commit]` commits a `mount: write` workspace as one CI-authored commit on the run's branch (see Writable workspace and CI commits). It never forces a head and never authors as a person.
- `write: [release]` publishes a release for the job's `release: <tag>` through the same pipeline as a manual publish (a public repository, the publish preflight, the publicignore/clone policy, the manifest and the archive build). The tag must already exist — a `write: [tag]` job in the same run, or an earlier tag — the job must run on an effectively protected branch, and the release is authored by the CI identity. The published release carries a signed provenance statement: `GET /orgs/<org>/repositories/<repository>/releases/<id>/provenance` (repository read access) returns the statement (workflow/run/job, commit, actor, builder image, attached artifacts), its RS256 signature and the public JWK it was signed with (stored per statement, so a key rotation never orphans older releases). Verify it offline with `scripts/verify-release-provenance.mjs` (pin the key out of band for a stronger claim); the claim is "signed by the hub, verifiable offline", and file digests live in the release manifest. The tag must point at the run's own pinned commit and branch, and a release needs a public repository, so a member cannot turn a protected-branch run into a release of another tree.
- A release attaches the job's artifacts as release assets: `release: { tag: v1, attach: all }` (or a list of artifact names) copies each stored run artifact into the release's asset store and records its size and SHA-256 in the signed statement (`attachments`). Limits: 20 artifacts, 1 GiB each, 2 GiB per release; a retry with identical bytes is idempotent, and a different asset with the same name is refused (`attachment_conflict`).
- Every CI write beyond a release (commit, tag, merge request) also records a signed **job attestation**. A member can fetch it from `GET /orgs/<org>/repositories/<repository>/ci/jobs/<job>/attestation` (add `?kind=commit|tag|mr` for one), and verify the JSON offline with `swarmfile ci provenance <file.json>` (or `--pin-key <jwk.json>` to pin the key). The statement names the run/job/workflow, commit/seq/branch, the CI actor and the triggering human, the builder image, and the write-specific subject (commit seq, tag, or MR id/number). A self-hosted runner must be updated to an engine that supports job-spec version 4 (the hosted images are current); an older runner fails the job with `unsupported job spec version`.
- `oidc` and `mr` are only minted for an effectively protected branch (and the run must be allowed there); `commit` is the opposite — it requires an unprotected branch and a job on an unprotected branch also needs the repository's `allowUnprotectedCi` setting on.
- A deployment that has not enabled CI write tokens fails at preparation (`ci_write_disabled`), and the commit/tag/MR routes re-check the switch per request.
- The organization must allow the audience before the job runs (`/orgs/:org/org-settings/ci-oidc-audiences`); a `403 ci_oidc_audience_not_allowed` means the audience is missing or the allowlist is empty.

Runner images

`image:` selects the container a job runs in. Omit it and the job gets `swarmfile-standard` (Debian bookworm with git, git-lfs, Node.js 22 LTS, Python 3 and a C/C++ toolchain). The curated catalog is available on every plan:

Image	Includes
`swarmfile-standard` (default)	git, git-lfs, Node.js 22 LTS, Python 3, curl, build-essential
`swarmfile-node`	the standard image plus pnpm and Yarn
`swarmfile-python`	the standard image plus Python pip and venv
`swarmfile-toolbelt`	the standard image plus jq, zip/unzip, rsync, cmake and sqlite3

The short form (`standard`, `node`, `python`, `toolbelt`) and the canonical `swarmfile-…` form are the same image; pick one and stay consistent.

Bring your own image (Pro and above)

An organization can register its own runner images under **Settings → Hosted CI → Runner images**. A registration is a **name** (what the workflow writes in `image:`) mapped to a **digest-pinned reference** in our managed registry:

`registry.cloudflare.com/<account>/swarmfile-runner/<organization-id>/<repository>@sha256:<64 hex>`

The organization-id segment is mandatory: it keeps one organization from selecting another organization's image. A tag, a reference outside `registry.cloudflare.com`, a missing digest, or a repository path that can escape its namespace is refused with `ci_image_ref_invalid`.

An organization image may deliberately shadow a catalog name — including the default, so an image named `swarmfile-standard` (or `standard`) is what jobs that omit `image:` use; both the short and canonical spellings shadow together.

A downgrade or a deployment that cannot select an image per start falls back to the catalog default for jobs that never named an image. Re-pointing an existing name (the **Edit** action) changes what every job selecting that name runs — it does not create a second row.

Importing an external image. Build a source image with Node.js at `/usr/bin/node` or `/usr/local/bin/node`, uid 1001 (`runner`) and a writable `/home/runner`; the runner base (`Dockerfile.runner-base`) is the intended starting point.

The base must also carry the engine's runtime libraries (at minimum `libssl3`, plus FUSE for a `mount:` job); the layer contract checks Node/uid/paths and the size, not arbitrary shared libraries, so a minimal image that satisfies the contract can still fail the job at runtime.

Push it to a registry and use **Settings → Hosted CI → Runner images → Import image** with its digest-pinned reference (a public registry, or a private one with its username/password in the same form). Swarmfile copies it into the managed registry, appends the runner agent/engine layer, verifies the layer contract and size, records the digest and prepares it; the source image is never executed.

The source may be a public image, the catalog, or this organization's own `swarmfile-runner/<organization-id>/` namespace; another organization's managed image is refused.

Building from a Dockerfile. An organization on Pro or above can skip the registry round-trip and build the image with the hosted builder. In **Settings → Hosted CI → Runner images**, choose **Build from Dockerfile** and attach a packed context (`.tgz`), or use the CLI:

```
swarmfile ci image build our-rust --context . --dockerfile docker/Dockerfile --wait
```

The CLI packs the context (default `.`) as a gzipped tar with `.git` excluded (64 MiB compressed cap) and pins it to the checkout's HEAD — a dirty working tree is refused unless you pass `--allow-dirty` — then uploads it to a trusted builder container.

The builder holds no registry credential: its Trivy pre-scan is advisory, it exports the image to the trusted import pipeline, which appends the runner agent/engine layer, scans the exact final image with a fresh database and applies the organization's waivers for that image name. A blocked scan is shown with its findings; the owner waives the CVEs and retries (or re-scans), and only then does the image prepare. `--wait` also runs the prepare step.

A secret-shaped `--build-arg` key is refused (secrets are not accepted as build args). `--secret NAME` (org CI secrets) is operator-gated and disabled by default; when a deployment enables it, `--secret` resolves server-side, is redacted from the build log, and is consumed with `RUN --mount=type=secret,id=NAME`.

The layer cache is not available in the trusted-import shape (`--cache` is refused with `ci_build_cache_unavailable`). `swarmfile ci image build-status|build-logs|build-cancel <id>` follow or stop a build; once the image import has started, the build can no longer be canceled. Builds are owner/admin, one at a time per organization and capped per day. Hosted builds are enabled in staging and production (`CI_IMAGE_BUILD=on`); an environment with the flag off answers `ci_build_disabled`.

Egress boundary. The importer resolves the source host and runs every fetch through a per-import loopback HTTP/CONNECT proxy; a source that resolves (or redirects) to a private, loopback, link-local or CGNAT address is refused before any pull. This relies on `crane` honoring `HTTP(S)_PROXY`.

External-source imports are **fail-closed** until the operator enables the scanned path, and a passing vulnerability scan is mandatory: every Critical finding and every High with a fix must be waived by an owner or admin, or fixed, before the image can be imported. The scan reads the image without executing it. The first-party catalog and the organization's own managed namespace are exempt.

Settings → Hosted CI → Runner images → Scan shows the latest findings; an owner or admin can grant a time-boxed waiver there (1–30 days) and retry the import, or waive and re-scan an already imported image. Waiver grants and revocations are written to the org audit log on a best-effort basis — a failed audit write is logged and does not block the waiver.

Scanning is gated like the rest of Hosted CI: when the operator has not launched CI for the organization the scan routes answer 404 `ci_not_available`, and each organization has a daily scan budget that manual re-scans, imported-image scans and the daily re-scan all draw from. A manual scan over that budget is refused with 429 `ci_image_scan_quota` (the scan never starts); the daily pass simply skips that run. Only owner and admin roles may start a scan or grant/revoke a waiver.

Building from a Dockerfile is a first-class path where the environment sets `CI_IMAGE_BUILD=on` (Settings → Hosted CI → Runner images → Build from Dockerfile, or `swarmfile ci image build`); the managed-registry push credential stays server-side, so you never handle one. An environment with the flag off answers `ci_build_disabled`.

A re-point updates the name first and then prepares the new digest; if preparation fails the name still points at the new reference and jobs selecting it are refused until it is prepared. A re-pointed catalog or this organization's own managed digest is exempt from scanning, while any scan recorded for the previous digest is stale — the Scan panel says so and does not treat its findings or waivers as current.

The image contract. A compatible source image is `linux/amd64`, runs as uid 1001 (`runner`), has Node at `/usr/bin/node` or `/usr/local/bin/node`, and a writable home. It must be glibc-based (the appended engine links glibc/OpenSSL; a musl/Alpine base will not run) — base built images on the runner base or another Debian-compatible Node image. The importer appends the agent at `/opt/swarmfile/job-agent.mjs`, the engine at `/usr/local/bin/swarmfile-engine`, and the launcher. The final image listens on port 8080; Hosted CI overrides the command with `node /opt/swarmfile/job-agent.mjs`, so you do not have to set a CMD.

The settings surface shows each image's size, preparation state and the organization's storage use, and prepares a saved or re-pointed image automatically (the CLI/tray show the same state). A job never starts an unprepared digest: admission refuses it with `ci_image_not_prepared`, so the first run cannot silently wait out its timeout.

Using an org image requires the Pro plan or above — a job on a lower plan is refused with `ci_byo_image_requires_plan`, but existing images stay listed and deletable. An unknown name refuses with `ci_image_unknown`; a pull that never becomes ready within the 10-minute delivery window fails with `ci_image_pull_failed`; an agent that answers the contract incorrectly fails with `ci_image_contract_invalid`; a start the platform rejects for size fails with `ci_image_too_large`. A run's job row records the exact digest that actually started (catalog map digest or org reference), so history remains attributable even after a name is re-pointed.

Environments and approvals

An organization defines **environments** (Settings → Hosted CI → Environments): a name, an optional URL, and a reviewer rule — zero or more reviewers as org members and/or owner/admin/member roles. A job binds to one with `environment: <name>`; if the environment names reviewers, the job **waits for an approval** once its `needs` are satisfied.

Nothing runs and nothing is billed while it waits; a reviewer (any one of the named users, or a member whose role satisfies a named role — `role:admin` admits owners too) approves or denies it from the run page, and the bell notifies every eligible reviewer. Denial fails the job `environment_denied` and skips its dependents; approval queues it immediately.

A job may also declare `approval: true` beside `environment:` — that adds a gate when the environment names no reviewers (owners and admins then approve). A declaration can only add protection: it never broadens or removes the environment's reviewer set. Each environment can also turn on **"the run's starter cannot self-approve"**, which refuses the person who triggered the run even if they are a listed reviewer.

A job naming an environment the org has not defined is blocked with `ci_environment_unknown`; retrying a failed gated job asks for approval again rather than starting unapproved. Reviewer changes apply to requests already waiting: removing a reviewer stops that person deciding a pending request, and clearing the reviewer list (with no `approval:` on the job) un-gates it — any member can then advance it from the run page. Deleting an environment leaves a pending request's reviewer set in place, so it can still be decided.

Reviewer user ids are checked against org membership when the environment is saved.

An environment also has its own **secrets and variables scope** (Settings → Hosted CI → Environments → **Secrets & variables** on the environment's row): a job that declares the environment resolves those on top of the repository and org scopes, and the environment wins on a name collision — the natural place for deploy credentials that must not be readable by a job running on a pull request.

Secrets and variables

Repository secrets and variables live in the repository's **CI** tab; org-scoped ones are available to every repository and live in the dashboard's **Settings → Hosted CI**, together with the org's environments (each with its own secret/variable scope), its custom runner sizes, its runner images and its CI policy — the org-level enable/disable control and the **Force unprotected runs off** override. All of these are owner/admin-managed.

Resolution precedence is environment (only for a job declaring it) → repository → org: the most specific scope wins on a name collision.

- Secret **values are write-only**: they are never shown again, in any surface.
- A job must list the secret names it uses in `secrets:`. Referencing an undeclared secret fails the workflow before the job starts.
- Org secrets are only resolvable from **private** repositories (an environment's secrets inherit this: a public repository's workflows see repository-scope values only).
- End-to-end encrypted repositories do not run **hosted** CI: a hosted runner holds no E2E key by design. A self-hosted runner serves them from the owner-maintained plaintext workflow sidecar, with the repository key delivered to a granted runner's device.

Artifacts and dependency caches

A job can publish files for later jobs, and restore/save a dependency cache between runs:

```
jobs:
  build:
    artifacts:
      - name: dist
        path: packages/app/dist
        if-no-files-found: error   # warn (default) | error | ignore
    cache:
      key: deps-${{ ci.branch }}-${{ ci.sha }}
      paths: [.npm]
      restore-keys: ["deps-${{ ci.branch }}-"]
    steps:
      - run: npm ci
      - run: npm run build

  deploy:
    needs: [build]
    downloads: [dist]              # fetched after checkout, before steps
    steps:
      - run: ./packages/app/dist/deploy.sh
```

- **Artifacts** are bundled (tar.gz) at job end — on success or failure, or gated by the entry's `if:` — and `downloads:` fetches each name from the newest successful producer in the job's transitive `needs` closure (a missing producer fails the job). Limits: 1 GiB per artifact, 5 GiB per run, 30-day retention; a `path` that climbs out of the workspace is refused.
- **Caches** restore after checkout and before the first step: the exact `key`, then the newest entry matching any `restore-keys` prefix, first in the job's own branch and then in the default branch (a branch can restore the default branch's toolchain cache). The save happens only when the job succeeds and always writes to the job's own branch. An organization owner may additionally open an **org-shared cache** (Settings → Hosted CI): `read` lets repositories restore from it, `write` also lets them publish. Sharing is org-wide and a `write` repository can influence another repository's restore for the same key — enable it only for repositories you trust with each other. Key supports `${{ }}` (no secrets). Limit: 5 GiB per entry. Repositories have a 20 GiB cache budget and organizations 100 GiB; when a save would pass the repository budget the least-recently-used entries are evicted, and a save that cannot fit at all is refused `ci_cache_quota_exceeded`. Entries expire after 30 days (the daily sweep also enforces the org budget). The run view's job table shows `hit` or `miss` per job, and `ci status` prints the same.
- The GitHub importer maps `actions/upload-artifact`, `actions/download-artifact` and `actions/cache` (or the split restore/save pair) onto these fields when their arguments are literal.

Services

Integration tests often need a database or broker. A native job can declare `services:` — processes started inside the **same** job container before the first step, on a loopback port, and always stopped with the job:

```
jobs:
  test:
    services:
      cache:
        run: redis-server --port 6379
        port: 6379                          # readiness: tcp
      api:
        # A service runs in its own empty directory, not the workspace: give it
        # an installed command or inline code, not a repo-relative script path.
        run: node -e "require('http').createServer((q,s)=>s.end('ok')).listen(8080)"
        url: http://127.0.0.1:8080/health     # readiness: http
        timeout: 90
    steps:
      - run: npm test
```

- `readiness` is `tcp` (with `port`), `http` (with a loopback `url`), `exec` (with a `ready` shell command or a `command:` argv, re-run until it exits 0) or `alive` (a settle delay); the default is derived from the fields present. An `http` probe may send `headers:` (for example an `authorization` value the health endpoint requires); `command:` runs its argv directly, without a shell, and cannot be combined with `ready:`. A service that is not ready in time — or exits first — fails the job before its first step. Service output appears in the log as `[service:<name>/out] …` (stdout) and `[service:<name>/err] …` (stderr).
- `needs:` (a name or a list) starts a service only after the named services are **ready**, and the compiler refuses an unknown reference, a self-need or a dependency cycle. Services in independent branches still start in parallel.
- `restart: on-failure` restarts a service whose process exits non-zero up to `max_restarts` times (0–5, default 2; `never` is the default). The restart bound is enforced before readiness and after it; when the bound is reached the service is left stopped and the job's steps see the failure. The job's own `timeout` still bounds the whole run — there is no unbounded health loop.
- A service is repository code run as the job's uid with the job's public `CI_*` variables and its own `env:` — never the job token or a secret value. It gets `SWARMFILE_SERVICE_<NAME>_HOST` and, when configured, `_PORT` and `_URL`. It runs in a directory **outside** the workspace, and `services:` is refused together with a `mount: write` job (a service could otherwise write the commit). It is killed with the job and never outlives it.
- `image:` on a service is refused: the hosted platform runs one container per job, so a service that needs its own image (Postgres, Kafka) is out of scope — use `image:` on the JOB (an org image that already carries the process) or start it in a `run:` step. On a self-hosted runner a service is a host process, so it must already be installed there; naming a service image is refused everywhere.
- At most 5 services per job. Declaring one moves the job spec to version 3, so a runner image that predates services refuses the job instead of running it without them; a depth field (`needs`, `command`, `headers`, `restart`) moves it to version 6, so a runner that would silently ignore the ordering or the probe refuses instead.

Migrating from GitHub Actions

`swarmfile ci import` and the web editor's import translate the run-centric subset of a workflow. A small, pinned set of `setup-*` steps is translated to a native **verification** step rather than fetched: `actions/setup-node@v3` / `@v4` and `actions/setup-python@v4` / `@v5` become a check that the selected `image:` has the requested major (`20`) or major.minor (`3.12`).

If the image's toolchain does not match, the step fails naming the image instead of downloading another toolchain; `with.cache` is refused (use the job-level `cache:` field). Quote a version with a trailing zero (`"3.10"`) so YAML does not read it as `3.1`. Every other marketplace action stays refused and is never executed.

Runner sizes, billing and limits

Size	vCPU	Memory	Disk
`small`	0.25	1 GiB	4 GB
`medium` (default)	1	6 GiB	12 GB
`large`	2	8 GiB	16 GB
`xlarge`	4	12 GiB	20 GB

Orgs can define custom sizes (quarter-vCPU steps, up to the platform ceilings); each custom size rounds **up** to the smallest capacity tier that covers it (the four sizes above, plus a 0.5 vCPU / 4 GiB / 8 GB tier between `small` and `medium`), and is billed at that tier's per-second rate.

Sizes are org-wide; manage them in **Settings → Hosted CI** or with `swarmfile ci size` (the repository CI tab lists them read-only, with rates). Define one with `swarmfile ci size set build-heavy --vcpu 2 --memory 8 --disk 16` (or the size form in Settings → Hosted CI), then ask for it by name in a job: `size: build-heavy`.

A self-hosted selector is `runs-on: [self-hosted, <label>...]`. It is accepted while the organization's policy is active (on by default; an owner can disable it), then queues for a registered runner matching the labels (and whose dependency frontier is released); a job with no matching live runner expires after the 2-hour queue TTL. Otherwise the job is refused with `self_hosted_runners_not_enabled`. The runner is a separate release and operator setup, not something the hosted service runs for you — see Self-Hosted CI Runners.

Billing is per second of job time at the size's rate (at-cost × 1.2), rounded up to the next whole second (a started job bills at least ten seconds). The run list shows each job's billed seconds and charge. Compute counts against the org's spend cap — over the cap, a new job is refused with a blocked `spend_cap_reached` run and reads keep working.

Every org except Enterprise needs an active, trialing or past-due subscription before its first job (otherwise the run is refused with `payment_method_required`): paid and trialing Starter and Pro orgs already pass, while a Free org (or one on a Verified-OSS grant) adds a card in **Billing** first. Enterprise is invoiced and exempt.

A Free org's compute runs under a Free-plan compute ceiling that owners can't change: it scales with account standing (the org's age, and whether its creator has a verified email, a verified card or a seat in a paid org), from about $2.50 up to $15 of compute a month. Platform ceilings apply.

What a job costs

Rates are the provider's list price for each size, recovered at cost × 1.2, at current provider pricing:

Size	vCPU / memory / disk	Per minute	Per hour
`small`	0.25 / 1 GiB / 4 GB	$0.00056	$0.03
`medium` (default)	1 / 6 GiB / 12 GB	$0.00258	$0.15
`large`	2 / 8 GiB / 16 GB	$0.00440	$0.26
`xlarge`	4 / 12 GiB / 20 GB	$0.00802	$0.48

A job bills `ceil(elapsed)` seconds at its resolved size's rate; the run list shows each job's billed seconds and exact charge. If provider pricing moves, this table and the in-app rates move with it.

How this compares to GitHub-hosted runners. Our billing is per second, with no included-minute allowance on any plan. GitHub's is per minute, with partial minutes rounded up, and its private-repo plans include a monthly minute allowance (public-repo jobs are free) — below that allowance, staying on GitHub can be cheaper, and we do not match free public-repo CI.

This comparison is for paid CI. Past it, the comparable Linux job costs less here: our `large` (2 vCPU / 8 GiB) is $0.00440/min, while GitHub's published 2-core rate works out about 35% higher per paid minute (we're about a quarter cheaper per minute at the quoted $0.006 list). Short jobs save more, because we bill per second with a 10-second minimum and GitHub rounds up to the minute: a 5m30s job bills 330 seconds here instead of six minutes there.

GitHub's larger, macOS and GPU runners are premium-priced and aren't offered on hosted CI; those jobs belong on hardware you run, through Self-Hosted CI Runners when the organization allows them. Verify GitHub's current rates at their pricing calculator, and see the dated side-by-side on the GitHub comparison page.

Job limits

Other limits: a job's `size` must resolve (an unknown name fails the job rather than guessing); a job's log is capped at 50 MiB and a run's log at 200 MiB (output past a cap is dropped and the log is marked truncated — the earliest output is what is kept) and logs are retained for 30 days — organization owners can shorten that per repository to 1–29 days; concurrent jobs are capped by plan (Free 1, Starter 2, Pro 8, Enterprise 16, weighted by size: `small` counts 0.25, `medium` 1, `large` 2, `xlarge` 4).

A size heavier than the plan's whole budget can never run, so Free can't run `large` or `xlarge` and Starter can't run `xlarge`; such a job is refused with `ci_size_exceeds_plan_budget`.

Public repositories and open source

Hosted CI is a paid compute service on public repositories too. There is no free public-repo tier and no included minute allowance: a public repository needs a card like any other, bills at the same per-second rate, and its runs and logs stay member-only. A Verified-OSS grant covers the Pro plan, not compute.

Use it when the repository is large enough that streaming a job's declared paths beats a clone, or when a self-hosted runner already has the bytes. If you need free CI for an open-source project, keep GitHub Actions for that. The open-source story here is the community server, the standalone tools and the public repository surface — not free CI compute, and the use of a GitHub-shaped workflow file does not make this an Actions marketplace.

Security

A workflow can read the repository's secrets, so hosted CI treats it like code that runs with credentials:

- Automatic runs come only from branches with **effective protection** (a protected branch rule) — a push to an unprotected branch is refused with a visible `blocked` run.
- A repository owner can opt into unprotected branches (the **Allow unprotected runs** switch in the CI tab, `swarmfile ci enable --allow-unprotected`). Only do this when everyone who can push to those branches is trusted like a branch protector.
- A blocked run still writes its check row as a failure, so a required check can never be skipped by breaking CI.
- Job tokens are scoped to one job, one repository and the triggering ref, expire with the job, and are stripped from every step's environment; the runner also blocks a step from reading the engine's `/proc/<pid>/environ` and masks the token from logs.
- A **bring-your-own image is trusted organization code**: it contains the agent that receives the job spec, so it can read the job token and every secret the job is allowed to read. Register only images you or your security team control; prefer digest pinning and re-point deliberately.

What is not supported

The dialect is small on purpose and fails closed: anything it does not understand refuses the workflow with a line number.

- No marketplace/remote `uses:` (never arbitrary code); a LOCAL composite action under `./.swarmfile/actions/<name>` IS supported (see Composite actions).
- No remote reusable workflows; same-repo `uses: ./.swarmfile/workflows/<name>.yml` IS supported (see Reusable workflows), including `workflow_call` inputs. No `permissions`, and no service *containers* — in-container `services:` ARE supported (see Services). `environment:` + `approval:` ARE supported (see above). Artifacts and dependency caches are native (see below).
- No `container:` (GitHub's per-job container) — use `image:` instead, which selects one of the curated runner images or an org-registered one; `size:` selects resources independently.
- `workflow_dispatch` inputs ARE supported (typed `string` / `boolean` / `number` / `choice`; see Manual inputs). A manual run otherwise takes the branch only.
- No merge queues, no fork PRs, no `pull_request_target`.
- Self-hosted labels require the organization's policy to be active (on by default; an owner can disable it). `runs-on: [self-hosted, ...]` queues for a registered runner matching the labels (or expires after the 2-hour queue TTL), and is refused with `self_hosted_runners_not_enabled` when the org policy is off. See Self-Hosted CI Runners; the branch-watching runner as CI is a separate tool, not a hosted-CI runner.

Going further

- `swarmfile ci list|status|logs|cancel|run` and the secret/variable/size commands: see the CLI reference.
- The desktop tray shows the same runs, logs and settings per repository.

Self-Hosted CI Runners

A self-hosted runner runs the workflows in `.swarmfile/ci.yml` on a machine your organization controls instead of on Swarmfile's containers. The job protocol, secrets, logs and checks are the same as Hosted CI; what changes is where the code executes and who holds the machine.

Self-hosted runners are **active by default**. An **organization owner** can turn them off in **Settings → Hosted CI → Policy** (the machines receive the org's CI secrets and, when opted in, repository keys, so an org that does not trust its teams' hardware can disable the surface). While enabled, an owner registers a runner for the organization and the runner claims jobs from the repositories on its allowlist. While an owner has disabled the policy, a self-hosted job is refused at admission with the visible `self_hosted_runners_not_enabled` reason instead of waiting in the queue; hosted jobs in the same workflow still run.

Register a runner

1. Open **Settings → Hosted CI → Self-hosted runners** (or the tray's **Repository settings → Hosted CI** card).
2. Choose **New runner**, give it a lower-case name (`linux-builder-1`), and pick the labels it advertises. A job's `runs-on` list must be a subset of these labels for the runner to claim it.
3. Scope the runner to at least one project. Self-hosted runners are project-scoped: the allowlist must name the projects the runner may serve, and a grant is refused for a project outside it.
4. Choose whether the runner may serve **managed** repositories (see Encryption tiers).
5. Copy the one-time `sf_shr_…` token — it is shown once and stored hashed.

Start the runner with the token and your hub URL:

```
export SWARMFILE_HUB_URL=https://hub.swarmfile.com
export SWARMFILE_SELF_HOSTED_RUNNER_TOKEN=sf_shr_…
swarmfile-self-hosted-runner
```

The runner writes its lease to a 0600 state file (SYSTEM/Administrators/current user ACL on Windows), claims one job at a time, writes the rendered job spec to a private file, runs it through the sibling `swarmfile-engine` binary, and relays the status to the hub. Keep `swarmfile-engine` next to the runner (or set `SWARMFILE_ENGINE_BIN`). `SWARMFILE_SELF_HOSTED_RUNNER_WAIT_SECS` (default 20, bounded 0–25) is the claim long-poll window, and `SWARMFILE_SELF_HOSTED_RUNNER_POLL_SECS` (default 10) is the poll interval used when it is 0. `SWARMFILE_SELF_HOSTED_RUNNER_ONCE` exits after one poll.

The package builder (`scripts/ci/build-self-hosted-runner-package.sh` , run by the hermetic packaging lane) stages the runner, its sibling engine, the installer and a SHA256 sidecar for a platform.

Release packages are built by `.github/workflows/self-hosted-runner-packages.yml` ; every published release attaches the Linux and Windows packages automatically, and the release Mac builds/attaches the macOS package (no GitHub-hosted macOS runners). Each package carries its own `.sha256` sidecar; the release's `SHA256SUMS` is generated before these assets attach, so verify the sidecar.

The package is a separate operator artifact, not part of the desktop install, and the deployment must also have self-hosted runners enabled for the organization (on by default; an owner can disable it). On Linux and macOS the package's `install.sh` registers a systemd service or launchd daemon; on Windows `install-self-hosted-runner.ps1` registers a scheduled task. See Windows below.

Fleet registration tokens

For an autoscaling fleet, create a **registration token** (`sf_srt_…`) instead of a single runner. A registration token is mint-only: each use mints one ephemeral runner that inherits the token's labels, project scope and encryption capability, with its own one-time credential and the token's expiry. `max-runners` caps live minted runners; revoking the registration token revokes every runner it minted and drops their sessions. Rotate the token by creating a new one and deleting the old.

Manage the fleet from the CLI (org owner/admin; the token is shown once and never stored):

```
swarmfile ci runner list
swarmfile ci runner queue
swarmfile ci runner create linux-builder --label linux --label x64 --repository <project-id>
swarmfile ci runner token fleet --label linux --repository <project-id> --max-runners 20 --ttl-hours 24
swarmfile ci runner tokens
swarmfile ci runner revoke-token <token-id>
swarmfile ci runner delete <runner-id>
```

`ci runner create` prints the `sf_shr_…` enrollment token; `ci runner token` prints the mint-only `sf_srt_…` token. Both are hashed at rest and cannot be shown again. A runner or registration token may allow at most 25 repositories; enrollment is unlimited, while live sessions are capped at 64 per org and plan concurrency still applies. `ci runner queue` reports this repository's per-label queued/running demand; the tray's CI card shows the same queue.

Labels and `runs-on`

A self-hosted job is a normal workflow job with `runs-on` :

```
jobs:
  test:
    runs-on: [self-hosted, linux, x64]
    steps:
      - uses: actions/checkout@v4
      - run: cargo test
```

`runs-on:` and `size:` are mutually exclusive: use `runs-on: [self-hosted, ...]` (not `size: self-hosted`) for a self-hosted job, and hosted sizes are never served by a runner. A job is claimable only when its `needs` have released it (the same dependency frontier hosted jobs obey; an `if: always()` / `failure()` dependent waits for its needs and then starts). If no live runner matches, the job stays queued for up to **2 hours**, then the queue TTL expires it; cancel the run to stop it sooner.

Security model

- **Protected branches only.** The claim route serves a job only from a branch with effective protection (or a branch protected by a rule); an unprotected branch is refused even when the job was dispatched manually.
- **No bucket credentials.** Cache, artifact and log bytes go through the hub routes with the job token. A runner never receives object-storage credentials.
- **One-time credentials.** A runner token is single-use; the minted lease is fenced, expires, and is revoked when the runner is deleted or its registration token is revoked.
- **Manual approval is honored.** A job with a pending approval is never claimed.

Canceling a run terminates the in-flight job process on the runner (SIGTERM on POSIX; the child process is terminated on Windows) and its checks are updated on the connected GitHub repository when the GitHub App is installed.

Encryption tiers

A self-hosted runner serves **plaintext** repositories by default. Choosing **Allow managed repositories** when registering (or on the registration token) lets it claim managed jobs too: the managed key is delivered to the job spec on your hardware, so only opt in on machines you trust with the organization's managed keys.

End-to-end encrypted repositories need two extra opt-ins. Creating a runner with **Allow end-to-end encrypted repositories** makes it a key holder: on its first start it enrolls an X25519 device key under the owner who created it (the private key never leaves the machine), and the existing wrap-on-grant pipeline delivers the repository key. Then, in the repository's **CI** tab, grant that runner under **End-to-end encrypted CI**. The key is delivered to the granted runner on the claim, never in the job spec. Revoking the runner revokes its device key and cuts every wrap.

Because the hub cannot read an encrypted tree, the workflow definition for an E2E repository lives in the **plaintext workflow sidecar** in the CI tab instead of `.swarmfile/ci.yml`. It is compiled with the same parser; local composite actions and reusable workflows are refused there, since their sources are encrypted tree files.

Retries

A job that reports `infra_failed` **before it started** (a runner that could not build or start the workspace) is retried automatically up to two times: the attempt is bumped, the log store reset and the run re-queued.

A job that ever reported `running` stays terminal for a human retry, and a runner that goes silent is reported `runner_lost` — never silently re-queued.

A restarted runner that polls with the lease of a job it already started is stopped the same way: the job is never executed twice, and the previous attempt's log, artifact and cache writes are refused once a retry has moved the job on.

A skipped job reports `skipped`, a job-deadline timeout reports `timed_out`, and a run canceled by a member reports `cancelled`, matching hosted jobs.

Checks and GitHub

When the organization's GitHub App installation covers the repository and the run's commit is the branch's latest pushed commit, job state is mirrored to a GitHub check run named `<workflow> / <job>`: in progress on start, success/failure/canceled on completion. Mirroring is best effort: a mirroring failure never blocks a job.

Billing

Self-hosted jobs are not metered for compute. The organization pays its own infrastructure; the hub records the spend as zero for the run.

Linux and macOS

```
sudo ./install.sh --hub-url https://hub.swarmfile.com --token sf_shr_...
```

The installer copies the runner and engine to `/opt/swarmfile/self-hosted-runner` (Linux) or `/usr/local/swarmfile/self-hosted-runner` (macOS), writes a 0600 environment file with the token, and registers a service that restarts on failure. `--dry-run` prints the plan, `--uninstall` removes the service, and `--uninstall --purge` also removes the cache/state directory.

Windows

The PowerShell installer registers a scheduled task and stores the configuration under `%ProgramData%\Swarmfile\self-hosted-runner` with a restricted ACL. Run it from the unpacked package (the `.exe` files sit beside it):

```
.\install-self-hosted-runner.ps1 `
  -HubUrl https://hub.swarmfile.com `
  -Token sf_shr_… `
  -InstallDir 'C:\ProgramData\Swarmfile\self-hosted-runner'
```

`-DryRun` prints the plan, `-Uninstall` removes the task and directory. Re-run the installer with a new token to rotate credentials.

Limitations

- Linux, Windows and macOS (the hermetic lane, including the E2E key ceremony) are exercised end to end; a real macOS mount lane is not automated.
- A runner claims one job at a time; run several runners for parallelism.
- `services:` and `container:` steps run on hosted infrastructure only.
- E2E workflows cannot use local composite actions or reusable workflows; the sidecar must be self-contained.
- Shared-cache reads and writes follow the organization's `off` / `read` / `write` policy (server-enforced; a job token cannot widen it).

Automating Swarmfile

Swarmfile is built to run without a person at a keyboard: a render-farm node, a CI job, a coding agent, or a seed box can mount, sync, and report on a repository on its own. This page is the map: which credential to reach for, the surfaces that are meant to be scripted, and the REST API contract.

The credentials

You need to	Use	Scope
Run an engine where nobody can complete a browser sign-in (render node, CI runner, agent fleet, seed box)	**Repository API key** (`sf_key_…`)	Exactly one repository
Run an untrusted agent or a least-privilege job on one branch (optionally only some paths)	**Branch- or path-scoped repository API key** (`sf_key_…`)	One branch; optional allow/deny globs (path-scoped keys are engine/API-only)
Act as yourself from a script — provisioning, webhook management, anything your own role can reach	**Personal access token** (`sf_pat_…`)	Whatever your account can do, live
Push and pull large objects with the stock `git-lfs` client	A **git-LFS credential** — a repository API key minted from the repository's git-LFS tab	That repository's LFS traffic
Work at a keyboard	Normal sign-in	—

Repository API keys are org-owned machine identities. A key is walled off to one repository: it cannot read or write anything else in the org, and it is never treated as an owner or admin, no matter who minted it.

Mint, list, revoke, and set a per-key request ceiling with `swarmfile api-key` (admin or owner), or from the dashboard's **API keys** page. The dashboard page also gives a key its own monthly **spend budget** — a dollar ceiling below the org's spend cap; enforcement uses the lower of the two, and a key over its budget has only its own writes paused. See CLI: api-key.

A key can be **scoped** before it leaves the dashboard. `--branch-scope <branch>` binds it to one non-default branch and keeps git, history and merge requests on that branch; adding repeatable `--path-glob` / `--deny-glob` makes it **path-scoped**, an engine/API credential with no git, history, commit walk or merge requests; it can use the content-addressed batch checkpoint and the mount surface, not the branch archive or inline batch content.

The scope is set at mint time — to change it, revoke the key and mint a new one — and each allow/deny list holds at most 64 globs of at most 1,024 characters.

The branch must already exist and be non-default; minting does not validate it, and the hub refuses a dead or default branch on first use. The dashboard's **API keys** page and the tray's API keys panel use the same fields. See Branch- and path-scoped keys.

Personal access tokens are not a separate identity — a PAT resolves to you, and every access check runs against your own current role, so it can never do more than you can right now and it dies with your access, with nothing separate to clean up. Mint one under **Settings → Access Tokens** (self-serve for any member; up to 20 active per member, per org, optionally expiring). See Personal access tokens.

A **Starter+** service delegation can mint a short-lived branch-scoped key from a service user's PAT; a glob-free delegation also hydrates a workspace through the branch archive. See Service-minted agent tokens and the branch archive.

Both are shown in full exactly once at creation, both are revocable with immediate effect, and both are audited. Two more rules worth knowing:

- On a machine with both an API key and a leftover `SWARMFILE_OIDC_REFRESH_TOKEN`, the API key wins — and the engine logs the choice rather than picking quietly.
- If your org enforces an authentication policy, repository API keys are exempt (they're org-provisioned machine identities), while a PAT snapshots the factors and email assertion of the session that minted it — a token minted before you enrolled stops satisfying the policy when enforcement begins.

Running an engine headlessly

Mint a key for the repository it serves, then hand it to the engine as `SWARMFILE_API_KEY`:

```
# On a machine where you're an org admin/owner:
swarmfile api-key create render-farm-04 --repository proj_9f2a

# A scoped key for an agent that should touch only one branch:
swarmfile api-key create agent-04 --repository proj_9f2a --branch-scope agent/worker-04 --path-glob 'src/**'

# On the headless machine:
export SWARMFILE_API_KEY=sf_key_...
export SWARMFILE_ORG_ID=...
export SWARMFILE_REPOSITORY_ID=proj_9f2a
swarmfile-engine
```

Key minting is a hard plan allowance, not a soft one: Free 1, Starter 2 per seat (floor 5), Pro 5 per seat (floor 10), plus additional headless seats at $2/month each. A mint past the cap is refused with `keys_cap`; revoking frees the slot immediately, and a downgrade blocks new mints without revoking keys you already have. See Billing & Plans.

Pin a key's own rate limit when a farm node should burst above the plan default:

```
swarmfile api-key rate-limit <key-id> 1000
```

For the full per-agent recipe — a labeled mount per agent and a shared dependency cache — see Coding Agents.

Scripting the CLI

- `--json` (global) makes the CLI print the engine's raw response — the intended path for scripts — and a usage error comes back as `{"code":"usage_error",…}` instead of prose.
- **Exit codes**: `0` success, `1` error, `2` a refusal a script should act on (permission, quota, a conflict that needs a human), `3` a long operation detached. See the CLI reference.
- **Long operations** (`merge`, `commit`, `checkout`, `materialize`, `repository delete`, and more) wait and print their result by default. `--no-wait`, or Ctrl-C, returns the operation id (exit `3`) and the work carries on in the engine; collect the outcome later with `swarmfile op` — even after an engine restart.

CI

There are two shapes, and they compose:

- **Swarmfile's own runner** — a headless engine that watches a branch and runs a job on every matching commit or tag, from an in-place checkout with no clone and no FUSE. See Runner (Headless CI).
- **Your existing CI reporting a check** — any system that can make two authenticated HTTP calls can gate a protected branch's **require checks to pass** rule through `/runner-runs`, using the same repository API key a runner uses. The runner guide has the exact two-call contract: Reporting checks from your own CI system.

A headless engine authenticating with an API key also serves `git clone` and `git push` through the remote helper, once git access is enabled for the repository — see Cloning a Repository with Git.

Webhooks

Webhooks deliver a signed HTTP `POST` to a URL you control when something happens in the org — no polling. An **org-wide webhook** is owner-managed, retried, logged, and auto-disabled after repeated failure; a **personal webhook** is a fire-and-forget notification channel for your own account. Deliveries are dispatched when the event is recorded — typically within seconds; the hourly tick is a backstop for a missed wake.

The full envelope, signature verification, event kinds, and limits live in Webhooks. The automation-relevant part: an API key cannot manage webhooks or any other org-wide admin surface — that's deliberate. Manage them from a script with a personal access token instead, which acts as you.

Rate limits

Request budgets are per-user and per-org, counted in requests rather than bytes, and content streaming isn't counted. A PAT shares its owner's per-user window; an API key gets its own window at the plan's per-user default, adjustable per key.

How the org window scales

The org window scales with the fleet: it is the larger of the plan's per-org default and (active keys × the plan's per-key default), capped by the plan's per-org maximum — 10 Pro keys give 5,000/min, not 1,000 + 10 × 500. An owner/admin org override replaces the scaled value (scaling is off while it is set).

Enterprise automatic scaling stops at 50,000/min and has no key-allowance cap; on the other plans the headless-seat allowance (seats plus purchased seats) caps how many active keys count, so over-cap keys retained from a larger plan still work individually but do not enlarge the org window. A per-key rate limit changes that key's own bucket, not its contribution to the org window.

The engine batches and paces bursts against the budget the hub reports — burst behavior is covered in Coding Agents.

The REST API contract

The hub's REST API is published as OpenAPI in the open-source `swarmfile-api` repository (MIT/Apache-2.0):

- `hub-openapi.json`: the hosted hub's routes that have authored request and response schemas.
- `hub-core-profile-openapi.json`: the core-profile contract the self-hosted community server answers for. A few community routes are deliberately **outside** the core profile — the branch archive and the batch checkpoint — so they are absent from this file (and from its generated client) even though the server serves them.
- `hub-core-profile-client.ts`: a dependency-free TypeScript client generated from that profile, built on `fetch`.
- `webhooks.md`: the webhook envelope, signature verification and event kinds.

Figure omitted in print (animation): The community core profile is a contract: generated OpenAPI and a typed, dependency-free client ship in the open swarmfile-api repository. The cast clones that repository with the open git helper, shows the whole demo program, and runs it against a real community server — fetching the live profile and listing the repository's root over the HTTP API. - https://swarmfile.com/demo/oss-sdk.cast

On hosted and self-hosted Swarmfile, the branch archive is a streamed tar (`GET /orgs/:org/repositories/:repository/files/archive?branch=…`, with `paths`, `format=tar.gz`, `ETag` /304 and a manifest sibling; the folder form too); see Hydrating without a mount. Both scan at most 20,000 rows; the community server caps output at 256 MiB (hosted: 1 GiB) and limits archive starts to 30/minute per repository. A sandbox can write its side-effects back in one call with the batch checkpoint; the community server's batch tree cap is documented there.

Authentication and versioning

Requests to the hosted hub authenticate with `Authorization: Bearer` and a personal access token or an API key; a community server uses its own keys, minted by its admin command or with `swarmfile api-key create`. Every response carries `x-swarmfile-api-version: 1`, which changes only with a deliberate breaking change; a tagged release of `swarmfile-api` is the exact contract at that version.

What to script against

There's no multi-language SDK. Routes that aren't in the published OpenAPI and aren't named in these docs are internal and can change without notice, so script against the documented routes, the `swarmfile` CLI (with `--json`), headless engines authenticated by API keys, the `/runner-runs` check API, and webhooks.

The branch archive and its manifest are the documented streaming/metadata exception: they are in the hosted `hub-openapi.json` but deliberately **outside** the community core profile (`hub-core-profile-openapi.json`), so the core-profile client does not cover them; their contract is also the route manifest, the agent SDK and the branch-archive runbook. The batch checkpoint is the same kind of documented exception.

On the community server, an archive's `X-Swarmfile-Archive-Head` may be a provisional member claim rather than a confirmed commit hash; read `X-Swarmfile-Archive-Claim-Status` (`confirmed` / `provisional` / `unknown`) before treating the value as repository history.

Handling the credentials

- Copy a secret once and store it in a secret manager or your CI's secret store — never in the repo, an image layer, or shared shell history.
- Prefer an API key over a personal refresh token on shared machines; revoke a key the moment a node is retired.
- Keys are org-owned: deleting a repository revokes its keys with it, and every mint, revoke, and rate-limit change lands in the org's Activity feed.

Where to go next

- CLI: `api-key` — mint, list, revoke, and cap keys.
- Personal access tokens — the script credential that acts as you.
- Runner (Headless CI) — the runner, external check reporting, and ref materialization.
- Coding Agents — one mount per agent, a shared cache, service-minted tokens, the branch archive and the batch checkpoint.
- Webhooks — event kinds and signature verification.
- Billing & Plans — key allowances and packs.

Sharing & Collaboration

Swarmfile has three collaboration surfaces: tools for your own org members working inside a repository (comments, watch, presence), tools for getting a specific file or folder in front of someone who isn't a member at all (share links, external collaborators), and anonymous drops on `/send`. This page covers all three, plus how they interact.

Comments and mentions

Every file has its own comment thread. Post from the CLI with `swarmfile comment <path> -m "message"`, or list a file's thread with `swarmfile comments <path>`. The web dashboard has a full thread panel with `@`-mention autocomplete for tagging teammates.

Threads live-update in the dashboard — anyone with the file open sees new comments and mentions arrive without refreshing.

A comment can be a reply, not just a new top-level note — click **Reply** under any comment to keep a back-and-forth in one thread instead of scattering it across separate top-level posts. Replies are one level deep (no reply-to-a-reply); resolving is a thread-level action on the original comment, not on each reply individually.

On an image preview in the web dashboard, click **Add pinned comment**, then click a spot on the image to leave a comment anchored to that exact location — useful for pointing at a specific area of a render or drawing instead of describing it in words. Pinned comments show as small markers on the image and, everywhere else the thread appears (including the Desktop App), as a 📍 badge next to the comment. Pinning is web-only in this version; PDFs and other file types don't support it yet.

Right-click any file in the web dashboard's Files view and choose **Open comments** to jump straight to its thread, without navigating there through the file list first.

Watch

Watch is a targeted, opt-in subscription to a specific file or folder, distinct from the general "someone else changed a file" bell notification you get for activity across a repository. You choose exactly what you want notified about.

Watching a folder is automatically recursive — you don't pick recursive vs. non-recursive, the server derives it from whether you watched a file or a folder.

- `swarmfile watch <path>` — start watching a file or folder
- `swarmfile unwatch <path>` — stop
- `swarmfile watches` — list everything you're currently watching

Watch is also available outside the CLI: the Desktop App's file list has a per-row watch toggle, and the web dashboard exposes watch/unwatch from the right-click context menu on a file or folder. That same menu also has **Open comments** (above), **View history** (Trash, History & Rollback), **Request unlock** (Working with Files), and **Copy Swarmfile path** / **Copy path in drive** for grabbing a file's location without opening it.

A watched-file change reaches you through the notification bell and a Desktop App toast; how notifications are delivered, and which kinds can also be emailed, is covered in Notifications & Inbox.

Presence

Presence shows who's actively editing what, in real time. Run `swarmfile presence` (or the shorter alias `swarmfile who`) to see it from the command line, or check the dashboard for the same view. Presence is tied to the entry lock lifecycle, not a free-running heartbeat — see Working with Files for how locking and presence relate.

Share links

A share link publishes a single file or folder to anyone with the URL, without giving them a Swarmfile account or a seat.

When you create a link you can add:

- A password, stored hashed and verified server-side — the file's contents are never exposed by the password mechanism alone, see below
- An expiry date
- A cap on total access count

The access-count cap is enforced by the hub on every tier, but the unit differs. For managed-key shares it counts **downloads**: each download is checked *and incremented* against the cap.

On an **end-to-end encrypted** repository the cap counts **distinct verified visitors**: the first open by a verified email consumes one access, and re-opens (or block-token re-mints) by that same address don't. The cap is a hard limit on distinct verified email addresses for E2E shares.

Expiry, revocation, email verification, and the per-recipient block list are all fully enforced regardless of tier, and revoking or blocking now takes effect on the very next block request rather than at session expiry. (The share dialog hides the access-count field for E2E repositories; an API-created limit still enforces as described.)

Every share link requires email verification, regardless of whether you've also set a password. Before a recipient can see any content — download, preview, or the comment thread — they have to enter their email and click a one-time magic link sent to it. This isn't optional and it isn't skippable by knowing the password: the password (if set) and the email verification are both required, independently. Once verified, a recipient gets a 30-day session, so a returning collaborator isn't re-verified on every visit.

This gives the share owner a real access log: every verified recipient, by email address, with when they accessed the link. You can block any individual recipient from that log at any time, and the block takes effect on their very next request — an already-open browser tab doesn't get to keep working.

Shares support inline preview for images, PDFs, and text files, plus an optional public comment thread scoped to the shared item.

Drops: send files without an account

The `/send` Finder window is a real drop surface: drag files or a folder onto it (or choose them) and Swarmfile creates a public-by-link repository for you, with no account. The first time you create a drop in a browser, you confirm that you have the right to share the files; the confirmation is remembered in that browser, so later drops don't ask again.

One drop is at most **5 GiB** and **200 files**, and it expires **7 days after upload**; anyone with the link can view and download the files until then. A drop is a **plaintext public repository** — a public link, not an encrypted share.

Up to **5 drops per network address per 24 hours** (25 per 30 days; the address is a hashed client IP) are allowed; a shared office, VPN or carrier NAT can reach that cap. Browsing the drop uses no disk; files are stored on Swarmfile until the link expires, then deleted. Files aren't scanned for malware — only open links you expect.

From the drop page you can:

- **Copy the link** (`/d/<id>`) and send it yourself. The share page's share block and the live card also draw a **QR code** for opening the drop on a phone.
- **Email the link** to up to five people after confirming your own address. Recipients see the name you provide (or "A Swarmfile sender"), never your email address; we don't use the addresses for marketing, and they're deleted when the drop is purged.
- **Email me a copy**, which sends the drop link to your own confirmed address. The page reports how many addresses the mail service *accepted*; delivery isn't tracked, so a bounce won't show there — ask a recipient to check spam if an email doesn't arrive.
- **Keep a copy in a repository** (the claim) — in the browser you uploaded from. The copy lands in a new repository in your organization (a paid org's is managed and private; a Free org's is plaintext and public) and enables Git on it; the public link closes once the hub records the copy as complete (usually within a few minutes, and within about 15 minutes even if the tab is closed).
- **Delete the drop**, which stops the link for everyone immediately.

A signed-in visitor drops directly into their last-used repository instead: the files are ordinary repository versions (no public link and no 7-day expiry), and the hero can show that repository's top-level entries. The 5 GiB drop limit is per drop and separate from your plan's repository storage.

External collaborators

External collaborators are named guest reviewers — an M&E client, an AEC consultant — invited to a specific folder with `read` or `comment` permission, without consuming a paid seat. See Permissions for how folder-level access grants work generally; external collaborators are layered on top of that same system, scoped to a folder rather than a whole repository.

What a guest gets:

- A read-only mount of the folder they were invited to. Writes are refused by the mount itself, not a UI restriction — a `read`-permission guest cannot write to the mount even by bypassing the dashboard entirely.
- An offline-first comment and upload queue in the Desktop App, for guests granted `comment_upload` — the tier that adds uploads to comment access.
- Instant revocation and a full audit trail on the org side, with a durable revocation sweep you can watch under Settings → Access revocations.
- If the same person is invited to multiple orgs as a guest, they get a picker in the Desktop App to switch between them.

External collaborators are plan-gated for **private repositories**: Starter includes 1 per seat, Pro includes 5 per seat, and Enterprise is unbounded by contract. Starter and Pro both allow billed overage up to 3x the included allowance before new invitations are hard-blocked — see Billing & Plans for how overage is metered and charged.

On public repositories, collaborators are free and consume nothing: a public repository is readable by anyone, so a `read`, `comment` or `comment_upload` grant on it never occupies a plan slot and never accrues overage — which is why a Free-plan org can have guests on its public repositories (its private-repository guest allowance is zero).

A repository can also refuse new guests for itself (inheriting or overriding the org policy; existing grants are not revoked) — see the repository's settings or `swarmfile repository show`.

On a public repository, guests can also arrive without an invitation: **public access requests** are on by default (turn them off in the **Repository settings** tab), so any registered user can ask for comment access — or **contributor** access, the `comment_upload` tier — and an owner or admin approves it from there. Approved grants land on the repository's `Contributions` folder; everything above about the guest experience applies unchanged. See Public Repositories & Org Profiles.

Anonymous Drops & Claiming

A drop is the shortest path from "here are my files" to "here is a link": drag files onto the drop page and you get a real, public-by-link repository you can share without anyone installing anything or creating an account. A drop expires after seven days unless you keep a copy (the claim) in a repository of your own.

Drops are a hosted-hub feature. A self-hosted community server does not serve them.

Creating a drop

1. Open the drop page while signed out (or signed in — a signed-in visitor uploads directly into their last repository instead).
2. Drag files or a folder onto the Finder window. Folders keep their relative paths.
3. The first drop in a browser asks you to confirm you have the right to share the files. The acknowledgement is remembered in that browser; later drops, a retry, and a signed-in direct upload do not ask again.
4. The upload runs in the browser. When it finishes, the drop's share page is ready.

A drop is capped at **5 GiB total, 200 files, and 7 days**. It is **public and plaintext**: anyone with the link can read and download it, so never put confidential material in a drop. The link is unguessable, but it is a bearer capability — treat it like a password.

The per-network-address create limits (5 per 24 hours, 25 per 30 days; the address is a hashed client IP) and the global byte/email budgets are abuse bounds, not a plan limit.

Sharing the link

The share page (`/d/<drop-id>`) is a delivery receipt. From there you can:

- **Copy the link** or show a **QR code**.
- **Email the link** to up to five people. See Emailing recipients.
- **Email me a copy** to your own confirmed address.
- **Keep a copy in a repository** (the claim) once you sign in.
- **Delete the drop** before it expires.

The public link keeps working until the drop expires, is deleted, or is claimed. A claimed drop closes for everyone once the hub records the copy as complete (usually within a few minutes; within about 15 minutes even if the claimant's tab is closed).

Emailing recipients

Emailing the link is consent-based and verifies the sender first:

1. Enter the recipients (up to five addresses) and **your** email address.
2. We email **you** a confirmation link. Recipients get nothing until you open it.
3. After you confirm, each recipient gets one email with the drop link and your optional message.

Only your email address is confirmed; your name is optional (leave it blank and recipients see "A Swarmfile sender"). Addresses are used only for this drop and are deleted with it — never for marketing. Delivery is not tracked beyond the mail service accepting a message, so ask the recipient to check spam before assuming the link never arrived.

Claiming a drop

Keeping a copy (the claim) copies the files into a **new, git-enabled repository** in an organization you belong to. The drop itself stays live until the copy finishes; if the copy fails, the drop is still there and you can retry.

1. Sign in from the share page.
2. Choose the organization and edit the proposed repository name.
3. Confirm. The copy runs in the background and the page shows byte progress.

A claim belongs to the browser that uploaded the drop: it needs both your session and the drop's upload key, which is kept in that browser. Opening the link in another browser or signing in as a different person cannot claim it. If you clear site data before claiming, the drop can no longer be claimed — only the public link remains.

The copy is asynchronous. Large drops can take several minutes; the copy continues if you leave the page, and reloading the share page in the same tab resumes the progress display — even if the public link expired while the copy was running, the copy's progress and outcome are still shown.

What the claimed repository looks like

- It is a normal repository with real versions and history; each copied file is a commit on the default branch.
- It is **git-enabled**: once the copy completes, the repository page shows `git clone swarmfile://<hub-host>/<org>/<repository>` (and the repository is available through the Git remote helper).
- Its encryption tier and visibility follow your organization's plan. A paid organization creates the usual managed/private repository; a **Free organization creates a public, plaintext repository**, because a no-card org has no managed key. The drop's public link closes when the hub records the copy as complete, but a public repository remains readable by anyone who can reach it.
- The uploaded bytes are re-encrypted when the target is a managed repository. The content is identical; the storage format is the target repository's.

Expiry, deletion, and retention

- A drop expires **seven days** after creation. Public reads then answer `410`, and the repository and its blocks are purged by the reaper.
- **Delete** from the share page closes the public link immediately and purges the drop's staging; the repository and its bytes are removed on the next GC pass (usually within minutes, at most about 15). Deleting is refused while a claim is copying.
- A claimed copy is an ordinary repository and is not subject to the drop TTL. It follows the repository's own retention and trash settings.
- The drop's registry row is retained for 90 days for audit and abuse handling after its content is purged.

Troubleshooting

- **"This drop's upload key is required" / "This key does not belong to this drop."** The claim must come from the browser that uploaded it. Use that browser (site data intact), or ask the uploader to claim it.
- **"This drop has already been claimed."** Someone claimed it; the copy is in their repository. The public link is closed.
- **The email never arrived.** The page reports what the mail service accepted, not delivery. Check spam on both the sender-confirmation and recipient sides.
- **The copy is still running after several minutes.** Large drops are copied in bounded background batches; the page keeps polling at a slower cadence. Keep the tab open, or reload it in the same tab to resume the status.

Worksharing (no plugin)

Revit, SolidWorks, and many CAD/BIM tools already take a native operating-system lock on the central model while someone has it open — they open the file for writing and *deny write-sharing* to everyone else. On a plain network drive that lock only means something on one machine. Swarmfile **honors and enforces that native lock across every Windows machine** on the drive, so two people can't open the same central model for editing at once — with **no add-in and no separate lock server**.

Because it works at the filesystem level, not inside the application, there's nothing to install into Revit or SolidWorks. The app takes the lock it always takes; Swarmfile makes it real across every Windows machine on your team. (This enforcement is Windows-only — see Platforms.)

How it works

When a native application opens a file on the mounted drive **for writing while denying write-sharing** — an *exclusive open* — Swarmfile registers a short-lived claim on that file for your machine. While that claim is live:

- Anyone on **another machine** who opens the same file the same way is refused. On Windows the open fails with a sharing violation, which the application surfaces the way it always does — "the file is in use," "could not open for writing," or the app's own central-model-locked message.
- The claim is renewed automatically for as long as the file stays open, so a model left open all day — or overnight — keeps enforcing. When the app closes the file, the claim is released and the next person can open it.

This is the same model a native app uses locally, extended across the internet. You don't change how you work: open the central model, and if someone else has it, your application tells you — just like it would on a shared drive that actually enforced the lock.

Who has it open

When your open is refused, Swarmfile also records a **Recent issues** entry in the Desktop App naming the user and machine that holds the file, so you're not left guessing at a bare "file in use." That is separate from the dashboard's Presence view, which is driven by Swarmfile's own lock lifecycle and can disagree with the passive open signal (see Working with Files).

Platforms

Native exclusive-open enforcement is a Windows capability, because the "open for write, deny write-sharing" semantics it builds on are a Windows filesystem concept — and Revit and SolidWorks are Windows applications. A cross-machine exclusive open is enforced between Windows machines.

macOS and Linux mounts don't take part. An exclusive open on a Mac or Linux mount registers no claim, and a Mac or Linux machine opening a file that a Windows machine holds exclusively is not refused by this mechanism. If a file is edited from mixed platforms, use an explicit edit lock instead.

For teams whose work spans macOS and Linux — the linked assets around a model, or a Rhino/QGIS/IFC ecosystem — the cooperative entry locks below work on every OS. The byte-range lock API is cross-platform too, but only on repositories set to the Revit-worksharing repository type — chosen at creation or set later with `swarmfile repository set-type revit_worksharing` (see layer 3 below).

When the hub is briefly unreachable

Worksharing claims are **hub-enforced**: a claim is registered with the service, and every machine checks it before allowing an exclusive open. Deployments that need an open refused when the hub is unreachable can enable that per machine with `SWARMFILE_WORKSHARING_FAIL_CLOSED=1` on the engine (see Environment Variables); a claim that can't reach the hub then **denies** the open instead.

The three layers of locking

Worksharing is the top layer of Swarmfile's locking; each layer answers a different need. From coarsest to finest:

1. **Native exclusive-open enforcement (this page).** Whole-file, automatic, no plugin, Windows-only. The app's own "open the central model" is the lock. Best for Revit/SolidWorks-style worksharing where opening the file *is* checking it out.
2. **Entry locks.** Two related kinds.
   - The **automatic entry lock** is the 60-second, heartbeat-renewed claim an open file takes while an app has it — nothing to manage.
   - An **explicit edit lock** is one you take yourself with `swarmfile lock` (7 days by default, up to 30, auto-renewed every 6 hours while your engine is online — offline, the lease simply counts down) or a `git lfs lock` (a fixed 30-day lease that nothing renews); both kinds have a "request unlock" flow (`swarmfile unlock-request` from the CLI).

   A Pack & Go reservation is a separate mechanism that reserves a whole folder or repository rather than one file. Best when you want to reserve a file you aren't holding open in an app.

   On a repository used as a git-LFS server the entry lock is the **same** server-side lock as a `git lfs lock`, so the two mutually exclude across surfaces — and files that live only in git (never written to the drive) get an equivalent path-keyed lock so `git lfs lock` / `unlock` / `locks` still work for them.
3. **Byte-range locks.** Element-level locking *within* a single file — the way a BIM tool locks just the elements one person is editing without locking everyone out of the model. This is a lower-level API a native-app plugin calls directly; the swarmfile-brlock CLI is the reference client. It is enabled only on repositories set to the Revit-worksharing repository type — at creation, or later from **Repository settings** or `swarmfile repository set-type revit_worksharing`; on any other repository the engine refuses byte-range requests.

Most teams never touch layers 2 and 3 for worksharing — opening the model is enough.

What this is not

Worksharing enforcement is a **write lock**, not a merge system. It stops two people editing one central model at once; it does not merge two divergent copies of a model back together (native BIM/CAD merge is the application's job). For versioning the model over time — branching a design option, rolling back a bad save — see Version Control and Branches & Merging.

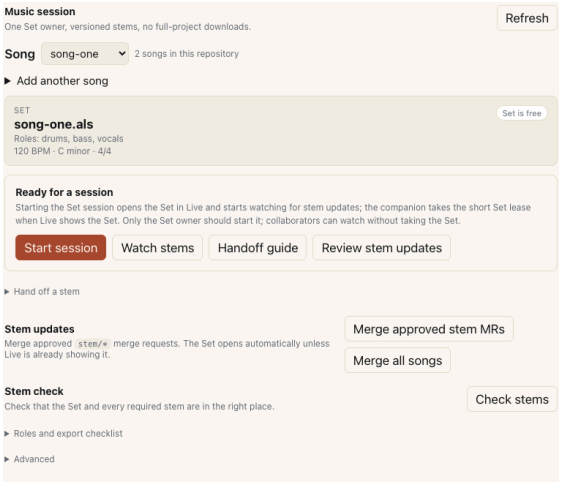
Ableton Live Collaboration (Stem Handoff)

Ableton Live has one writer per Set. Two people cannot edit the same Set at once, so the usual workaround is exporting stems to a shared folder and reconciling versions by hand. Swarmfile does not pretend Live can be multi-user. It makes the **stem handoff** real: one shared repository, one folder per role, branches and merge requests for each handoff, a validator that catches a wrong sample rate before the mix owner opens the file, and a session lease so the Set has one owner while Live is running.

This is **parallel production**, not real-time co-editing. It is the workflow most production teams already use with a shared folder — with versioning, per-folder access, and no full downloads.

The model

- **One Set owner (the mix owner).** Only they write `Songs/song-one/song-one.als`.
- **Stem owners.** Each collaborator owns `Songs/song-one/Stems/ROLE/` and never edits the Set.
- **Stems are the merge surface.** The Set is not merged. A collaborator branches from `main`, overwrites their stem, and opens a merge request. The changed paths are disjoint, so the request merges cleanly. The hub scopes a `stem/song-one/<role>` merge request to `Songs/song-one/Stems/<role>/`: a request that also carries the Set, another role's stem or any other path is refused rather than partially merged. The mix owner reopens the Set and hears the new stem from the same fixed path.
- **The Set is a single-writer document.** Branch it to try an arrangement, but hand the whole Set back; never merge two Sets.



Multiple songs

A repository can hold several songs. Each is a folder under `Songs/` with its own `stems.json`, project file and `Stems/` tree; the template starts with `Songs/song-one/`. Rename the starter `song-one` folder to a meaningful song id, copy it to `Songs/<song>/` and update its manifest, or use **Add another song** on the Desktop App's **Music** page; a folder name is 1-32 characters, starting with a letter or digit; then lowercase letters, digits, `_` or `-`.

The Music page and stem branches

The Music page lists the songs and works on the selected one: the song selector swaps the Set, roles, handoff, merge action and session state. A song's stem branch is `stem/<song>/<role>` and the hub scopes its merge request to `Songs/<song>/Stems/<role>/`.

Each role's `path` in `stems.json` must stay under `Stems/<role>/`; a repository with a custom role path should move the file into that folder and update the manifest before validating.

Keep song ids stable

Give each song's Set a unique filename: the companion matches Live's window title, which carries only the file name. Do not rename or move a song folder while a stem handoff is open — the folder name is the song id, so old `stem/<old>/<role>` branches and MRs keep pointing at the old path; merge or close them first. Existing single-song repositories with a `Song/` folder keep working with `stem/<role>` branches.

Create the repository

Create one from the **Music production (Ableton Live)** template in the web app, the Desktop App's **New repository** dialog, or:

```
swarmfile repository create "Song" --option template=music
```

The template seeds:

- `Songs/song-one/` — Live's project folder, with `Songs/song-one/Samples/` and `Songs/song-one/Stems/ROLE/`.
- `Songs/song-one/stems.json` — the handoff contract (Set path, tempo, format, roles).
- `tools/stem-check.mjs` — the validator.
- `tools/ableton-companion.mjs` — the Node fallback Set lease and stem watcher (the native command is `swarmfile stems companion --manifest Songs/song-one/stems.json`).

- `Exports/`, `Docs/`, and a `.swarmfile/ci.yml` MR check.

The examples below use the starter song `song-one`; substitute a song's folder name for another song.

It also enables **protected ACLs** and keeps every saved version. Your plan must include folder ACLs to use the template. On a self-hosted community server the key model is owner/member; the Set-owner convention is a team rule there rather than a per-folder ACL.

First setup (mix owner)

1. Save the Set as `Songs/song-one/song-one.als`. Use **Save** or **Save As** without collecting; see the Collect All and Save trap.
2. Add one audio track per role and a clip that references the fixed path `Stems/ROLE/ROLE.wav`, relative to the song folder. Keep the filenames and the export format fixed. `swarmfile stems setup --manifest Songs/song-one/stems.json` prints the exact track/clip checklist (it never edits the Set); `--add-role ROLE --write` adds a role and its folder.
3. Edit `Songs/song-one/stems.json` for the roles, tempo and format you use.
4. Grant access. Give each collaborator **write** on their `Songs/song-one/Stems/ROLE` folder and **read** on the project root; keep write on `Songs/song-one/song-one.als` and `Exports/` for the mix owner. See Permissions.
5. Run the strict validator:

```
swarmfile stems check --strict --manifest Songs/song-one/stems.json
```

The collaborator loop

Branch from the song's base branch (usually `main`), open a mount on that branch, and open the Set from the **branch mount** so your new stem plays in context:

```
tools/handoff.sh drums main song-one
# Windows: tools\handoff.ps1 -Role drums -Song song-one -Base main -MountPoint S:
```

In the Desktop App, **Music → Hand off a stem** does the same: choose the role, create the handoff mount, export to the fixed path, then **Commit + create MR**. **Grant access** opens the role folder ACL and **Open folder** reveals the path. The equivalent manual commands are:

```
swarmfile branch create stem/song-one/drums --from main
swarmfile mounts open ~/Swarmfile-handoff-song-one-drums --label song-one-drums --branch stem/song-one/drums --set-current --wait
```

On Windows, pass a drive letter instead of a folder path. See Multiple Mounts. These examples assume `main`; if the repository's default branch has another name, use it here and in the CI `branches` filter. A song's branch is `stem/<song>/<role>`; a one-segment `stem/<role>` is only for a single-song repository using the root `Song/` layout.

Then:

1. Overwrite only `Songs/song-one/Stems/ROLE/ROLE.wav` — do not edit `Songs/song-one/song-one.als`.
2. Save and commit. The drive auto-commits after an idle window; run `swarmfile commit` to close the batch now.
3. Validate and open the handoff:

```
cd ~/Swarmfile-handoff-song-one-drums
swarmfile stems check --strict --manifest Songs/song-one/stems.json
swarmfile commit -m "drums stem"
swarmfile mr create --title "Drums v3" --from stem/song-one/drums
```

Use `Songs/song-one/Stems/ROLE/alt/` for alternate takes; keep the canonical file as the one the Set references.

The mix owner loop

From the Desktop App, open **Music** in the activity bar and click **Start session**. That starts the companion and opens the Set in Live; the companion takes the lease when Live is showing the Set, and the page shows the holder, Live detection and stem-update notices. Use **Watch stems** instead when you are a stem collaborator. The commands below are the equivalent for scripts and older checkouts.

Run the companion from the mounted repository root before opening Live:

```
swarmfile stems companion --manifest Songs/song-one/stems.json
```

The seeded `tools/ableton-companion.mjs` is the Node fallback for older checkouts; the native command compares each stem entry's current content CID (`rootCid`; local/committed, not a timestamp) and needs no Node.

While Live is running, the companion takes a **short session lease** on `Songs/song-one/song-one.als` with `swarmfile lease` and renews it. The lease is a reference on the ordinary standard lock: the engine heartbeats it, so another machine's write is refused and it sees who holds it. The default lease is 2 minutes and the companion renews at half; when Live stops, the companion drops its lease. If Live still has the Set open, close it to free the hub lock (Ctrl+C drops the companion lease too). If someone else holds the Set, the companion names them and prints:

```
swarmfile unlock-request create Songs/song-one/song-one.als
```

The companion also watches the stem files and reports when one changes, so you know when to reopen the Set to hear it. Stem collaborators run the same command with `--watch-only` to get that notification without holding the Set; in the Desktop App that is **Music → Watch stems**.

If your Live install reports a different process name, set `SWARMFILE_ABLETON_LIVE=1` while Live is open (or `0` while it is closed) to override detection. Pass `--title-match song-one.als` to lease only while Live's front window shows that Set (macOS may ask for Accessibility permission; Windows uses the main window title). macOS asks for notification permission on first use and Windows toasts are best-effort when the CLI companion is run with notifications enabled. The Desktop App Music page shows these notices in-app and also allows the companion's desktop notifications.

If the companion is killed rather than stopped, its session lease is reaped in about two minutes; **Finish session** in the Desktop App drops it immediately. A manual `swarmfile lease Songs/song-one/song-one.als --release --token TOKEN` needs the companion's token; a tokenless release cannot drop another process's lease. The hub lock remains while Live (or another mount) still holds the Set open.

After a stem merge request lands, reopen the Set. If you keep the Set and the stems at their fixed paths, the new audio plays from the same clip — no Set edit is needed.

Merge approved stem MRs

From the Desktop App's Music page, **Stem updates → Merge approved stem MRs** merges the approved `stem/*` requests that target the branch you have open and whose diff stays inside its role folder; **Merge all songs** runs the same action across every song.

An owner/admin can enable **Merge settings → Merge approved stem MRs automatically**; it merges only stem requests whose target is the repository's default branch, leaves the request open on any conflict or failed gate, and emails/notifies the owner when a refused automatic merge needs attention.

Anything that still needs a reviewer, a conflict resolution or a fresh approval is reported and skipped, and the action never touches the Set. When the merge succeeds, the page opens the Set unless Live is already showing it; use **Reopen Set** if the merge came from elsewhere or Live was already open.

The validator

`swarmfile stems check` checks:

- the Set exists and the manifest's roles are present;
- every present WAV's sample rate, channels and bit depth match `Songs/song-one/stems.json`;
- stems are aligned in duration (or match an explicit `durationSeconds`);
- the Set references the fixed stem paths;
- the Set does not carry absolute media paths;
- the Set has not been rewired to a collected copy under `Songs/song-one/Samples/`.

Use `--strict` for the handoff gate: missing required stems, missing Set references, absolute paths, collected stems and duration mismatches fail. `--json` gives scripts a machine-readable result. `--max-set-mb N` raises the Set scan cap for a very large arrangement. `--quality` adds sample peak and clipped-sample counts for PCM/float stems; integrated LUFS and oversampled true peak remain a separate metering step.

The seeded `.swarmfile/ci.yml` runs the strict Node validator on merge requests that touch a song folder once an owner enables **Hosted CI**; `tools/stem-check-all.mjs` checks every song. An MR run executes on the **source** branch (`stem/*`), so the owner must allow automatic runs on unprotected branches even when `main` itself is protected (see Hosted CI).

The Collect All and Save trap

Collect All and Save copies media referenced from **outside** the project folder into `Songs/song-one/Samples/` and rewires the Set to those copies. Stems inside `Songs/song-one/Stems/` are already in the project folder, so they are not collected. The trap is a stem that lives outside the song folder: collecting moves it and the fixed path is no longer what the Set plays.

For the master Set, use **Save** (or **Save As** without collecting). Keep `Songs/song-one/Samples/` for samples the Set actually uses and `Songs/song-one/Stems/` for the handoff bus. The validator flags a Set that references a collected stem copy. If you must collect, do it on a copy of the Set and keep the master on the fixed paths.

Keep paths **relative inside the song folder**. Live stores files inside the project folder relatively, which keeps the Set portable across machines — including a different Windows drive letter.

Ableton analysis files (.asd)

Live writes `.asd` files next to audio with transient/warp analysis. The template ignores `*.asd` so machine-specific analysis does not collide; Live regenerates it on first play. A rendered stem's `.asd` is disposable — the Set's warp markers live in the Set, not the stem.

A shared sample with hand-edited warp markers is the exception: its `.asd` is not synced. Keep that sample in `Songs/song-one/Samples/` and sync its `.asd` explicitly (remove the ignore line for that path in a branch), or bake the warp into the rendered stem. Never sync the global cache under `~/Library/Application Support/Ableton`. See `Docs/analysis-files.md` in the repository.

External collaborators

Public repository: the guest requests access on the repository. The self-serve request targets **Contributions/** (not a chosen folder) and grants `comment_upload` — upload and comment, not a branch write or a lease. The guest uploads the stem there; the mix owner moves it into `Stems/<role>/` (or adds a role and wires it in the Set) and commits. A guest cannot create a subfolder, overwrite a canonical stem, or hold the Set lease. The access request records the approval; the flow does not record a separate sharing-rights assertion.

Private repository: a guest read/comment grant cannot upload. Invite a recurring mixer as a member with write on their role folder (a seat), or use an upload-only drop the owner creates. The seeded `Songs/song-one/Stems/incoming/` folder is for that member/owner drop, not a self-serve public request target. See Permissions.

Other DAWs

Create the **Music production (DAW-agnostic)** template for Pro Tools, Cubase, Studio One or Reaper (file-based projects). Save the project as `Songs/song-one/Session.<ext>` and set `set` in `Songs/song-one/stems.json` to that filename. The manifest's `setFormat: "opaque"` tells the validator not to parse the project; it still checks the project exists and every stem, so keep the fixed `Stems/ROLE/ROLE.wav` paths in the DAW's own bin or playlist. The validator, handoff script and stem model are identical.

A Logic `.logicx` is a package directory: the validator accepts it, but the standard session lease on the bundle does **not** gate its child files. Use a Pack & Go checkout for the Logic bundle — the Music page's **Start Pack & Go** starts it; the generic **Start project session** can hold the project lease but is not a substitute for the bundle checkout.

The live Set companion is Ableton-specific. For a file-based DAW, the Desktop App starts a generic project session. A manual `swarmfile lease Songs/song-one/Session.<ext>` returns a `sessionToken`; pass `--token` when renewing and releasing, or adapt the companion's process probe.

What not to do

- **Do not edit or merge `Songs/song-one/song-one.als` on a collaborator branch.** Branch the Set only to try an arrangement, then hand the whole Set back.
- **Do not rename a stem file** without updating the Set and the manifest. The fixed path is what removes the Set edit from every handoff.
- **Do not change sample rate or bit depth mid-song.** Update `Songs/song-one/stems.json` and re-export deliberately.
- **Do not use a guest upload grant for a canonical stem.** A guest `comment_upload` grant is upload-only and lands in `Contributions/`; the owner moves it, or use a member seat for a recurring collaborator.

Troubleshooting

- **Live cannot save the Set / the companion says it is held.** Another person holds the Set. `swarmfile locks` shows the holder; the holder can drop their companion lease from the Desktop App (or with its `--token`), and the Set is free once they close Live; or ask with `swarmfile unlock-request create Songs/song-one/song-one.als`.
- **A new stem is not heard.** Reopen the Set; check that the Set references `Stems/ROLE/ROLE.wav` (not a collected `Samples/` copy); run the validator. `swarmfile stems status --manifest Songs/song-one/stems.json` shows the Set and each role's engine sync state and current `rootCid` if you need to confirm a stem actually synced.
- **The companion cannot take the lease.** If you are a stem collaborator, run it with `--watch-only`; only the Set owner holds the lease. Run `swarmfile lease Songs/song-one/song-one.als` directly to see the hub's refusal; the session lease does not use the Pack & Go policy.
- **`stems check` reports a missing required role.** Export the stem to the manifest path, or mark the role `"required": false` if the song does not use it.
- **The validator reports an absolute media path.** Move the sample inside the song folder and relink it in Live.

RFIs & Submittals

RFIs and submittals are structured, numbered, auditable review workflows layered directly over your repository files. They're the coordination loop AEC teams run constantly — *ask a formal question and get a formal answer* (an RFI), or *submit a document and get it stamped* (a submittal) — but there's nothing construction-specific about the machinery. It's a review workflow with a paper trail, and it works for any team that needs "someone submitted this, someone decided on it, and here's the record of who and when."

The workflow is built by composing the pieces the rest of Swarmfile already provides: an RFI or submittal has a folder of attached files, so those attachments get previews, comment threads, version history, locking, and ACLs for free. What the workflow adds on top is state, assignment, due dates, numbering, and a decision of record.

Who can use them

The **RFIs** tab in the web dashboard is visible to **owners and admins**. Access to an individual RFI's contents is governed by the ACL on the folder it lives under, the same as any other file — so a member granted access to that part of the repository can see and act on it even though the top-level tab is admin-facing.

RFIs vs. submittals

They share one engine but differ in intent and in the decisions available:

- An **RFI** asks a question. Its decision is an **answer** — or, if you need it, Approved / Rejected / Revise & Resubmit / a plain comment.
- A **submittal** puts a document up for review. Its decisions are the review stamps: **Approved as Rev A**, **Approved as Rev B**, **Approved as Noted**, **Rejected as Noted**, and **Revise & Resubmit**.

You pick the kind when you create the record.

Creating one

Two entry points:

- **From a folder** — right-click a folder in the Files view and choose **Create RFI from this folder**. The new-RFI form opens with that folder as the attachment location already set.
- **From the RFIs tab** — create one directly and point it at a folder.

On the form you set the title and question (or submittal description), the kind (RFI or submittal), the reviewers it's assigned to, a due date, and a priority (**low**, **normal**, **high**, or **urgent**). You can attach files, which land in the record's own folder and behave like any other repository files. Save it as a **draft** to keep working, or **submit** it to open it and notify the reviewers.

Lifecycle and ball-in-court

An RFI moves through a defined set of states:

draft → open → in review → answered → closed. A record can be **voided** at any point.

Alongside the status, every record tracks **ball-in-court** — whose turn it is, the originator or the reviewer. Submitting a draft puts the ball with the reviewer.

Revise & Resubmit always leaves the ball with the reviewer, on both kinds — it's not a terminal decision, and a submittal specifically also bumps its revision (a → b → c, capping at c).

What flips the ball back to the originator differs by kind:

- For an **RFI**, only the **answer** decision does — **Approved**, **Rejected**, and a plain **comment** all leave the ball with the reviewer.
- For a **submittal**, all four terminal review stamps do — **Approved as Rev A**, **Approved as Rev B**, **Approved as Noted**, and **Rejected as Noted** each hand the ball back to the originator and move the record to *answered*, since each one closes out the review; the originator's own `closed` action is the only step left after that.

Ball-in-court is what the due-date reminders key off, so it's worth knowing which decision types move it.

Making a decision

A reviewer responds with a **decision** — an answer or a review stamp from the sets above — plus a message, and optionally a signature. The **decision timeline is the record of account**: it's a distinct, append-only history, separate from both the comment thread (informal discussion) and from file changes (which land as changesets). When you need to show *what was decided, by whom, and when*, that timeline is the answer, and it's what an audit looks at.

Requesting revisions sends the RFI back to *in review* with the ball returned to the reviewer; an answer moves it to *answered*; the originator accepting closes it.

Numbering

Each repository has its own RFI counter, and numbers are **gap-tolerant**: a record gets its number when it's created (drafts included), and voiding it leaves that number as a gap rather than renumbering everything after it. No number is ever silently reused — the same convention tools like Procore follow, so the numbers you cite in correspondence stay stable.

Attachments, comments, and previews

Because an RFI's attachments are ordinary repository files under the hood, everything else in the product applies to them: thumbnails and inline preview in the detail view, comment threads with `@`-mentions, per-file version history, and locking. Marked-up PDFs, drawings, and photos attached to an RFI preview inline the same way they would anywhere else.

Notifications

RFI activity generates notifications so nothing sits waiting on someone who doesn't know it's their turn:

- **RFI created** and **assigned** — to the reviewers it's routed to.
- **Response posted** and **revisions requested** — to whoever the ball moves to.
- **Due soon** and **overdue** — to the current ball-in-court, driven by a daily sweep.
- **Closed** and **voided** — to everyone involved.

There's also a **ball-in-court digest** for a periodic summary of what's waiting on you. These reach you through the in-app bell, Desktop App toasts, and email (with the same per-kind preferences and quiet hours as everything else — see Notifications & Inbox).

External consultants

The person who answers an RFI or reviews a submittal is often outside your org — a consulting engineer, an architect of record. Two paths, both fully audit-trailed:

- **A "respond" share link** — for first contact with no account. The consultant opens the link, verifies their email once (no Swarmfile account is created), sees the RFI read-only, and posts a decision directly. The decision is recorded against the share with their verified email, so the response has a real trail even though there's no account behind it.
- **A full external collaborator** — when the consultant is engaged properly, invite them as a guest (they get their own account). Their decisions carry their own identity, for a complete audit trail.

An external consultant you invite as a guest counts against your plan's external-collaborator allowance when their grant is on a **private** repository — a public-repository guest is free and consumes nothing; a respond-link consultant has no account and doesn't count at all. See Sharing & Collaboration and Billing & Plans.

Public repository intake

A public repository opens a filing window (on by default, per repository) so any registered Swarmfile user — a client, a consultant, a reviewer — can raise an RFI from the repository's public page without joining the organization (turn it off under **Repository settings** → **Public page** → **Public RFI intake**).

The filing lands in an auto-created `RFIs/` folder at the repository root with a priority the submitter picks; the team sees the submitter's verified name and email (recorded from their sign-in, never taken from their text), owners and admins get a dedicated notification and email, and the submitter gets an email when a member answers.

Submitters track and reply to their filings under **My requests**, and members can filter the RFI list by public origin. Non-member attachments aren't accepted yet — a filing can describe the issue, and a member attaches files afterward.

Public discussions

A public repository's page can also host **Discussions** — the same RFIs, published instead of private. Off by default; the team turns them on under **Repository settings** → **Public page** → **Public discussions** (the Desktop App has the same control, and so does `swarmfile repository set allow-public-discussions=true`).

- **Publishing a thread.** Any member can publish an RFI from its detail panel (**Publish as discussion…**). A preview names what becomes public — the title, the question, and your display name — and nothing written under confidentiality goes with it: existing decisions stay internal until you publish them individually (**Publish to public page**, reversible with **Hide from public page**). A filing that came through the private intake form can't be published at all; its submitter was promised it would never be public.
- **Replies.** On a public thread the composer defaults to **Public reply**, which appears on the page and emails the people who posted there (one-click unsubscribe included); **Internal note** keeps a message on the member thread only. Non-members reply from the page itself with a verified account.
- **Moderation.** Any member can **Pause public comments** on one thread, or freeze them repository-wide (**Repository settings** → **Public page** → **Freeze public comments**) — reads and internal notes continue either way. **Make private** demotes a thread off the page and notifies its participants; nothing is deleted. A public author who abuses the page can be **banned** from the RFI timeline (the ban list and **Lift ban** live under **Repository settings** → **Public page**), and a thread can be **Reported** by a signed-in reader, which rings the owners' and admins' bell.

Finding them

RFIs and submittals are full-text searchable by title and body from the dashboard omnibox, alongside files, repositories, and people — see Search. A match jumps straight to the RFI.

Desktop App and CLI access

From the Desktop App

Reviewing and deciding is coordination work, not work you do inside the mounted drive, so there's no RFI panel in the Desktop App. What it does surface is **RFI notifications as toasts**, and a deep link that opens the RFI in your browser, so someone working in the Desktop App still hears about an RFI waiting on them.

From the CLI

Full command coverage — create, list, decide, and manage RFIs and submittals without leaving a terminal, the same workflow a build script or a headless render-farm node can drive:

```
swarmfile rfi create ./Projects/tower-a/foundations --title "Slab thickness at grid C4" --body "Drawing S-201 shows 300mm, spec calls for 350mm — which governs?" --priority high --reviewer user_88
swarmfile rfi list --outstanding
swarmfile rfi status rfi_9f2a
swarmfile rfi decide rfi_9f2a answered -m "350mm per spec — drawing will be revised"
```

Unlike `mr`, RFI commands (other than `create` / `list` / `search` / `by-file`) take an RFI reference that you can give either way: the friendly display number (`RFI-003`, case-insensitive, resolved through the mounted repository) or the internal `id` from `rfi create` / `rfi list`. `rfi status <ref>`'s exit code doubles as a CI gate the same way `mr status`'s does: `0` if `answered` / `closed`, `2` for every other state. See the CLI reference for the full command list.

Where to go next

- Sharing & Collaboration — comments, share links, and external collaborators, which RFIs build on.
- Notifications & Inbox — how the RFI alerts above are delivered and configured.
- Version Control — changesets, which is where an RFI's *file changes* (as opposed to its decisions) are recorded.

Public Repositories & Org Profiles

A repository can be **public**: its releases are browsable and streamable by anyone with the link, it can appear on Explore, and its organization gets a public profile page. (The release page's main download is the repository's binary for your platform — or the self-hosted bundle — served anonymously. Individual release files and the release's source archive are open to everyone too, with a quick proof-of-work check. Attached release assets are grouped below the README, with checksums.) Nothing about this changes how the repository works for your team — the drive, locks, history, and permissions are identical.

Public repositories are for distributing things — release assets, datasets, model weights, sample content. A team can also open a public repository to **access requests**: any registered Swarmfile user (no organization membership needed) can ask for comment access, contributor access, or an invitation to join the organization, and an owner or admin approves each request.

Approved guest and contributor requests on a public repository are **free**: they consume no collaborator allowance and are never billed, on any plan (including Free). That's separate from share links and external collaborators, which are for inviting specific people into private work.

What "public" means

Two things are set when a repository is created and don't change afterward:

- **Visibility.** `Public` or `Private`. The choice is permanent — there's no "make this repository public later" switch. If you need a public copy of private work, publish a release into a new public repository (the **Copy** action on a public release copies it into a repository in your own org — always a public one, since it copies a public release; a **private** destination takes the Desktop App's CLI copy instead, see Publishing releases → Getting a copy).
- **Storage tier.** Public repositories store content on the plaintext tier, because anything the world can download has nothing left to encrypt at rest. A repository that needs managed or end-to-end encryption at rest can't be public.

Where a public repository shows up depends on its releases. Its card appears on the org profile as soon as the repository is public, but Explore and the repository's own page stay empty until the first release is published — there is simply nothing to browse yet. Archiving a repository removes it from the listings until you unarchive it.

The repository page

Every public repository has a page at `/public/<org-slug>/<repository-slug>`. It is organized as **Files / Issues / Discussions / Releases** tabs: Files is the tag-scoped README and file browser, the release you are viewing follows you when you switch tabs, and Releases lists every published release with its file and download counts. The Issues and Discussions tabs appear only when the repository enables them and show their open counts.

The page also shows the contributors of the latest release. When the page is shared, the link unfurls with a generated preview card — the repository name, its description, and its public counts; a link to a recent release previews that release (tag, file count and size). From here a visitor can:

- **Browse a release** and download individual files (no account needed — each download passes a quick proof-of-work check), or the release's source tree as an archive where the release is set up for it.
- **Install the release.** When a release publishes an `install.sh` (the server's does), the **Install & clone** shelf carries an **Install this release** command pinned to that tag; the README's install section follows the latest release. The shelf command has a Copy button.
- **Follow releases** — a verified free account gets an email whenever a new release is published. No account is needed to subscribe to the **Atom feed** instead.
- **Star a repository** — a signed-in visitor can star it. The count sits next to the button (with the last week's gains) and feeds Explore's **Most starred** sort.
- **See provenance.** A card or release marked **commit-pinned** means the release names the content-addressed commit it was published from. You can check the repository's signed commit chain yourself with `swarmfile-verify-history`; the badge itself is the pointer, not a verification result.
- **Read the commit history** on a repository with git access enabled: when anonymous clone is available, the release page lists the commits leading to the release's pinned commit (short hash, message, author display name, date), with older pages a click away — the same history `git clone` serves. The history boundary can refuse it while a committed `.swarmfile/publicignore.yml` hides paths. Each commit opens a detail view with the full message, exact time and its parent hashes.
- **Clone it** — on a git-enabled repository with anonymous clone on (the default), the **Install & clone** shelf carries `git clone swarmfile://<hub-host>/<org>/<repository>` (one Copy button); a signed-out visitor can use it read-only. An owner can turn it off, and a committed `.swarmfile/publicignore.yml` that hides paths refuses anonymous clone entirely (see the git clone guide).
- **See a repository's forks.** A copy of a release made elsewhere links back to where it came from ("Forked from"), and this page lists the repository's public forks. A copy made through the CLI rather than the web action is labeled *claimed*: its source is real, but the copy itself isn't verified. A fork's page also compares the upstream's latest release with the release the fork was copied from, and Explore cards show each repository's fork count.
- **Browse by topic.** Each repository's slug-safe topic chips link to a topic page (`/topics/<topic>`) listing all public repositories carrying it, with the same sorts as Explore, and a **Follow this topic** Atom feed of new releases across those repositories. Explore's landing also shows a **Gaining traction this week** rail — a rolling 7-day window of new stars and active downloaders, never an all-time ranking.
- **Search inside the release.** The file panel's **Search file contents…** box searches the text files of the latest release (source, docs, configs; binary assets are never indexed, and the index stops at 300 files, 256 KiB per file and 2 MiB in total) and links each match straight to the file; the same query in Explore finds repositories by content, showing the matching file's path on the card. Hidden release paths are never indexed, and search covers the latest release only.
- **Sponsor the repository.** When the team has added funding links, a **Sponsor** button sits beside Follow (one link opens directly; several open a small menu), and the org profile's repository cards carry the links too (a single **Sponsor** link, or up to three labeled links). Swarmfile only shows the links; it never handles the money.
- **Embed a badge.** The page's **Badges for your README** row copies ready-made Markdown for five live shields: **release** (latest tag and date), **downloads**, **stars**, **provenance** (*commit-pinned* when the latest release names its commit, matching the wording above), and **verified oss** (green while the repository holds a Verified OSS grant, gray otherwise). The SVGs are rendered and served by Swarmfile itself (no third-party badge service), and each is cached at the edge for five minutes, so a hot README never turns into unbounded traffic.

Requesting access

A public repository accepts **access requests** by default (the team can turn them off in the **Repository settings** tab), so the repository page shows a **Get Access** button to signed-in visitors who aren't members. Anything below is refused until your email is verified.

- **Comment access** — read the repository and comment on files. No paid seat.
- **Contributor access** — comment, plus upload new files to the repository's `Contributions` folder. No paid seat; no overwrite or delete rights, and uploading needs the desktop app or the web file browser, not a raw upload elsewhere.
- **Membership** — ask to join the organization. Approval sends the ordinary organization invitation (which uses a seat); you accept it from the email.

One request per repository can be pending at once, and there's a small per-organization limit on simultaneous pending requests. You can **withdraw** a pending request from the page or from My requests, where every access request you've filed and its outcome are listed. An owner or admin approves or denies from the repository's **Repository settings** tab; you get an email either way. If a request is declined, a new request needs a short message explaining what changed. An approved membership whose invitation expires unaccepted shows as **Invitation expired** in My requests, and the repository page offers to ask again.

Public pages show the requester nothing beyond what they already see while signed out; your name and verified email are shared with the repository's owners and admins only as part of the request.

Discussions

A public repository's page can carry a **Discussions** tab — questions and notes the team has published, open to read by anyone with no account. Discussions are off by default; the team turns them on under **Repository settings** → **Public page** → **Public discussions**. Turning it off hides the tab (a direct URL says the surface is not enabled) and deletes nothing.

- **Replying** takes a free account with a verified email. Your display name appears on the thread and in the reply emails other participants get; your email is never shown. Whoever started a thread can close it; closed and paused threads stay readable.
- **Search** narrows the list by title, and a signed-in reader can **Report** a thread to the repository's owners and admins.
- **Emails.** Reply notices go to everyone who posted on the thread, with a one-click unsubscribe. Unsubscribing applies to that repository only, and My requests lists the discussions you've joined along with a **Turn emails back on** control.
- **The team's side** — publishing threads, internal notes, locks and bans — is covered in RFIs & Submittals.

Issues

A public repository can also carry a light **Issues** tracker. It's off by default; turn it on under **Repository settings** → **Public page** → **Public issues** (public repositories only, on the web or in the desktop app; the Desktop App groups the public-page switches — RFI intake, access requests, discussions, issues and anonymous clone — under **Public page features**). Turning it off hides the tab (a direct URL says the surface is not enabled) and deletes nothing.

- **Anyone can read** issues with no account. **Opening an issue or commenting** takes a free account with a verified email.
- An issue has a title, a body, up to eight labels, an open or closed state, and an optional assignee hint (display text, not an assignment). Any member can set labels, close or reopen an issue, and set the hint; the issue's author can close or reopen their own issue.
- **Search** matches one phrase in issue titles and bodies (a multi-word query must appear as written).
- **Moderation** shares the Discussions controls: an account banned from discussions can't open or comment on issues, and the repository-wide comment lock freezes both.
- The org's owners and admins get a **Public issue posted** notification (bell and email) for each new issue. Comment emails to participants work like discussion replies, with the same per-repository unsubscribe.
- Deliberately not included: boards, milestones, attachments, or editing an issue after posting.

Hosted federation export

If Swarmfile staff have pinned a federating community peer for your repository, an owner or admin can let the hosted hub serve this public repository's reads to it under **Repository settings** → **Public page** → **Hosted federation export** on the web. This is the owner-consent half: staff pin the peer and import its actor; the owner enables or disables the export.

It requires a public repository with git access and anonymous cloning on (the repository is already plaintext). It exposes public clone, history and LFS-download reads only — never private or E2E data, uploads or locks.

A staff override locks the switch (contact the operators), and the control shows a status instead when the hosted origin is disabled or no peer has imported the repository. See Self-hosting → Federating public repositories and git clone → Federated public reads.

The organization profile

An organization with public repositories gets a profile at `/public/<org-slug>`: its logo, bio, website and support contact, a **README**, its public repositories sorted by recently updated, most downloaded, or name (pinned repositories first), and a contributors strip merged from its public repositories. Shared, the profile unfurls with the organization's name and description. Member names are not shown — only people whose commits are already public on the repositories themselves.

Owners and admins edit this under **Settings → Organization → Public profile**:

- **Bio** — one line, shown under the organization name.
- **Website and support contact** — shown as links on the profile.
- **README** — markdown, rendered below the header. Use full `https://` links; relative file links aren't resolved there. The editor has a **Preview** toggle and a **View public page** link.
- **Pinned repositories** — up to 12 public repositories to feature at the top, in the order you pick.

Every edit to the public profile is recorded in the organization's audit log, with the fields changed but not their contents.

Following an organization

The profile's **Follow releases** button subscribes one verified email address to every public repository in the org. The same email rules as the repository follow apply:

- If you follow both the organization and one of its repositories, you get **one** email per release, not two.
- Every email carries a one-click unsubscribe link, and you can also unfollow from the profile while signed in.
- Prefer a reader? The profile links an **Atom feed** that carries the org's releases across all its public repositories: `/feeds/<org-slug>/releases.xml`.

Tools for maintainers

These are for the repository's owners and admins.

- **Funding links.** In the dashboard's **Repositories** list, open the repository's menu → **Edit public profile** and add up to five funding links: an Open Collective page (`https://opencollective.com/<your-collective>`) or any other `https://` link, each with an optional label. They power the **Sponsor** button.
- **Contributor recognition.** The release's contributor strip marks a **top contributor** by verified commits: the top one, or the top three once a release has ten or more contributors. There's nothing to configure.
- **Where readers come from.** A release's **Assets** dialog on the **Releases** tab shows which sites sent readers to the public page over the last 30 days, as referring host names and counts only. What is and isn't recorded is in Telemetry & Data Collection.
- **Weekly public-repository report.** Owners and admins of an org with at least one public repository get a weekly email covering the past seven days for each public repository: new stars, new followers, downloads, the newest release, and anything that needs attention (a takedown, a failed GitHub Releases mirror, or public forks that haven't picked up your newest release). It respects quiet hours; mute it under **Settings → Notifications → Weekly public-repository report**.
- **Upstream released.** When a repository you forked publishes a release your fork doesn't have yet, your fork org's owners and admins get one notification per release, by bell and email. The email names the upstream and the new tag and links to the upstream repository's public page. See Notifications.

Verified OSS grants

An open-source repository that is public and actively maintained can apply for a **Verified OSS grant**: the Pro plan for its organization at no cost, for up to 10 seats. The Swarmfile team reviews every application.

- **Apply** from **Repository settings → Public page → Verified OSS grant** (owners and admins). Optionally give the repository's SPDX license id (for example `Apache-2.0`) and a short motivation.
- **Eligibility**: the repository is public, its latest ready release is between 7 and 365 days old, and the organization is at least 30 days old. The checks run when you click **Apply**; if the repository doesn't qualify yet, the card lists the checks it failed.
- **Term**: a grant usually runs for a year; renewing is a fresh review. After a declined or withdrawn application you can apply again once a 14-day cooldown has passed.
- **While it's active**, the org gets Pro's limits and features (not Enterprise ones), **Billing** shows the plan as coming from the Verified OSS program with its expiry date, and the public page shows a **Verified OSS** chip and the green **verified oss** badge. The chip is a program status, not a security claim. Metered hosted CI still needs a card on file.

What visitors can see, and what they can't

Visible on public pages	Never shown
Release files and their download counts	Anything from a private repository
The README each release ships	Member names, roles, or email addresses
Contributor display names and commit counts	Account details or avatars
Org bio, website, README, pinned repositories	Storage configuration or bucket details
Follower count (a number, never a roster)	The follower list itself

Publishers can also keep specific paths out of a release entirely — see Keeping paths out of a release.

Archiving a repository or deleting a release removes it from the public pages, the org profile, and Explore; public pages are cached at the edge for a minute or two, so a removal can lag by that much.

Publishing Quickstart for Open Source Repositories

This is the shortest path from "code and big files on my machine" to "a public release anyone can browse and download, without an account." It is the open-source walkthrough of Publishing Releases — read that for every knob; this page is the ten-minute version.

What you get:

- A **public repository page** at `/public/<org-slug>/<repository-slug>`: rendered README, file browser, previews, release history, contributors, and a download button.
- **Downloads that need no account.** Visitors browse, preview and stream byte ranges anonymously; each file or source-archive download passes a quick one-time proof-of-work check (a few milliseconds in the browser) that makes bulk scraping expensive. Attached release assets are direct anonymous links. Egress is free and unmetered on every plan.
- **Explore and community surfaces**: a listing on Explore once a release is ready, optional topics and website on the repository's profile, follow-by-email and an Atom feed for new releases, stars from signed-in visitors, and optional public Issues and Discussions.
- **A Free plan that needs no card** for public repositories and private unencrypted repositories, with a flat 5 GiB of storage. Encrypted private repositories start on a paid plan. An established open-source repository can also apply for a Verified OSS grant (the Pro plan at no cost).

What to expect: Swarmfile is not a full forge (no repository boards, milestones or code intelligence), but a public repository gets a light issue tracker you can turn on, search inside the latest release's text files, and a fork-to-upstream proposal flow that lands as an ordinary merge request in the upstream repository. Git `clone` and fast-forward `push` work when the repository has git access enabled.

1. Create a public repository

In the web app, **New repository → Visibility: public**. Choose an organization slug and repository slug — they become the public URL and the git/LFS path, so pick names you're happy to keep. On a brand-new org, keep the default plaintext tier; a public repository must be plaintext (managed encryption is for private work).

No organization yet? Sign up, create one, and stay on the Free plan — public repositories are free.

2. Get your files in

Pick whichever fits your repo:

- **From your machine**: install the desktop app, mount the repository, and copy files onto the drive (Getting Started). Saves stream in the background.
- **From an existing git repository**: `swarmfile git import` brings the whole repository in with its history, branches and tags:

  ```
  swarmfile git import https://github.com/you/your-repository.git
  ```

  It defaults to a new repository named after the repository; `--name` / `--org` override, and `--into <org>/<repository>` retries a partial import. Fidelity limits are in Known Limitations.

- **Keep your repo where it is and send only the big files**: point `git-lfs` at Swarmfile and keep source on your existing host (Git-LFS):

  ```
  git config lfs.url https://hub.swarmfile.com/orgs/<org>/lfs/<repository>
  ```

Whatever the path, use `.swarmfile/publicignore.yml` to keep private paths out of a release **before** you tag (see step 4).

3. Tag what you're shipping

A release always publishes a **tag**, and a tag names one commit. Commit the state you want (on the drive, via `swarmfile commit` / `swarmfile tag create`, or via `git push` if git access is on), then:

```
swarmfile tag create v1.0
```

`swarmfile tag` never moves an existing tag (a forced `git push` can). Once a tag is published as a release, treat it as permanent.

4. Publish the release

The web **Releases** tab has a publish wizard with a preflight; the CLI is the same thing:

```
swarmfile release preflight v1.0   # exactly what would publish; exits non-zero if it can't
swarmfile release create v1.0      # publish
swarmfile release list             # pending → ready (or failed, with a reason)
```

The preflight shows the files and total size, anything `.swarmfile/publicignore.yml` keeps out, and any tracked paths your ignore files hide from the browser but would still publish. If the tag's policy hides paths, publishing requires anonymous cloning off (`409 publicignore_clone_exposure`); once the release is ready, turning anonymous cloning back on does not re-expose those paths. A new release starts **pending** while the download archive builds, then becomes **ready**. If a publish failed permanently, tag a new commit and publish that — re-running `release create` on the same tag does nothing.

5. Attach build artifacts

Compiled binaries don't belong in the tagged tree. Attach them to the release instead:

```
swarmfile release attach v1.0 dist/your-tool-1.0-macos-arm64.tar.gz
```

Or drag them into the release's **Assets** dialog in the web app. Files over 100 MiB attach with the CLI. Assets are served anonymously on their own rate-limited bucket — unlike release files, they carry no proof-of-work challenge (they're plain download links, which is what package managers and install scripts need) — and the member Releases tab shows a per-asset download count plus a per-path download-activity panel.

6. Polish the public page

An owner or admin sets what the public page shows from the dashboard's **Repositories** list: the repository's menu has **Add description** (or **Edit description**) for the description and **Edit public profile** for topics, website and funding links (more in Public Repositories & Org Profiles). The Files tab opens on the release's README, with Issues / Discussions / Releases alongside it, and the org profile at `/public/<org-slug>` collects your public repositories.

7. Keep the door open for contributors

- **Follow / Atom feed**: visitors can subscribe to release emails with the **Follow releases** button (a verified account) or the feed at `/public/<org-slug>/<repository-slug>/releases.xml`.
- **git** (if enabled in repository settings): `git clone swarmfile://<org>/<repository>` gives the full history when anonymous clone is available — it is on by default, refused while a committed `.swarmfile/publicignore.yml` hides paths, and member clones remain ACL-scoped; fast-forward `git push` sends commits back. The public page shows the release's commit history too, subject to the same boundary.
- **Requests in**: a signed-in visitor can request comment or contributor access from the public page; approved contributors upload into the repository's `Contributions` folder, and neither uses a seat (Requesting access).
- **Forks**: **Copy** in the repository actions copies the release into another org. The copy's public page links back with **Forked from**.
- **Issues and Discussions**: turn on **Public issues** and **Public discussions** in **Repository settings** → **Public page** (both off by default). Anyone can read them; posting takes a free account with a verified email. See Issues.
- **Contributions back**: an owner or admin of a fork picks a branch in repository settings' **Propose to upstream** card and sends it. The branch is copied into the upstream repository as a temporary `contrib/…` branch (a single flattened commit, up to 512 MiB) and reviewed there as an ordinary merge request. The upstream decides, and a proposal never merges itself. The upstream controls who may propose with its **Cross-repository contributions** setting: **Off** (the default), **Proposals** (a direct fork's owner or admin may send one), or **Open** (also a fork of a fork, up to five hops away).
- **Funding**: add up to five funding links (Open Collective or any `https://` link) under the repository's menu → **Edit public profile** in the dashboard's **Repositories** list, and the public page shows a **Sponsor** button beside Follow.

8. Mirror to GitHub Releases (optional)

If the repository is linked to a GitHub repo (the GitHub App), the repo row's **Mirror releases** checkbox mirrors each future ready release there — tag, notes and attached assets. It is off by default, one-way (GitHub-side edits are never imported), and unpublishing a release here deletes its mirrored GitHub release. A CI-driven team can instead paste a workflow that calls the repository's mirror endpoint on `release: published` (see Git-LFS → Mirror releases); both routes run the same idempotent mirror, so either can be safely re-run.

9. What's deliberately not built

Named so you can decide with the real picture: Issues are deliberately light (no boards, milestones, attachments or author edits); content search covers the latest release's text files only, not every branch; `git push` is fast-forward only, and git access is off by default (turn it on per repository); public repositories are plaintext. The Known Limitations page keeps the full list.

Publishing Releases

A **release** is a tag of a public repository, published for anyone to browse and download: a fixed, cached snapshot of exactly what that tag points at. Releases are how you distribute a dataset, a set of sample files, model weights, an asset pack, or a game build from Swarmfile, without anyone needing to join your organization. Distributing an open-source repository end to end? Start with the Publishing Quickstart and come back here for every knob.

This page is the publisher's walkthrough. Public Repositories & Org Profiles covers what visitors see and how your organization's public page works.

Before you start

- **The repository must be public.** Visibility is chosen when a repository is created and never changes, and public repositories use the unencrypted storage tier (anything the world can download has nothing left to encrypt at rest). On the Free plan repositories may be public or private, but only a public one can be published; on a paid plan, choose **Public** when you create the repository. A private repository can't be published — copy the content into a new public repository instead.
- **You need the right access.** On a protected-mode repository, publishing needs repository-wide admin access (org owners and the repository's creator always have it). On an open-mode repository, any member who can write the whole repository can publish. You can't publish a tag that contains anything you're denied read on, and you need a verified email. Permissions has the details, including how a publish fails safe when permissions can't be checked.

1. Tag the commit

A release always publishes a **tag**, and a tag always names one commit. Commit the state you want to ship, then tag it:

```
swarmfile commit -m "Dataset v1.0"
swarmfile tag create v1.0
```

Tags are immutable pointers for the native `swarmfile tag` command: there's no "move this tag." To point a name somewhere else, delete the tag and create it again, or, if the repository has git access, move a lightweight tag through the git remote (`git push -f origin <tag>`; `git push origin :refs/tags/<tag>` deletes it).

Once a tag is published as a release, treat it as permanent: people may already have downloaded it, and the release keeps serving until you unpublish it. Tagging also protects that commit from history clean-up for as long as the tag exists.

The tag names `issues`, `discussions` and `releases` are reserved for the public repository page's tabs, and `releases.xml` is its Atom feed URL, so publishing under one of those exact names is refused; use a different tag name.

2. Publish it

Publish from the dashboard's **Releases** tab (repository → **Releases** → **Publish release...**). The wizard walks through choosing the tag, checking exactly what will become public, and confirming — the preflight shows the files and total size, anything `.swarmfile/publicignore.yml` keeps out, and any tracked paths `.gitignore` / `.swarmfileignore` hide from the web file browser that will still publish. If you fixed `.swarmfile/publicignore.yml` after cutting the tag, the preflight says the tag's policy differs from the branch's current one — publish a new tag to apply the fix.

You can also publish from the Desktop App's **Releases** page, or from the CLI — `swarmfile release preflight v1.0` shows exactly what would publish (files, hidden paths, ignore-file mismatches, a policy that differs from the branch) without publishing anything, and exits non-zero when the tag can't be published:

```
swarmfile release preflight v1.0
swarmfile release create v1.0
swarmfile release list        # shows pending → ready (or failed)
```

A new release starts **pending** while Swarmfile builds its download archive, then becomes **ready** (or **failed**, with a reason). Re-running `release create` on a tag that's already published does nothing, unless the earlier attempt failed in a way that can be retried. If a publish failed permanently, tag a new commit and publish that.

To take a release down, use **Unpublish** on the Releases tab, or `swarmfile release delete <id>` (the id comes from `release list`). Unpublishing is a real delete, needs the same access as publishing, and removes the release from your public pages and Explore within a minute or two of edge caching.

What a release includes

- **The whole tagged tree**, browsable file by file on the repository's public page at `/public/<org-slug>/<repository-slug>`. Anyone can browse, preview, and stream byte ranges of a release's files without an account — a visitor can scrub a large video or dataset before downloading anything. (On a bring-your-own-storage org, anonymous streaming from your bucket is off until an owner enables **Allow anonymous streaming from my bucket**.)
- **Downloads without an account.** Downloading individual files, or the release's source tree as a single `.tar` **archive**, needs no account. Every file or archive download request carries a quick, one-time proof-of-work check in the browser (a few milliseconds of CPU that makes bulk scraping expensive), plus per-network fairness limits — a visitor never has to sign in. Attached release assets are separate anonymous downloads; the public page's primary download picks the platform build (or self-hosted bundle). From the CLI: `swarmfile release fetch --org <org> --repository <repository> --tag v1.0 --out release.tar`, or add `--path` for one file. Files marked executable (`swarmfile chmod +x`) extract executable from the archive (mode `0755`); every other file extracts as `0644`. Serving those downloads is free: egress is unlimited and unmetered on every plan, so a widely used release costs storage, not bandwidth. (Anonymous reads carry an abuse backstop — a platform-wide hourly ceiling that can refuse further anonymous reads under extreme volume; see Telemetry & Data Collection.)
- **The README** the release ships, rendered as the landing view of the repository page.
- **Generated release notes** summarizing what changed since the previous release.
- **Download counts** per release (they lag by up to about an hour), and the contributors of the release.

- **Attached assets.** A release can carry named binaries that are deliberately outside the repository tree — build artifacts, installers, model checkpoints — served at `/public/<org-slug>/<repository-slug>/assets/<name>` and listed in the release page's Assets section. From the CLI:

  ```
  swarmfile release attach v1.0 dist/widgets-1.0-macos-arm64.tar.gz
  swarmfile release assets v1.0
  swarmfile release detach v1.0 widgets-1.0-macos-arm64.tar.gz
  ```

  The dashboard's Releases tab manages the same set from **Assets** on a release: attach a file up to 100 MiB, or remove one. Larger files attach with the CLI only, which uploads them directly to the repository's storage (up to 5 GiB). Re-attaching the same name replaces the asset. It also shows each asset's download count and bytes served, plus a per-path, 30-day **Download activity** panel for the whole release (members only).

Streaming allowance

Anonymous byte-range reads of a public repository — browsing a file's bytes, scrubbing a release video, or reading release media — draw on the organization's **public streaming pool**. The pool refills continuously; the plan sets its soft and hard limits, and a hard cap refuses further anonymous ranges with a short `Retry-After` while the soft crossing only warns.

The dashboard's **Public streaming** card shows the live pool to any member (owners also see it under **Settings → Usage**), including the warning line and any operator override. This allowance is separate from egress: member and P2P transfer stays unlimited and unbilled, and a platform-wide anonymous-read backstop is an additional ceiling (Telemetry & Data Collection).

Keeping paths out of a release

Some of a repository's files shouldn't go to the public page even when the rest should — internal notes, key material, licensed assets, an unpublished cut. Commit a `.swarmfile/publicignore.yml` at the repository root and list the paths to hide (and, optionally, set how platform-generated derived data is exposed):

```
version: 1
paths:
  - "internal/*"
  - "**/*.secret"
  - "!internal/public/"     # re-include a folder a pattern above hid
```

`paths` uses the same gitignore-style globs as `.swarmfileignore`: `*`, `**`, a trailing `/` for a folder, and a leading `!` to re-include — last matching line wins. As in git, a folder must be re-included before anything inside it can be.

`derived` sets the visibility of data the platform generates around a release rather than files you committed. Its `ci_logs` key sets the exposure of CI-run logs attached to a release (`member-only` by default; `public` exposes them to anonymous visitors):

```
derived:
  ci_logs: member-only   # or public
```

`member-only` (the default when the file or the key is absent) keeps those CI logs visible to members only; `public` exposes them on the public page alongside the release.

The file is read from the version committed on the tag being published, and applied as the release is built:

- Hidden paths are left out of the public release entirely — no browse entry, no raw stream or download, nothing in the archive, file search or Explore. They aren't merely blocked; they never enter it.
- The policy file itself is never published, since its globs name exactly the paths it hides.
- If the file is invalid, the publish **fails with the reason** instead of publishing everything. Fix it and publish a new release. (A release that fails because storage couldn't be read can simply be retried.)
- Changing the file affects **future publishes only**. A release that's already published keeps the contents it was built with — publish a new tag to apply a change.

The policy hides nothing from members; ACLs still apply. The policy governs the public release and browser surfaces, and it also sets the **anonymous git history boundary**: while any active branch's committed tree carries a `.swarmfile/publicignore.yml` that hides a path in it, any tag's committed tree does, or a ready (published) release hid paths, anonymous cloning is refused outright while that committed head, tag or ready release hides them, though a release already published keeps the contents it was built with.

The boundary evaluates both committed trees a branch can advertise (its confirmed head changeset's tree and its live head tree — they differ for a fork taken from a dirty source) and every tag's changeset tree. A path that exists only in an older commit is not retroactively hidden.

The publish wizard's preflight audits a bounded slice of the tag's history (up to 100 commits, under its own 800-tree-object and 200,000-file caps — separate from the anonymous-clone boundary's 100-tree/500-object/200,000-file caps) and warns when paths hidden by the policy also exist in older commits, listing a small sample of their names for owners and admins; that check is advisory and never blocks publishing.

Treat the warning as a remediation prompt: rotate any exposed secret and rewrite the history (or move the repository) rather than relying on the policy, because a policy is not a retroactive history scrub. The audit reports `truncated` when it stops at a cap, so its count is then a lower bound.

There is no filtered clone, because git history is content-addressed: hiding paths at the same commit ids would change every tree's OID and contradict the public history the project page shows. An anonymous clone also carries commit messages and the committed `.swarmfile/publicignore.yml` verbatim (filtering either would change commit ids); the web commit list and the federation activity/commit feeds redact path names the policy hides, but a clone does not — never put a secret in a public commit message.

An invalid or unreadable policy, or a boundary read/walk error, refuses anonymous clone (fail closed). A change takes effect within about a minute as the boundary verdict expires; the git refs themselves are answered `Cache-Control: no-store` and served live, not from an edge or browser cache. A policy that hides nothing does

not block.

Publishing still requires anonymous cloning off while a release would hide paths: the web wizard says so, disables **Continue**, and offers a one-click **Turn off anonymous clone**; scripts get a `409` with code `publicignore_clone_exposure` and the same remediation. Once the release is ready, turning anonymous cloning back on does not re-expose the paths while the ready release or an active branch's committed tree still hides them. Hiding a path is not a retroactive takedown either: it keeps the path out of new releases, but copies already downloaded stay where they are.

Provenance: commit pinning

A release names the commit hash its tag points at, and the repository page shows it with a **commit-pinned** badge. Because commit hashes chain every file's content and all the history before it, the hash pins exactly what was published.

The release also carries a manifest digest signed with the hub's Ed25519 history key, and the public page recomputes that digest in your browser and checks the signature, without downloading anything. When the signature checks out against Swarmfile's signing key, which is built into the site itself rather than taken from the release, the page shows a green **Verified by Swarmfile** badge. A valid signature by any other key reads *Signed by an unrecognized key*, and a digest or signature that doesn't match is called out explicitly.

The commit-pinned badge itself is the pointer, not a proof: anyone can check the repository's signed commit chain with `swarmfile-verify-history`, as Verifiable History explains, and `swarmfile materialize hash:<commit-hash>` reproduces the exact tree on any machine with access to the repository.

Owners can edit the generated release notes from the release page (a per-release summary of up to 4,000 characters), and the edit survives re-publishes.

Being found: Explore, following, and feeds

- **Explore.** Every public repository with at least one release is listed on Explore, Swarmfile's public directory, with search over names, descriptions, topics, README text and file paths. There's no separate opt-in: public plus a release means listed. Until the first release, a public repository's card shows on your org profile, but there's nothing to browse yet. Archiving a repository removes it from the listings until you unarchive it.
- **Follow releases.** A visitor with a verified free account can follow a repository (or your whole organization) and gets one email per new release, with a one-click unsubscribe.
- **Atom feed.** Your organization's releases across all its public repositories are published as a feed at `/feeds/<org-slug>/releases.xml`, no account needed.

Getting a copy: fork into your own org

On a release's public page, **Copy** (in the repository actions) copies that release into a new repository in an organization where the visitor can create repositories, then offers to open it in the Desktop App. It's a one-time snapshot: the copy doesn't track the original, and later releases don't flow into it.

Small releases copy straight away; a large one runs as a background job with progress and a cancel button. The destination organization's plan limits and storage quota apply, and copies are rate-limited per person.

The copy the web flow creates is a **public** repository (it holds a public release); pasting that release into one of your **private** repositories runs on your own machine instead, through the Desktop App's CLI — `swarmfile clone <org>/<repository> --into <your-org>/<repository>` (add `--tag <tag>` to pick a release) — which encrypts the files there before they leave your computer.

A copy records where it came from: the new repository's public page links back with **Forked from**, and the original's page lists its public forks. The web action's copy is a *verified* fork (the platform performed it); a copy created by the CLI's `--new --public` records the same canonical source but is labeled **claimed**, because the copy itself isn't verified.

This is also the way to get a public copy of work: publish into a public repository, or copy a release into a new repository of your own.

Current limitations

- Forks are one-time snapshots: later upstream releases don't flow into a fork (its owners and admins get an **Upstream released** notification instead), and a fork's public page links back with "Forked from". A fork can propose a branch back to the upstream as a merge request when the upstream's **Cross-repository contributions** setting allows it (Publishing Quickstart, step 7). Stars and release-follow live on each repository's public page (Public Repositories & Org Profiles).
- Public commit history is git-enabled-only: on a repository with git access turned on, the release page lists the commits leading to the release's pinned commit; on a repository without it, visitors see releases, not the commit log — that data was never public.
- Anonymous cloning is **on by default** for a public, git-enabled repository: the page's **Install & clone** shelf shows the repository's `git clone` command and a signed-out visitor can use it — read-only, no push. An owner turns it off from **Repository settings → Public page** (web and desktop), or `swarmfile git anonymous-clone off`; while it's off the page doesn't advertise the command. Members of a git-enabled repository can always `git clone` it.

Troubleshooting

Something not behaving? Start with the Desktop App's **Diagnostics** panel, or run the standalone report:

```
swarmfile-doctor --json --report doctor-report.json
```

It checks the engine, DNS, the hub, the sign-in token, cloud storage, the peer-to-peer network, mDNS, the firewall, and the mount itself, and its exit code is a verdict (`0` all passed, `1` a failure, `2` warnings only). Most problems below show up there by name, and its output is the first thing support will ask for. See swarmfile-doctor for every flag.

The drive isn't there

- **Windows default drive letter changed.** If the usual letter was taken at startup, Swarmfile picks the next free one — that is per-machine and runtime-determined. Read the actual location from the Desktop App's Status/rail entry rather than expecting a fixed letter.
- **The boot mount's drive letter is held by something else.** The failure card offers **Use another drive letter** — it forgets any explicit letter you set and reconnects, so Swarmfile picks a free one; no `config.json` edit or manual relaunch needed. `Open Another Drive…` stays available too, for working at a different letter (or an empty folder) right now, without restarting. If the letter comes from `SWARMFILE_MOUNT_POINT` on that machine, the action refuses and says so — change the environment variable instead. If every letter is taken, choose an empty folder instead — `auto` has nothing left to pick.
- **A second mount failed.** Windows cannot mount over a folder that already has files in it: pick an empty folder, or let `swarmfile mounts open auto` choose the next free drive letter. A drive letter something else holds is refused with a message naming what holds it. From the CLI, `swarmfile mounts list` shows every open mount and `swarmfile mounts open <path> --label "…"` opens another.
- **The mount was healthy and stopped answering.** The drive is mounted to *wait* rather than fail mid-write, so a stalled mount looks like a hung Finder/Explorer window instead of an error. Open **Settings → Diagnostics** and use **Repair drive** (not available on Linux); it reconnects in seconds without touching your queued uploads. On Windows, an administrator can have Swarmfile do this automatically when a drive operation stays stuck, with the `SWARMFILE_FS_CALLBACK_RESCUE_SECS` setting (off by default). The full background is in If the drive stops responding.
- **macOS: no virtual-drive slots are free.** Every FUSE-T app on the Mac shares 18 mount-helper slots. When they are all held, Swarmfile reports `fuse_t_helper_ports_exhausted` and no drive can attach. Close a drive or quit the app holding a slot, then use **Try again** or `swarmfile mounts retry` for a boot mount that never attached; a boot mount that attached earlier needs a restart, and a failed runtime mount is re-opened. `swarmfile doctor` names the holders it can identify.
- **macOS: the drive keeps dropping** (the Desktop App's "keeps disconnecting" card). A leftover mount from a killed session is the usual cause; the engine clears those before each mount, so **Repair drive** normally reconnects it. If Diagnostics still reports a stuck mount, restart the computer — a reinstall won't clear it. See If the drive stops responding.
- **The drive has never appeared and the app says the config file is broken.** A syntax error in `config.json` drops the whole file, including `org_id` / `repository_id`, so status names the file and the parse position instead of asking you to pick a repository. Fix the JSON and restart Swarmfile — see Engine Config File.
- **Access is still being granted.** A repository you have just been added to (or your account's first device) shows its drive only once its encryption key reaches this machine — for an end-to-end-encrypted repository that means another member's client, or your org's granter (about a five-minute cadence at worst), has wrapped the key to it — and if no grant was ever queued, your engine asks the hub to open one itself. A running engine watches for the grant on its own and mounts the drive as soon as it lands, so leave the app open and let it finish rather than reinstalling or re-adding the repository; **Recent issues** in Diagnostics shows the wait. See End-to-End Encryption.
- **macOS/Linux:** the mount is a folder, and on a machine that has never mounted anything the engine may still be finishing startup. Give it a moment, then check Diagnostics.

A file won't open, or opens very slowly

- **First open pays for the network.** Opening a file you've never read streams from a directly reachable peer if one has it (LAN first), otherwise from the cloud (a peer reachable only through the public relay is skipped by default); the Desktop App's Status shows peer counts, and `swarmfile status` prints the same. If peers exist but transfers never come from the LAN, run the doctor — the `mdns listen` and `host firewall` checks name the cause.
- **macOS is silently denying local network access.** On macOS 15+, the engine needs *Local Network* consent (System Settings → Privacy & Security → Local Network) to find peers on your LAN. The doctor's `macos local network` check says so explicitly when a LAN connection fails for that reason, and that row in **Settings → Diagnostics** offers **Open Local Network settings** to take you straight there.
- **Corporate Wi-Fi blocks mDNS.** Discovery is multicast; on a VLAN or Docker bridge it may be blocked even though the hub is reachable. Set an office grouping (`swarmfile office set <name> --lan-from-office`) so same-office peers are treated as LAN anyway — see Network Requirements.
- **If it's a wait, not an error.** A version that has never finished uploading can be opened; reads that outrun the upload return a retryable "try again" rather than an I/O error, deliberately, so applications don't mistake a slow upload for a damaged file. For the part you actually need, ask for it first: `swarmfile fetch <path> --tail 209715200`, or follow a moving viewer with `--follow` — see Working with Files.
- **Every file fails, not just one.** If opens across a whole folder or drive return I/O errors while the listing still looks fine, the engine treats it as a stale key or access problem: it notices the pattern and refreshes the repository's key and scope in place, and the files start opening again within a minute or two — no restart needed. **Recent issues** (Status/Diagnostics) shows the automatic refresh while it happens. If it is still failing after a couple of minutes, restart Swarmfile; that is the last resort, not the first step.

Writes are refused, or an app reports a plain I/O error

A filesystem can't say "someone else changed this too", so blocked writes look like ordinary I/O errors to applications. **The Desktop App remembers why**: click the Status pill and read **"Why did my save fail?"** — it lists recent refusals with the file's path, the reason, and how many times it happened (the engine keeps the last 64). Match the cause:

- **Unresolved conflict** — the same file changed in two places. The file shows as read-only (macOS/Linux) and saves fail with "permission denied" ("access denied" on Windows) rather than an I/O error. `swarmfile conflicts` lists them; resolve with `swarmfile conflicts resolve <path> --auto` (or `--winner`

`local|remote|keep-both`). See Branches & Merging.

- **Someone else holds the file** — the refusal names who holds it and their machine; `swarmfile locks` shows who, and `swarmfile unlock-request create <path>` asks them to release it. See Entry locks.
- **A Pack & Go lease lapsed while you were offline** — the reservation expired (leases only renew while online). Re-run `swarmfile offline prepare` when you have connectivity; see Working Offline (Pack & Go).
- **A bulk Pack & Go reservation covers the file** — a single-file lock and a *scope* reservation both refuse writes with `locked`, but a scope reservation blocks checkout/restore/merge too, not just saves. Its owner releases it with `swarmfile offline return`; see Working Offline (Pack & Go).
- **A native worksharing tool has the file open elsewhere** — the Desktop App notification names who and which machine. See Worksharing.
- **The file is still being created** (`pending_create`) — a lock, comment, share, or ACL action on a file whose first upload hasn't reached the hub yet. If the file is your own, renaming or deleting it from the app still works while it's pending (applied on this computer); this refusal is for the actions that need the hub. It's a hub `409` rather than a filesystem refusal, so it won't appear in "Why did my save fail?"; the Desktop App badge says "still being created". Wait a moment and retry, or check `swarmfile status` → `pendingCreates`.
- **The save is a new file at the repository's top level** (`root_create_restricted`) — a protected repository only lets its owner, its creator, or someone with repository-wide write access add files at the root. The save is held and retried, not lost: ask an org owner to give you repository-wide write, or create the file inside a folder instead. The same rule refuses a `git push` that adds a name at the root — the push error names the remedy; push the commit inside a folder instead. See Permissions and Clone a repository → Pushing.
- **You're offline and the file isn't in your reserved scope** (`offline`) — the drive is offline and this path wasn't covered by an offline reservation. Reconnect, or include it the next time you run `swarmfile offline prepare`.
- **This computer is quarantined** (`quarantined`) — the account was suspended after a burst of changes that looked like ransomware; an org owner clears it from **Settings → Quarantine**. On macOS and Linux, reads keep working and only writes are refused; on Windows the drive stays mounted and saves are accepted locally but simply don't sync until it's cleared. Already-queued saves are held, not lost — they resume once it's cleared. See Security.
- **Another machine is still uploading the file** (`uploading_elsewhere`) — its first upload is still in flight from wherever it was created; wait for that to finish.
- **Windows lost a cached write** (`unflushed_write_lost`) — a save was refused. Reopen the file and save again (or Save As).
- **Another drive on this computer has the file open** (`locked_by_mount`) — with exclusive-write mode, a drive refuses a sibling drive's write — and its rename, delete, or replace of that file — while it holds it (on Windows the write-open itself fails as "file in use"); the reason names the holding drive. Close the file there, or wait for that drive's save to finish.
- **Folder not empty** — a plain (non-recursive) delete of a folder that still contains files is refused, as on any disk. Delete it from your file manager, or with `rmdir /s` or `rm -r`, which empty it first. (PowerShell's `Remove-Item -Recurse` doesn't work through the mount — see Filesystem Compatibility.)
- **Pack & Go disabled by policy** — an administrator can disable bulk offline reservation with `SWARMFILE_PACK_AND_GO_POLICY=disabled` on the engine; the CLI reports `pack_and_go_disabled`. If the value was meant to be different, check its spelling: an unrecognized value disables Pack & Go and now warns at startup and in `swarmfile-doctor`.

Files aren't syncing

The Desktop App header always reports the truth: **Syncing N**, **Synced**, or one of the attention states below.

- **Waiting on plan** — the hub is refusing saves on plan grounds, usually because the organization's storage allowance is full. Raise the limit or free space; retries resume automatically. The plan panel shows the allowance, and if the pause is the org's spend cap rather than the plan allowance, an owner or admin raises it under **Billing → Spend cap**.
- **N changes not syncing** — the hub keeps refusing these saves for another reason. They're safe on this computer and Swarmfile keeps trying. `swarmfile sync stuck` lists each file, why, and since when; `swarmfile sync discard <entry-id>` drops the local change and returns the file to its cloud version (it asks first, and it cannot be undone).
- **This computer is quarantined** — the account was blocked after a burst of changes that looked like ransomware. On Windows the drive stays mounted and saves are accepted locally but don't sync; on macOS and Linux writes are refused. An org owner or admin clears it from **Settings → Quarantine**, and queued work resumes on its own. See Ransomware quarantine.
- **N changes failed** — work stopped retrying. The banner offers **Retry**; saving the file again also works.
- **A file was never supposed to sync** — check the exclusion rules: `swarmfile check-ignore <path>` says whether a path is ignored and by which `.gitignore` / `.swarmfileignore` rule. To remove already-synced, now-ignored content, `swarmfile ignore-clean` (soft delete, recoverable from Trash).
- **Signed out** — a lapsed sign-in pauses the queue without burning retries; sign back in and it drains. The doctor's `oidc token` check reports it directly.

A file others can see is missing for you

If a file others can see is missing on your drive, run `swarmfile cache resync`. It re-fetches this repository's change feed from the beginning; nothing local is dropped and cached bytes are kept. Run `swarmfile doctor` to confirm the hub and the mount are healthy, and if the file still doesn't appear, get help with the doctor output.

Uploads are slow (very large file)

Saving never waits for the upload: the write returns at local-disk speed and the durable queue drains in the background. A multi-hundred-gigabyte save on a modest uplink can legitimately take days. To see what's live, `swarmfile uploads` lists every in-flight upload on the mount (yours and other machines'); `--wait` blocks until none remain, which is handy before a checkout in CI. While a file is uploading, other people can fetch the part they need first — see While a large file is still uploading.

The same is true of **many files at once**: a mass edit or overwrite of hundreds of small files is paced to your plan's request allowance and can take minutes to reach **Synced** — see Uploads are designed not to block.

Storage says full

Storage is enforced per organization, and a Free plan's allowance is a hard stop rather than a bill. Free space (`swarmfile hydrate free-space`, or **Free up space** in the Desktop App) or move the organization to a paid plan; if the pause is the org's spend cap, raise it under **Billing → Spend cap** instead. The Billing & Plans page explains the allowance, overage, and where to change it.

Search returns nothing

Search queries a local metadata cache first, then the hub, so a file you've never seen in this repository may not appear until the hub is reachable — and it matches names, not full file contents. See Search.

An update didn't happen

Updates are never installed silently. The Desktop App checks for a new version at startup and periodically, and a found update only shows as a pill in the header plus a card in Diagnostics; **Install now** is the only path that installs it. If the pill has sat uninstalled for a week, you'll also see a one-time desktop notification reminding you — still no automatic install.

If an update won't take, use **Diagnostics → Reinstall...** for a full reinstall that keeps your queued uploads. On Windows, the doctor's `updater-helper-task` check reports whether the updater helper is registered; `sudo swarmfile-doctor --repair` (macOS/Linux; on Windows from an administrator prompt) does the same full reinstall from the command line, verifying the download before it runs.

Windows showed a SmartScreen warning during install

The blue **"Windows protected your PC"** box is expected the first time you run a newly downloaded installer — the `Setup.exe` / `.msi` installers are Authenticode-signed, but SmartScreen warns until their signing identity has built up download reputation, and that is not a sign anything is wrong with the download. Click **More info** (the small link, not **Don't run**), then **Run anyway** and finish the install; Getting Started has the step-by-step. It only appears when you run a freshly downloaded installer — the app's automatic updates don't show it.

If your IT department enforces an AppLocker or Software Restriction Policy, it may block Swarmfile's Explorer overlay DLL even though the app installs and syncs: files work, but status icons do not appear in Explorer. Ask IT to allow `C:\Program Files\Swarmfile`.

Signing in

- A message like "You may need to sign in again" means the session's refresh token expired or was revoked — sign in again and queued work resumes.
- SSO: enter your **organization's slug** (the short name in your org's Swarmfile URL). There's no email-domain auto-detection, so a personal email alone won't route to your IdP.
- A machine that can't sign in at all can still be diagnosed: `swarmfile-doctor` runs without an engine or a session.

"A file is still uploading" when branching or merging

`branch create`, `merge`, checking out or locking a file, and copying one are refused with `entry_uploading` while a file they'd pick up is still uploading. Taken mid-upload, they would silently use the file's previous version and still look complete. Usually you just wait for the upload to finish and retry. `swarmfile status` on the machine doing the writing shows its queue.

If that machine crashed or went offline mid-upload, restart Swarmfile on it: its queue picks up where it stopped and finishes the upload.

If the machine can't come back, the stuck upload stops blocking branch creates, merges from its branch, and copies once it has sent no progress report for 30 minutes. Pausing sync doesn't count as silence: a paused machine keeps reporting, so its uploads still hold. Locking or checking out that file, and a merge that would change the file itself, keep waiting for up to six hours: taking them early would refuse the upload if its machine comes back.

To clear the pointer explicitly — so readers stop waiting on it too — `swarmfile uploads retract <path>` (needs write access to the file; retracting a live upload is a no-op).

A commit wasn't recorded

When the hub won't record a commit, your saves are kept: they stay on the branch, just not grouped into a commit. The Desktop App shows a notification saying why:

- **"A commit was too large to record as one commit."** The change is bigger than the hub records as one commit (`too_many_entries`). Commit smaller batches.
- **"This branch is too large to record commits on this way."** The branch inherited more files than the hub can index (`fork_too_large`), so no commit on it will land, however small. You see this once per branch, not once per commit. Work on the default branch, or ask the repository owner for help.
- **No notification, the commit is just late.** A commit refused with `fork_index_building` (the hub is still indexing a new branch on a very large repository) or `do_write_budget_exceeded` (the repository's writes are paused) is held and retried on its own: after the hub's retry interval for the first, and when the pause lifts for the second. Neither counts as a failed attempt.

A merge won't finish

A large merge applies in the background once the hub accepts it — the command or app reports success before every file has landed, and `swarmfile merge --wait` blocks until it finishes. If one looks stuck:

- **Check the doctor.** The `merges in progress` check lists in-progress merges with their age; the service rolls back abandoned merges.
- **`stale_merge` is not stuck.** That refusal means the target branch moved after the merge was prepared; run `merge` again. It either lands cleanly or comes back with the genuine conflict that race exposed, which the normal resolution flow handles.
- **A repository owner can repair a genuinely stuck one.** A repair action for owners and support resumes a merge that stopped, or rolls it back (forced rollback is owner/admin-only). Nothing is lost either way: entries already applied stay, and a follow-up merge picks up whatever is still divergent.
- **A merge still being planned can be canceled** by the person who started it or an org owner or admin; one that is already landing answers `merge_in_progress`. See Branches & Merging.
- **A merge refused with `do_write_budget_exceeded`** was not started because the repository's writes are paused for the day. Retry after the time it gives, or have an owner or admin release the pause.

A drop link won't work, or a drop won't send

- **The link says the drop expired, was claimed, or was deleted.** All three close the public link for everyone. Ask the sender for a new drop; a claimed drop lives on in the repository it was claimed into.
- **The `/send` page refuses a new drop.** Anonymous drops are capped per network address (5 per 24 hours, 25 per 30 days; the address is a hashed client IP) and by global capacity. Wait for the window, or sign in and upload into a repository (no 7-day drop expiry).
- **An emailed link never arrived.** The page reports what the mail service *accepted*, not delivery; check spam and ask the recipient to do the same. If **Email me a copy** asks you to confirm, open **Email the link** once to verify your own address, then use **Email me a copy** again.
- **The "Keep a copy" option is missing on another device.** Keeping a copy and deleting are uploader-only and use a key stored in the uploading browser; clearing site data or using another device loses the claim (the link still works until expiry).
- **The first drop asks for a confirmation.** Tick "I have the right to share these files"; it is remembered in that browser, so later drops don't ask.

Getting help

When you contact support, include:

- `swarmfile-doctor --json --report doctor-report.json` (attach the JSON file).
- What you did, what you expected, and what happened — with the exact time and timezone.
- Whether it happens on one machine or everyone's, and on one file/folder or everywhere.

See Operations for admin-side diagnostics and connectivity checks, and Network Requirements if your firewall or proxy team needs the full list of hosts and ports.

Open Source at Swarmfile

Swarmfile's open-source half is the server you can run yourself and the standalone tools and libraries that work with or without the hosted service.

They are dual-licensed MIT/Apache-2.0 — source included — and published as public repositories on our own hub: browse the swarmfile organization to see each repository's code, releases and README (it also carries the LGPL source of the modified libfuse3 the macOS app ships). The open-source recordings on the demo page are these tools doing their real jobs, nothing re-enacted; for the design behind them, read Open-sourcing the server behind our AI-engineering VCS.

Run the server yourself

Swarmfile Server (Community) is a single-tenant server — SQLite for metadata, a local directory (or an S3-compatible bucket) for file bytes — that speaks the same API as the hosted service, so the desktop app, git, git-LFS and the standalone tools can all point at yours:

```
curl -fsSL https://get.swarmfile.com/server | sh
```

Self-Hosting the Community Server walks the whole path: the first-run sequence, connecting the desktop app's self-hosted edition, git and git-LFS against it, keys and roles, the peer-to-peer swarm, storage and backups, and upgrades. See it happen end to end:

Figure omitted in print (animation): First run of the community server on localhost: initialize a data dir, get an org and a repository, mint a key, serve the API — then the open doctor verifies it. - https://swarmfile.com/demo/oss-server.cast

The community server deliberately leaves parts of the hosted service out (access rules and sharing, end-to-end encryption, previews, hosted CI, the Office caches feature set, billing). The guide lists what stays hosted, and the client tools say so in place rather than failing.

Hosted CI is not part of the open-source offer. It is a metered hosted service on every plan, including public repositories, and we do not position it as free CI for open source. The workflow compiler and dialect are open source (`swarmfile-ci-workflow`); the compute is not.

The standalone tools

Each of these runs standalone — no engine and no mount required. Most have a one-line installer: `curl -fsSL https://get.swarmfile.com/<tool> | sh` , with `lfs` , `migrate` , `seed` , `mirror` , `doctor` and `git-v` as the channels; the git-LFS agent and the git remote helper also have Windows PowerShell forms (`irm https://get.swarmfile.com/lfs.ps1 | iex`).

Every tool release carries binaries for macOS, Linux and Windows, each on x64 and ARM. The macOS binaries are signed and notarized by Apple, the Windows ones are Authenticode-signed, and each archive has a `.sha256` file plus a `SHA256SUMS` list for the release, which the installers check before installing. On a public release page the assets are grouped by family; the per-file `.sha256` files and `SHA256SUMS` sit under **Checksums**, and each row's digest control copies the full SHA-256.

The standalone Rust tools that talk to a hub over HTTPS use rustls with the platform certificate verifier, so HTTPS trusts the host's certificate store rather than compiled-in roots. A minimal container image needs `ca-certificates` installed for the tools to reach an HTTPS hub.

- **`swarmfile-lfs`** — the git-LFS transfer agent: chunked, resumable large-object pushes through git-lfs's custom-transfer protocol. Source: swarmfile-lfs. See Git-LFS.
- **`swarmfile-git-remote`** — the standalone `swarmfile://` git helper: clone, fetch and push a repository with no engine installed, including an anonymous clone of a public repository. Source: swarmfile-git-remote. See Cloning a Repository with Git.
- **`swarmfile-migrate`** — import an existing directory tree, file list or S3 bucket into a repository. Source: swarmfile-migrate. See Migrating Existing Data.
- **`swarmfile-seed`** — place one local file at a repository path without mounting a drive. Source: swarmfile-seed. See Seeding a File Into a Repository.
- **`swarmfile-mirror`** — copy a repository, or a subtree, from one hub to another. Source: swarmfile-mirror. See Mirroring a Repository Between Hubs.
- **`swarmfile-doctor`** — connectivity diagnostics for a hub or a self-hosted server: DNS, TLS, health, the key, the block path, the declared profile. Source: swarmfile-doctor. See Diagnosing Connectivity.
- **`swarmfile-verify-history`** — the offline verifier for signed history; it ships with every desktop installer, and its source, releases and test vectors are at swarmfile-verify-history. See Verifiable History.

The libraries and contracts

The shared pieces are open source in the same repositories, so the tools and the engine are never forks of each other:

- **`swarmfile-chunker`** — the FastCDC content chunker the engine, the CI runner and the git-LFS agent all carry, published as a crate (`cargo add swarmfile-chunker`). The git-LFS agent itself installs with `cargo install swarmfile-lfs-transfer` .
- **`swarmfile-lib`** — the connectivity library: the hub wire contract, the route plan, the HTTP transport and the git helper's hub client. It ships as source inside the published repositories (the git remote helper above is built on it).
- **`swarmfile-api`** — machine-readable contracts: the hub's OpenAPI document, the community server's core-profile OpenAPI, and a generated dependency-free TypeScript client. Source: swarmfile-api.
- **`swarmfile-ci-workflow`** — the offline compiler for Swarmfile's native `.swarmfile/ci.yml` dialect, with a one-way GitHub Actions importer. Source: swarmfile-ci-workflow.
- **`swarmfile-sync-agent`** — the customer-hosted Active Directory / LDAP directory-sync agent. Source: swarmfile-sync-agent. See LDAP Directory Sync.
- The source for the modified **libfuse3** the macOS app ships (LGPL) is published too: swarmfile-libfuse3.

Figure omitted in print (animation): A standalone Rust program whose only dependency is the published swarmfile-chunker crate — the same chunker the engine, the CI runner and the git-LFS agent carry. It chunks an 8 MB master, then inserts 16 bytes in the middle: every chunk outside the edit keeps its content id, which is what makes dedup survive edits. - https://swarmfile.com/demo/oss-lib.cast

Work with Git through it

A public, git-enabled repository clones **with no credential at all** — the helper mints a short-lived proof-of-work ticket — and a key holder can clone, fetch and fast-forward push any repository with git enabled:

```
git clone swarmfile://my-org/my-repository
```

Figure omitted in print (animation): A public repository on the hub clones with the open helper and no credential — no sign-in, no key, nothing stored. The helper serves a shallow, depth-1 history by default and says plainly that git fetch --unshallow or --depth=N gets the rest. - https://swarmfile.com/demo/oss-anonymous-clone.cast

Cloning a Repository with Git covers turning git access on and what a clone contains; the helper reference covers install. None of this asks you to leave your git host: Git and Swarmfile maps the three ways the two fit together, including pointing stock git-LFS at a repository so only the big files live here (Git-LFS).

Get involved

- **File an issue.** A public repository can carry a light issue tracker — anyone can read it, and a free account with a verified email can post (Public Repositories & Org Profiles).
- **Fork it.** An org can fork a public release into itself, with provenance back to the source (Publishing Releases).
- **Publish your own.** Publishing Quickstart (Open Source) is the ten-minute path from a repository to a public release on the Free plan, no card required.
- **Check our work.** Every release on a public repository page carries its provenance, and `swarmfile-verify-history` lets you recompute a repository's commit chain from outside (Verifiable History).

Self-Hosting the Community Server

Swarmfile Server (Community) is the open-source server you run yourself (dual MIT/Apache-2.0; source and releases at swarmfile-server): a single-tenant service with SQLite for metadata and a local block directory (or an S3-compatible bucket) for file bytes. It speaks the same API as the hosted service, so the client tools that already talk to Swarmfile (the desktop app, the git-LFS agent, stock `git-lfs`, the migration tool, and the `swarmfile://` git helper) can all point at yours.

Running it yourself means running it on a computer you trust: the server holds your data, and on a managed repository it also holds the encryption key. What stays on the hosted service is everything around the core: access rules and sharing, **anonymous drops**, end-to-end encryption, previews, hosted CI, office caches, billing, and the hosted agent identity/delegation and service-minted agent-token surface (the community server still mints short-lived TTL member keys, a separate and narrower capability). The community server does serve the branch archive and the batch checkpoint, with its own caps.

Install

Binary (macOS, Linux, and Windows, each on x64 and ARM). The install script downloads the release for your machine and verifies its checksum; on Windows it points you at the release `.zip`; unzip it and run `swarmfile-server.exe`:

```
curl -fsSL https://get.swarmfile.com/server | sh
```

The server release page also carries an **Install this release** command inside its **Install & clone** shelf; that one is pinned to the tag you are viewing, while the one-liner above follows the latest release.

Container. The release publishes a multi-arch image (`ghcr.io/mirrexa/swarmfile-server`, tagged `latest` and the release version), and the server repository carries a `compose.yaml` that uses it:

```
docker compose up -d
docker compose exec server swarmfile-server admin --data-dir /data create-org --id my-org --name "My Org"
docker compose exec server swarmfile-server admin --data-dir /data create-repository --org my-org --name "My Repository"
docker compose exec server swarmfile-server admin --data-dir /data mint-key --org my-org --repository <repository-id> --name cli
docker compose exec server swarmfile-server admin --data-dir /data enable-git --org my-org --repository <repository-id> --threshold-bytes
50000000

export SWARMFILE_HUB_URL=http://127.0.0.1:8080
export SWARMFILE_API_KEY=sf_key_…          # the key minted above
git clone swarmfile://my-org/<repository-id>
```

The clone needs the standalone `git-remote-swarmfile` helper (install); the server's own `GET /api/profile` is served on `http://127.0.0.1:8080/api/profile`. Set `SWARMFILE_SERVER_VERSION` to pin the image, or build the source Dockerfile when you need a local build. `/data` is a named volume; back it up. The GHCR package must be public for an anonymous pull.

Figure omitted in print (animation): Self-hosting in Docker from the published snapshot's own Dockerfile: the container holds only the binary, the volume is the entire server state, and the open doctor verifies it from the host. - https://swarmfile.com/demo/oss-docker.cast

From source: `cargo build --release -p swarmfile-server --features s3` in the published repository (the `s3` feature is optional but the release binaries and images include it).

First run

```
swarmfile-server admin --data-dir ./data init
swarmfile-server admin --data-dir ./data create-org --id my-org --name "My Org"
swarmfile-server admin --data-dir ./data create-repository --org my-org --name "My Repository"
swarmfile-server admin --data-dir ./data mint-key --org my-org --repository <repository-id> --name desktop
swarmfile-server serve --data-dir ./data --listen 127.0.0.1:8080
```

`create-repository` prints the repository id, which every command that follows names it (`--org` and `--repository` also accept a live or historical slug). `mint-key` prints a key (`sf_key_…`) exactly once; treat it like a password. A repository's encryption tier is chosen when it is created: `--tier plaintext` (the default) stores bytes as written, `--tier managed` keeps a key on the server so authorized clients encrypt and decrypt through it.

Renaming keeps published addresses working:

```
swarmfile-server admin --data-dir ./data rename-repository --org my-org --repository <repository-id-or-slug> --slug new-name
swarmfile-server admin --data-dir ./data rename-org --org my-org --slug new-org
```

`admin list-aliases --org my-org` prints the historical org and repository slugs, so an operator can audit what old URLs still resolve. `rename-repository` and `rename-org` also take `--name`.

The old slug is recorded as a reserved alias, so an existing `swarmfile://<host>/<org>/<repository>` URL or a git remote keeps resolving to the same repository (the community server has no anonymous public pages or share links; those are hosted-only); a live slug always wins, and another repository cannot create or rename onto a historical name. Renaming back to an old slug frees it again. (The community server has no repository or org delete surface, so a historical name is never released by a delete.)

A `serve` process holds an exclusive lock on its data dir, so there is **one server instance per data dir**: a second `serve` refuses at startup (the admin CLI may run alongside it).

Serve plain HTTP and put a TLS reverse proxy in front (Caddy or nginx); the client tools expect `https://` outside a purely local setup. Set `SWARMFILE_SERVER_PUBLIC_URL` to the public origin (for example `https://files.example.com`): the server's signed host identity (`.well-known/swarmfile.json`) signs that configured origin, never the request `Host` header.

The document is origin-root by design (RFC 8615): a deployment served under a path prefix must map `/.well-known/swarmfile.json` at the root for verifiers to find it. Without it the identity document answers 503; `swarmfile-verify-history --check-host-identity` then fails closed.

Connect the tools

- **Desktop app** - install the self-hosted edition from your server's bundle (the `swarmfile-self-hosted-<version>-<platform>` download on the swarmfile-server release page), then choose **Connect to your server** and enter the server's address and the key. If the key opens more than one repository, the app asks which one. The drive, listing and saves work the same as against the hosted service. The first connect asks you to accept the edition's license (`EULA.md` in the bundle).
  - This edition does not update itself, and **Reinstall** and `swarmfile-doctor --repair` refuse on it rather than replace it with the hosted app: upgrade by installing the next bundle, server first. **Disconnect from server** forgets the key on that computer.
  - A key set by the `SWARMFILE_API_KEY` environment variable can't be changed from the app: **Connect** and **Disconnect** say so and leave it to whoever set the variable. An address set by `SWARMFILE_HUB_URL` is fixed the same way: **Connect** says so. The desktop app from swarmfile.com always connects to the hosted service.
  - On Windows and Linux the two editions can't be installed together: each installer refuses while the other is installed, so uninstall one first. On macOS they install side by side as separate apps.

  Each bundle holds, for one platform, the server (`server/`), the self-hosted desktop installer (`desktop/`), the standalone tools (`tools/`), the license, the third-party notices and a `versions.json` naming every component's release. A bundle is published only after the community end-to-end tests have passed for that exact version set.
- **git-LFS**: a repository serves git-LFS under its own path; set it in the repo's `.lfsconfig`:

  ```
  git config -f .lfsconfig lfs.url https://your-server/orgs/my-org/lfs/<repository-id>
  ```

  Stock `git-lfs` works for ordinary objects; the bundled agent handles very large ones. Native ingest is plaintext-only on the community server: on a managed repository both are refused `422`, because the server stores only sealed blocks and will not write unsealed content at rest, so move large files through the mounted drive there. See Git-LFS for how the two fit together.
- **git clone** - turn on a repository's git access (one-way, as on the hosted service), then clone over the helper:

  ```
  swarmfile-server admin --data-dir ./data enable-git --org my-org --repository <repository-id-or-slug> --threshold-bytes 50000000
  git clone swarmfile://your-server/my-org/<repository-id>
  ```

  Set `--threshold-bytes` above the largest blob the repository already contains before the first enable: the value is frozen, and a push refuses a literal blob at or above it. The default is 1 MiB. Name the server (the host-qualified form) in anything you hand to a teammate: a relative `swarmfile://my-org/<repository-id>` resolves against whichever hub that machine is configured for.

  If the server is plain HTTP rather than behind the TLS proxy above, the client must list its authority in `SWARMFILE_GIT_INSECURE_HOSTS` (loopback needs no entry). A teammate whose Desktop App is connected to a **different** hub presents no session here: connect the app to this server, store the server key for this host with `git credential approve`, or set `SWARMFILE_API_KEY` together with `SWARMFILE_ALLOW_CROSS_HUB_KEY=1`.

  An owner key can do the same over the API: `POST https://your-server/orgs/my-org/repositories/<repository-id>/metadata/git-view/enable` with the key as a bearer token.

  Cloning here always presents a key: the **anonymous public clone** is a hosted-service feature, and the community server's git surface is member-scoped (no public gates, no clone ticket).

  See Cloning a Repository with Git for what a clone contains and how push works. On the community server there is no background derive worker: a push confirms its commits itself as it lands, and if a push is interrupted, run `swarmfile git index` from a mounted drive to confirm the commits it left undecided.
- **Existing data**: bring a local tree or object store across with the standalone migration tool (swarmfile-migrate): `swarmfile-migrate --hub-url https://your-server --org-id my-org --repository-id <repository-id> --token <key> --source ./tree`. See Migrating Existing Data.
- **Place a single file**: the standalone `swarmfile-seed` (swarmfile-seed) puts one local file at a repository path, no mount: `swarmfile-seed --hub-url https://your-server --org-id my-org --repository-id <repository-id> --token <key> --file ./build/app.zip --path /Releases/app.zip`. See Seeding a File Into a Repository.
- **Mirror a repository**: `swarmfile-mirror` (swarmfile-mirror) copies a repository (or a subtree) to another, e.g. off the hosted hub onto yours: give it a `--source-*` and a `--dest-*` hub/org/repository and token. See Mirroring a Repository Between Hubs.
- **Check connectivity**: the open `swarmfile-doctor` (swarmfile-doctor) confirms DNS, TLS, your key, the block path and your server's descriptor before you rely on it. See Diagnosing Connectivity.

These open tools each install on their own (`curl -fsSL https://get.swarmfile.com/{seed,mirror,doctor} | sh`) and share the migration tool's pipeline: an API key, no engine, no mount.

Figure omitted in print (animation): The standalone swarmfile-migrate moves an existing tree into a repository on a community server: the first pass is killed mid-flight, the same command again resumes exactly where the state file stopped — only the files that never landed move — then re-chunks every source file and checks it against its manifest. - https://swarmfile.com/demo/oss-migrate.cast

Keys and access

```
swarmfile-server admin --data-dir ./data list-keys
swarmfile-server admin --data-dir ./data revoke-key --key <id|name|prefix>
```

Each key is scoped to one repository, or to the whole organization when it is minted without `--repository`. A revoked key is refused on its next request; revoke and mint a replacement to rotate.

On a community server a key stands for a person, so give each teammate (and each CI job) their own. Every key has a role:

- **owner** (the default for `admin mint-key`, and what keys minted before roles existed became): manages every key in the organization, and its commits are recorded as confirmed history straight away.
- **member** (`admin mint-key --role member`): manages only its own keys and the keys it minted, and its commits wait for another person's key to confirm them, the hosted service's two-person rule. Keys a member mints count as that member, so a CI key is a good member key: it cannot write confirmed history on its own.

Keys can also be managed without the admin CLI: `swarmfile api-key list`, `swarmfile api-key create <name>` and `swarmfile api-key revoke <id>` work against the server. A key created that way is a member key for one repository; an owner key can create another owner key. A key scoped to one repository, an owner's included, manages only that repository's keys. Revoking a key that has been used also rotates the organization's swarm secret, like `admin revoke-key`.

Scoped branch/path keys are a hosted-hub feature: the community server rejects the scope fields (`scoped_keys_not_supported`), so use an ordinary repository/org key there until P1.

Locks belong to the key that took them. A `git push` that changes a file locked by another key is refused (`423 locked`) until that lock is released; your own lock never blocks your own push, and only the key that took a lock (or an owner key) can release it. The same goes for a folder reservation: changelists and pushes that touch a folder another key has reserved are refused until the reservation ends.

Exposing it to your team

The server has no browser UI: the tools are native apps and CLI, so an ordinary LAN/TLS setup needs nothing extra. A few knobs exist for the rest:

- **Git surface.** Git serves clone and push by default. A mirror or read-only replica can narrow it with `SWARMFILE_SERVER_GIT_MODE` (or `swarmfile-server serve --git-mode`): `read` serves clones only (pushes, branch/tag writes and git-view enablement refuse with a typed `403 git_read_only`, while cloning an unsettled repository still derives and mints its tip), and `none` turns git off entirely (`403 git_disabled` on every git route). It gates only the git surface; metadata, locks, LFS and blocks stay served, and `GET /api/profile` reports the mode as `features.git`.
- **Browser clients.** CORS is off by default. Set `SWARMFILE_SERVER_CORS_ORIGINS` to a comma-separated list of the origins allowed to call the API from a browser (or `*` to allow any); without it a browser tool cannot read any response.
- **Addressing.** Every route accepts the org and repository **slugs** wherever it accepts ids (`https://your-server/orgs/my-org/repositories/my-repository/…`), so links written with names keep working. Slugs are derived from the name at creation and can be set with `--slug`.

Peer-to-peer: the swarm

The server also runs the swarm's rendezvous: desktop apps pointed at it join your organization's private peer-to-peer network, so teammates' machines stream file bytes to each other **LAN-first**: one machine fetches a block from the server and the rest of the team fetches it from that machine, and live edits gossip between them instead of waiting for a poll.

This is on by default; nothing needs configuring on the clients. A client that can reach a teammate only through the public relay falls back to your server by default; `SWARMFILE_ALLOW_RELAY_FETCH=1` on both serving and fetching engines opts into relayed block transfer (throttled).

Figure omitted in print (animation): Two Macs on one LAN share a branch through the community server's swarm: a save here reaches the other machine in seconds, its stale save is refused, and an 8 MB render is then fetched peer-to-peer — 8.0 MiB from the colleague's machine, 0 bytes from the server. - https://swarmfile.com/demo/oss-swarm.cast

`SWARMFILE_SERVER_P2P` (or `serve --p2p`) picks the posture:

value	what it means
`all` (default)	Full peer-to-peer, and the swarm may carry **cleartext**: plaintext-tier repository blocks, and cleartext stored on managed repositories (LFS objects), included. The swarm is exactly the machines holding authenticated keys to *your* server. Note the swarm is per-organization: any org key holder's machine can fetch any repository's blocks by content id, without the per-repository scoping the HTTP routes apply; choose `sealed` if that matters to you.
`sealed`	Full peer-to-peer, but only sealed (managed-tier, encrypted) blocks travel between machines; cleartext is never served to peers, including the LFS objects managed repositories store in the clear. (A machine may still cache cleartext blocks locally; it just won't hand them out.) Choose this if you want cleartext never to leave a machine over the swarm.
`off`	No peer-to-peer: no discovery, no peers; every read comes from the server.

The posture is read when an engine starts, on repository switches, and by every running engine's periodic server check, about every 5 minutes for desktop apps and headless seeds/runners alike (set `SWARMFILE_DOCTOR_PERIOD_SECS=0` on an engine to disable that check; then only a restart applies a change). A server that narrows its swarm therefore stops its running engines serving, without waiting for a restart. To cut a whole fleet's swarm instantly, rotate the secret and restart the clients.

How membership works: on the first request the server mints a per-organization swarm secret and hands it out only over the authenticated discovery route, so the swarm is exactly your team's engines. To remove a machine from the swarm, revoke its key; the server rotates the swarm secret in the same command:

```
swarmfile-server admin --data-dir ./data revoke-key --key <id|name|prefix>
```

Engines pick up the rotated secret within about 90 seconds; the replaced secret stays accepted during a short overlap so nobody drops mid-read. Until a rotation happens, a machine that still holds the current secret keeps swarm access: the rotation is the cutoff.

Use `--no-rotate` for housekeeping revokes (a build-robot key that never joined the swarm); run `admin rotate-swarm-key --org <id>` by hand when you want a rotation without a revoke. `admin swarm-status --org <id>` prints the current generation, how long the old secret is still accepted, how many engines are live, and the cumulative peer-to-peer bytes the engines have reported.

What the swarm needs on the network: plain QUIC/UDP between your machines (engines bind a random local UDP port and advertise it through the server, so allow UDP egress), multicast for same-LAN discovery (mDNS), and, for teammates on different networks, the public iroh relay that lets two machines behind NAT meet. The relay is for meeting/address discovery: block bytes are not sent over it by default, so a relay-only peer also falls back to the server (`SWARMFILE_ALLOW_RELAY_FETCH=1` opts in, throttled). Block UDP entirely and the swarm quietly degrades: every read falls back to the server.

Pin the bind address with `SWARMFILE_IROH_BIND_ADDR`, advertise a stable one with `SWARMFILE_IROH_PUBLIC_ADDR`, or turn the swarm off on a single machine with `SWARMFILE_HUB_ONLY=1`. To check a machine, `swarmfile doctor` prints the server profile (look for the `p2p=` / `p2pPlaintext=` fields) and probes iroh and mDNS.

What the community server does not do here: it does not sign change attestations (that trust layer is hosted-only), and it has no per-node approval list: membership is the swarm secret itself.

History

Commits and autosave

Every change becomes a commit, the same as on the hosted service. The desktop app commits each burst of saves on its own once the drive has been quiet for a few minutes.

Changes that arrive any other way are committed by the server itself once the branch has been quiet for five minutes: an import with `swarmfile-migrate` or a file placed with `swarmfile-seed`. While a desktop app is in the middle of its own commit the server waits twice as long, so a desktop app that went offline before committing is covered after ten minutes. Those commits are authored `server:autosave` and read `Saved N files`; `git push` records your own commits as they are.

Set `SWARMFILE_SERVER_AUTOCOMMIT_IDLE_SECS` to change the quiet window (in whole seconds, 1 or more), or to `off` to leave such changes uncommitted until the next push or save.

Changelists

Staged changelists work too: `swarmfile changelist open`, then save, then `swarmfile commit -m` (or `changelist submit`) lands the staged files as one commit, and nothing reaches the branch before that. `swarmfile describe` renames a commit afterwards.

Open sessions and changelists with no activity for seven days are closed automatically, as on the hosted service.

Conflicts and locking

When two people save the same file at the same time, the version that reaches the server first is kept and the other save is recorded as a conflict, as on the hosted service. That person's drive keeps their version, readable but read-only, and `swarmfile conflicts` names the file.

`swarmfile conflicts resolve <path>` settles it: `--winner local` keeps their version, `--winner remote` keeps the one on the server, `--winner keep-both` keeps both (theirs is saved alongside as `name (conflict <user> <date>).ext`), and `--auto` merges text files; the desktop app offers the same choices.

Until it is settled, saves to that file are refused from every computer, and deleting the file settles it. A `git push` that changes the file is refused until it is settled too. Locking the file first (`swarmfile lock`, or **Lock** in the desktop app) avoids this: while the lock is held, saves from other computers are refused, and so is a `git push` from any other key, until it's unlocked.

Storage and backups

By default everything lives under the data directory: `system.db`, one SQLite file per repository, the block store, and `history-signing.key` — the server's own Ed25519 checkpoint key, created on first use with owner-only permissions (0600 on macOS/Linux; on Windows the file takes the data directory's inherited ACLs). The server signs a checkpoint of each branch's confirmed head with it, and `swarmfile-verify-history` checks those signatures from outside.

The key is part of the server's identity: back it up with the directory (a restored server keeps verifying), and replacing it starts a new identity — checkpoints signed by the old key stop verifying, and the replacement is loaded on the next restart.

To keep block bytes in an S3-compatible bucket instead (metadata stays local), set `SWARMFILE_SERVER_S3_ENDPOINT` and `SWARMFILE_SERVER_S3_BUCKET`, plus `SWARMFILE_SERVER_S3_ACCESS_KEY_ID` / `SWARMFILE_SERVER_S3_SECRET_ACCESS_KEY` (`SWARMFILE_SERVER_S3_FORCE_PATH_STYLE` defaults to path-style, so MinIO-style endpoints work as-is; set it to `0` only for virtual-host-style endpoints; `SWARMFILE_SERVER_S3_REGION` for buckets outside `us-east-1`; `SWARMFILE_SERVER_S3_PREFIX` namespaces keys in a shared bucket).

This needs a server built with the `s3` feature, which the release binaries and the container image are. The bucket must already exist.

Back up the data directory (and the bucket, if you use one): that is the whole server.

Figure omitted in print (animation): Day two on a community server: mint a key, prove it works, revoke it with one command — the org's swarm key rotates with the revocation — then the backup drill: stop the server, copy its one data directory, and serve the copy. The restored server passes the same doctor checks. - https://swarmfile.com/demo/oss-server-ops.cast

Copying a repository with exact history (optional)

A same-server copy duplicates a repository's database, hub-held key and block/LFS/git-object data into a new repository. It is the one operation that preserves **managed** commit history exactly, because nothing is re-sealed. It is deliberately opt-in and the **real** copy runs with the server **stopped** (the command takes the serve lock and refuses while `serve` is running; `VACUUM INTO` would otherwise block writers). `--dry-run` does not create or copy repository data or keys (normal startup schema maintenance may still run) and may run while serving. A detailed operator checklist ships with the server source:

```
# See exactly what would happen; writes nothing.
swarmfile-server admin --data-dir ./data copy-repository \
  --org my-org --repository <repository-id-or-slug> --name "Copy" --slug copy --dry-run

# The real copy; prints the new id, head, key and object counts.
swarmfile-server admin --data-dir ./data copy-repository \
  --org my-org --repository <repository-id-or-slug> --name "Copy" --slug copy
swarmfile-server admin --data-dir ./data mint-key --org my-org --repository <new-id> --name copy
```

What it does and does not do:

- It copies the repository DB (entries, branches, commits/claims, changesets), the hub-held repository key (`managed`), and the `p/`, `lfs/` and `o/` data prefixes. The commit DAG, head hash and content CIDs are byte-identical; the source is never modified.
- It is a **point-in-time** copy; later writes to the source are not followed. It is **not a mirror**, does not cross servers or hubs, and is **not a backup**: it lives in the same data directory and block store, so a disk/store loss loses both copies. Transient locks, conflicts and checkouts are **not** carried; the copy opens clean.
- A crash during the copy can leave a marker-named destination (`__r219copy-…`) and its data prefixes under a `*.copy-<nonce>` staging name. The source is unaffected; the next `serve` start or `copy-repository` removes marker rows, staging prefixes and their DB/key automatically. Inspect the server log for the cleanup line before retrying.
- It requires the **filesystem block store**; a configured S3 block store refuses before touching metadata (prefix copy is not implemented there yet).
- API keys are not copied (they are scoped and hashed); mint a destination key as above.
- The community server has no E2E tier (only `plaintext` / `managed`); E2E key sharing is a hosted/cross-hub gap and is not copied by this command.
- A pre-existing destination block prefix is refused (no silent merge), and a failed copy rolls back the destination metadata and copied prefixes.

Federating public repositories

Two allowlisted hubs can serve **public plaintext** repository data through an allowlist, without copying bytes: one server (the *origin*) exports a repository to a named peer, and the other (the *federator*) pins that peer's signed host identity and allowlists the repository.

The hosted Swarmfile service can also be either side; its side is managed by Swarmfile staff (peer pin/import) and, for a hosted origin, the repository owner's consent in **Repository settings** on the web (Public repositories → Hosted federation export). The commands below are the community-server side. A federated clone then reads through the federator:

```
git clone swarmfile://server-a.example/fed/server-b/acme/widget
```

Set `SWARMFILE_SERVER_PUBLIC_URL` on each community server you run first (the signed origin-root `/.well-known/swarmfile.json` must be reachable, as described under Connect the tools). On the **origin** (`server-b`), pin the federator and export the repository — the repository must be plaintext, its git access enabled, and the export named by the right slug:

```
swarmfile-server admin --data-dir ./data federation peer add   --name server-a --origin https://server-a.example
swarmfile-server admin --data-dir ./data federation repository allow --export   --peer server-a --org acme --repository widget
```

On the **federator** (`server-a`), pin the origin and allow the remote repository. The command fetches the origin's signed identity and its public clone gate, so the origin must already be exported and serving:

```
swarmfile-server admin --data-dir ./data federation peer add   --name server-b --origin https://server-b.example
swarmfile-server admin --data-dir ./data federation repository allow   --peer server-b --org acme --repository widget
swarmfile-server admin --data-dir ./data federation repository list
```

`federation peer list` shows the pinned origin and key id; `federation repository list` shows each row's role (`proxy` on the federator, `export` on the origin), whether its actor signature verifies, whether the actor is `local` or `imported` from the peer's host-signed `/.well-known/swarmfile-federation.json`, and the export's effective auth mode.

The import is explicit and two-step: export locally, let the federator allow the proxy, then re-run the export with `--import-actor`; the federator's advertisement must verify against its pinned host identity, and no private key is copied.

`peer disable` stops all of a peer's proxying and exports immediately (the pin stays for audit); `peer remove` then drops a disabled peer and its allow rows (the recovery path after a permanent host-key rotation); `repository deny` removes one allow row.

The per-repository **actor key** is generated at allow time (or imported from the peer's signed advertisement with `--import-actor`), advertises the opt-in record, and is separate from every content key (which stays null for plaintext) and from the server's history-signing key.

Signed peer reads (optional)

An export is public-by-URL by default. To require a per-request signature from the federator's per-repository actor key, run this on the **origin** after the actor is imported and the federator has allowed the proxy:

```
swarmfile-server admin --data-dir ./data federation repository allow --export \
  --require-actor --peer server-a --org acme --repository widget
# later, back to anonymous public reads:
swarmfile-server admin --data-dir ./data federation repository allow --export \
  --allow-anonymous --peer server-a --org acme --repository widget
```

- The requirement is effective only when **every enabled export** of the repository requires an actor; any anonymous export keeps the repository public (`federation repository list` shows `effectiveAuth`). The origin cannot attribute an anonymous request to a peer, so this is deliberately weakest-export-wins.
- The federator signs every proxied repository read with the actor key (`swarmfile-fed-request-v1` : method, origin path + query, body hash, timestamp, nonce, hop chain); it also signs `_resolve` when it has that repository's proxy actor key, and a configured proxy row with a missing secret fails closed. The origin checks a ±60 s timestamp, a short nonce replay cache and a per-actor budget (`SWARMFILE_SERVER_FEDERATION_ACTOR_RATE_PER_MIN` , default 6,000/min). An unsigned, stale, replayed or wrongly signed request gets the **same 404** as a repository that is not exported.
- A signed POST (the LFS batch and oid-coverage) hashes the exact forwarded body; an unsigned POST to a required repository is refused before its body is read.
- The setup order is: export (anonymous) → federator `repository allow` → origin `--import-actor` → origin `--require-actor` . A federator re-allow after that signs its own gate check; a brand-new federator allow cannot, so switch the requirement on only after the first allow.
- Direct public links stop working while every export requires an actor. That is an access boundary, not a secrecy one: the data is still public plaintext, and a peer that is allowed to read it can.

Public activity feed

Every exported repository (and, with signed reads, every allowlisted peer) can read a metadata-only activity feed:

```
GET /public/<org>/<repository>/activity?limit=50&before=<seq>
```

It returns `{org, repository, generatedAt, branches, commits, nextBefore}` ; commit messages are capped at 4 KiB (`messageTruncated` says so) so one huge message cannot make the feed unbounded.

`branches` are the live public refs with provisional hashes withheld; `commits` are recent AUTHORITATIVE committed changesets (`seq` , `hash` , `status` , `treeCid` , `parents` , `message` , `committedAt` , `createdAt` , `branch`) — provisional/claim-only rows are filtered out, path-like tokens matched by the repository's `.swarmfile/publicignore.yml` are redacted from messages, and there are **no file contents and no author identities**.

`limit` is 1–200 (default 50); pass `nextBefore` as `before` to page older commits. It uses the same export/auth gate as the clone and is carried through a federator at `/fed/<peer>/.../activity` . It is not a public directory: the repository URL is still unlisted.

What this does and does not do:

- **Only public plaintext data.** Managed/E2E repositories, ACLs, member data and keys are never federated. git-LFS **downloads** are served through the federator; LFS uploads and locking stay member-only. Exporting makes the repository's public reads answerable to anyone who knows the origin's `/public/…` URL (unless every enabled export is actor-required) — the data is public and the URL is unlisted, not secret. Do not export private data.
- **The committed policy gates the public surface.** The origin's `/public` reads (and therefore a federated clone) honor the `.swarmfile/publicignore.yml` committed in every served tree.
  - A slash-free entry matches a name at any depth, a path containing a slash is relative to the repository root, and a trailing slash matches directories only.
  - A policy that hides a path refuses the **whole** public surface with the same 404 as a non-exported repository — there is no filtered clone — and a scan that cannot prove the repository clean (read, byte, ref, depth or wall-clock budget) refuses the same way.
  - A glob or a `!` entry is not evaluated (its absence cannot be proven) and also refuses.
  - A change takes effect within about a minute (the verdict is cached for up to 60 seconds). The activity feed redacts matched path names in commit messages; a clone does not.
- The federator acts as a **GET/HEAD proxy plus two read-only LFS POSTs** (`git/oid-coverage` and the download-only `git/lfs/objects/batch` ; upload batches are refused) to the pinned origin's public reads.
  - It refuses redirects, strips client credentials/cookies and hop-by-hop headers, caps response size and time (`SWARMFILE_SERVER_FEDERATION_MAX_BYTES` , `SWARMFILE_SERVER_FEDERATION_TIMEOUT_SECS`), rate-limits per peer/repository (`SWARMFILE_SERVER_FEDERATION_RATE_PER_MIN` , default 600 per minute; raise it for large clones, which fetch many blocks) and writes one `[federation]` audit line per request.
  - The origin has its own per-repository budget (`SWARMFILE_SERVER_FEDERATION_ORIGIN_RATE_PER_MIN` , default 6,000/min, 429 + `Retry-After: 60`), which a direct anonymous reader also consumes; keep that in mind when sizing. It caches no bytes and never re-signs the origin's history.
- The peer's signed identity is re-verified against the pin every `SWARMFILE_SERVER_FEDERATION_IDENTITY_TTL_SECS` (default 60). Restoring the original key keeps serving; a **replaced** key fails closed until the operator pins it: `federation peer disable` , `federation peer remove` , then `federation peer add` with the new statement.
- A rename on the origin keeps old URLs working through its slug alias, but a federator allow row names a slug: using the new slug requires `federation repository allow` again on the federator. The federated URL is explicit (`/fed/<peer>/…`); the federator never owns or aliases the origin's namespace. `repository deny` keeps a proxy actor secret so a re-allow can sign a require-actor origin's gate; removing the peer clears it.
- The federator's host-signed `/.well-known/swarmfile-federation.json` lists every **proxy** row it federates (remote origin, org, repository and the public actor key/signature). It is anonymous and signed, but it does publish the federation relationship; the current document is host-scoped (every proxy row) and a peer-scoped advertisement is not built. Public repository data and URLs are already public by design.
- **Mesh routes are experimental and disabled by default.** A federator can point a peer at another federator's `/fed` path (A→B→C) only when every participating server sets `SWARMFILE_SERVER_FEDERATION_MESH=1` ; without it, path-scoped peers, signed `/fed` hops and two-plus-hop reads are refused. The intermediate `/fed` is not a published endpoint.

  Do not share that URL, do not rely on anonymous reads through it, and expect a future per-reader tightening. The supported public contract is `/public` , plus your own federator's client-facing `/fed` URL. The intermediate's opt-in flags (`allow_transitive` on the middle proxy, `allow_reexport` on the origin) are prerequisites for that future tightening, not a per-reader guarantee today.
- This is not ForgeFed/ActivityPub, not private/E2E federation, and not a mirror. Hosted/cross-hub managed copy and E2E key sharing are owner-deferred and not built; hosted public-data participation is the operator/owner-managed path above. The open-source server's `test:federation` lane proves the two-server slice end to end.

Upgrades

Stop the server, replace the binary or image, start it again: the schema migrates on first use. **This repository rename is the exception:** there is no data migration from a pre-rename data directory. A `projects/` data dir or a system DB with a `projects` table is refused at open with an actionable error; start from a fresh data directory (keep the old one as a backup) rather than expecting an in-place upgrade. A data directory (or repository) that a newer server has opened refuses to open on an older one, so downgrades need a backup rather than a rollback.

What the community server leaves to the hosted service

- Access rules and external collaborators, branch merges, the commit-level `revert` and `restore` operations, repository management (archiving, retention and the other repository settings beyond the administrator's `admin create-repository` / `admin rename-repository`; as in First run).
- End-to-end encryption (refused by design: the server can never hold those keys), previews, share links, **anonymous drops** (the `/send` public-link upload surface), RFIs, notification emails and webhooks, hosted CI, the Office caches feature set (the enrollment wizard, fleet dashboard and reporting; a bare seed node runs against this server too).
- Hosted agent identity and delegations (the `/agents/v1` surface) and service-minted agent tokens. A community server does mint short-lived TTL member keys (`POST /orgs/:org/api-keys` with `ttlSeconds`), but that is not a hosted delegation or a service-minted agent token. Multi-organization hosting and billing are also hosted-only.

The branch archive and batch checkpoint are available on the community server; see Known limitations for their community caps. If your team needs the hosted-only features, the hosted service keeps them. The same clients work with both, so you can start yourself and move when it helps.

The clients say so rather than failing. Any request for one of these gets the same refusal from the server itself, so a command with no special handling still prints why it was refused. Where the desktop app reaches one of these (the **Repository settings** panel, **New repository**, the CI and notification settings, a share-link or RFI link, the plan usage card), it shows one line saying the feature isn't available on this server, with a link back to this section.

The CLI refuses the same commands with the code `not_supported_on_this_server` and adds a `hint:` line pointing here (a `hint` field under `--json`); `swarmfile repository create`, `git import` and `clone --into` name the `admin create-repository` command.

The server has no web dashboard either, and its profile says so, so the desktop app's links into one (**Open in browser**, **Team**, account settings, the full notification inbox, a file's web link, a release's public URL) explain that in one line instead of opening the hosted site, and the tray menu drops **Open on the Web**.

Seeding a File Into a Repository

`swarmfile-seed` places one local file at a path inside a repository over the public API — without mounting a drive. Use it from a script or a CI job to drop a release artifact, a fixture, or a large asset where you want it.

It chunks the file, seals each block client-side when a key is configured, and registers the entry — the same pipeline the desktop mount and the migration tool run. The file is streamed, so a multi-gigabyte source does not have to fit in memory.

This guide covers the **standalone open-source build** of `swarmfile-seed` (API-key auth, every target given as a flag), published at swarmfile-seed under MIT/Apache-2.0. The Desktop App also puts a `swarmfile-seed` on `PATH`, but that bundled build is a different tool: it takes its repository and credentials from the environment and has no `--org-id`, `--repository-id`, `--token` or `--branch` flags. Its reference is `swarmfile-seed`.

On a machine with both, the first one on your `PATH` wins; `swarmfile-seed --help` tells them apart (the standalone's summary says "standalone, API-key auth").

Install

```
curl -fsSL https://get.swarmfile.com/seed | sh
```

The install script downloads the release for your machine, verifies its checksum, and installs `swarmfile-seed` into `~/.local/bin` (or `/usr/local/bin`), warning if that directory is not on your `PATH`. A binary archive and the source are on the swarmfile-seed public repository.

Use

Set a token, then point it at a file and a destination path:

```
export SWARMFILE_API_KEY=sf_pat_…          # a personal access token, or an org API key

swarmfile-seed \
  --hub-url https://hub.swarmfile.com \
  --org-id my-org --repository-id <repository-id> \
  --file ./build/app-2.1.0.zip \
  --path /Releases/app-2.1.0.zip
```

- **`--path` is the destination inside the repository.** Missing parent directories are created. The file name is the last path segment.
- **`--token`** passes the token directly; it defaults to `$SWARMFILE_API_KEY`.
- **`--org-id`** accepts the org's id or its slug.
- **`--branch`** targets a non-default branch; omit it for the hub's configured default.
- **Chunking.** Fixed-size by default. `--cdc` uses content-defined chunking (fast, dedup-friendly for large files that change little); `--auto-profile` picks a per-type profile (the same AEC and media profiles the mount's own writes use). `--cdc` and `--auto-profile` are mutually exclusive.

Figure omitted in print (animation): Engine-free and mount-free: swarmfile-seed places one local file at an exact repository path, and swarmfile-mirror copies a subtree from one repository into another — the same chunk, seal and upload pipeline the whole ecosystem shares. - https://swarmfile.com/demo/oss-tools.cast

What it does when something is already there

The hub treats a create at an existing name as get-or-create — it will not overwrite. If a file with the same name but **different content** already sits at `--path`, `swarmfile-seed` fails loudly rather than reporting a false success; delete the existing entry or pick another path. An **empty file** registers as a zero-byte entry (no content address), the same way the mount represents one.

Sealing

With a managed repository, the hub-held key is fetched and every block is sealed before upload, so the hub never sees plaintext. An `SWARMFILE_ENCRYPTION_KEY` environment variable is used as-is — supplying it also lets you seed an end-to-end-encrypted repository. With no key at all, the file is stored plaintext.

Exit codes

`0` success; `1` a failure, including a bad `--path` or a refused upload; `2` a flag/usage error from the argument parser. A refusal is on stderr.

Mirroring a Repository Between Hubs

`swarmfile-mirror` copies a repository — or a subtree — from one Swarmfile hub to another. The common case is moving off the hosted service onto your own community server, or keeping a second copy on a self-hosted box.

It is the repository-to-repository counterpart of the migration tool: `swarmfile-migrate` brings a local tree or an S3 bucket *in*, and `swarmfile-mirror` copies a *repository* to *another repository*.

Install

```
curl -fsSL https://get.swarmfile.com/mirror | sh
```

Use

Give it a source and a destination — each with its own hub, org, repository and token:

```
export SWARMFILE_SOURCE_API_KEY=sf_pat_…    # source hub token
export SWARMFILE_DEST_API_KEY=sf_key_…      # destination hub token

swarmfile-mirror \
  --source-hub-url https://hub.swarmfile.com \
  --source-org-id my-org --source-repository-id <id> \
  --dest-hub-url https://your-server \
  --dest-org-id my-org --dest-repository-id <id> \
  --path /Repositories --branch main
```

- **`--path`** mirrors a subtree (default: the whole repository).
- **`--branch`** selects the branch to read on the source; the destination is written to its default.
- **`--exclude <glob>`** skips entries by name (repeatable, e.g. `--exclude '*.tmp'`).
- **`--dry-run`** lists what would be copied and touches nothing.
- **`--state <file>`** records each mirrored source path → (cid, size) and **resumes** on a re-run: a file already mirrored with the same source content is skipped, so an interrupted copy of a large repository picks up where it left off.
- **`--json`** prints a machine-readable summary (files, bytes, dirs, symlinks, resumed, skipped).
- **`--concurrency <n>`** (default 4) copies up to `n` files in flight; `1` is sequential. It's ignored with `--state` (resume bookkeeping is ordered).
- **Chunking** follows `swarmfile-seed`: fixed-size by default, `--cdc` or `--auto-profile` to change it. The provenance mode below keeps the source's chunks instead.
- **`--provenance-out <file>`** writes the origin's portable provenance manifest and, for a plaintext whole-repository mirror, preserves the source's block CIDs and entry **lineage** so the destination's tree is identical. It requires an empty destination branch (or a matching resume state), a confirmed/quiet source, no `--exclude` / `--path`, and a credential that can read the whole tree; sync-ignored entries are included. `--concurrency`, `--cdc` and `--auto-profile` are ignored in this mode (the source's chunks are copied verbatim). See Provenance.

Figure omitted in print (animation): Two community servers, and the standalone swarmfile-mirror copies a repository between them over the HTTP API: run it again and the state file resumes the copy, re-uploading nothing, and the destination then passes the same doctor checks as the source. - https://swarmfile.com/demo/oss-two-hubs.cast

What it copies

Files, directories and symlinks. A file is re-encoded for the destination — chunked and sealed under the **destination's** key — so a plaintext or managed source lands correctly on a plaintext or managed destination. (The provenance mode above copies the source's exact chunks instead, so its tree matches.) Deleted and trashed entries are not copied.

Two refusals protect you, matching the migration path:

- a same-name/different-content collision on the destination is an **error**, not a silent skip; and
- a **symlink with an absolute target** is refused (it would not resolve on another machine).

Provenance

A mirror is **not a signed replica**: the destination has its own commits, and `swarmfile-verify-history --mirror-*` only matches a replica that preserves the origin's confirmed head **commit** (the same commit DAG). The verifier checks commit/tree identity, not that every block is served.

`--provenance-out <file>` makes a **plaintext, whole-repository** mirror verifiable offline. The mirror verifies the origin's signed history (confirmed head, commit chain, every checkpoint, key ring, and its signed host-identity statement when served), fetches the confirmed head's canonical tree and compares the LIVE source listing to it (refusing a partial/ACL-filtered listing, ignored-entry omission or live-uncommitted write), copies the source's block CIDs/manifests/entry **lineage**, then compares the destination's live listing to the same tree.

It also pins a resume `--state` to the source head, so a resumed run after the origin advanced is refused. Add `--dry-run` to perform the source tree/listing comparison only: it prints the result, writes no manifest and leaves the destination untouched. The destination's tree — and usually its confirmed head commit — matches:

```
swarmfile-mirror … --provenance-out origin.json
swarmfile-verify-history \
  --hub-url https://your-server/orgs/my-org/repositories/<dest-id> \
  --org-id my-org --repository-id <dest-id> \
  --header "Authorization: Bearer $SWARMFILE_DEST_API_KEY" \
  --pin-key "$PUBLIC_KEY" --provenance origin.json
```

The manifest is a **file**: its key ring is untrusted input, so `--provenance` requires `--pin-key` (the key obtained out-of-band). When the origin serves `/.well-known/swarmfile.json`, the manifest carries that signed statement and the verifier reports `originAttested`; without one, the origin NAME is only a label. A matching commit/tree proves identity, not that the mirror serves the tree object or any block.

Managed and E2E destinations

A **managed or E2E** destination (or source) is re-sealed under the destination's key, so every content id changes and no tree match is possible; `--provenance-out` refuses it rather than writing a manifest the destination cannot satisfy. A **mirror** cannot preserve a managed/E2E commit DAG.

On a self-hosted community server the optional `swarmfile-server admin copy-repository` is the supported same-server exact-history copy for managed repositories (see Self-hosting); it copies the DB/key/data rather than re-sealing. Cross-hub/E2E key sharing remains a separate, deliberate gap.

Other provenance caveats

A subtree (`--path`) mirror, `--exclude`, a non-empty destination and a copy-mode resume state are likewise refused: the manifest describes the origin branch's whole tree. An ACL-limited source key sees only a partial tree — use an owner key; the consumer's verification fails loudly rather than claiming a match.

The manifest proves the origin's confirmed head/tree and its signed checkpoints; it does not prove every block is served by the mirror. See Verifiable History.

Scale

The listing follows the hub's cursor pages, so a directory with more than a thousand entries mirrors completely. Up to `--concurrency` files copy in flight at once (default 4; each still uploads its blocks through a shared 8-way pool), and `--state` makes a whole-repository copy resumable — an interrupted run re-reads only the files it has not mirrored. A very large repository is still a full byte transfer the first time.

Exit codes

`0` success; `1` a failure (including a refused collision or symlink); `2` a flag/usage error from the argument parser.

Diagnosing Connectivity

`swarmfile-doctor` checks that a hub, or your own community server, is reachable and correctly configured, from a machine with no mount. It runs the same probe core the desktop app's **Doctor** uses (shared through the open `swarmfile-lib`) and prints the same report (`--json` for scripts).

Install

```
curl -fsSL https://get.swarmfile.com/doctor | sh
```

Use

```
# reachability + the server's descriptor, unauthenticated
swarmfile-doctor --hub-url https://hub.swarmfile.com

# add the block-path probe (needs a key and a repository)
SWARMFILE_API_KEY=sf_key_… swarmfile-doctor \
  --hub-url https://your-server --org-id my-org --repository-id <id>

# machine-readable, and to a file
swarmfile-doctor --hub-url https://your-server --json --report /tmp/doctor.json
```

`--hub-url` defaults from `$SWARMFILE_HUB_URL`; `--token` from `$SWARMFILE_API_KEY`.

Figure omitted in print (animation): swarmfile-doctor reads a hub end to end — DNS, TLS, health, the key, the block path, the declared profile — then walks three breakages deployments actually ship: a revoked key, a service that isn't running, and a typo'd repository id. Every answer is a real request against a real misconfiguration. - https://swarmfile.com/demo/oss-doctor-debug.cast

The checks

Check	What it proves
`dns <host>`	the hub hostname resolves
`hub https /health`	`GET /health` (or `/healthz`) answers 2xx over TLS
`hub deep health`	`GET /healthz/deep` (a 404 is expected for a community server)
`cloud path`	the block route answers 404 for a never-written content id (the storage path is open)
`oidc token`	an authenticated repository metadata read against the active repository (a valid session or token)
`server_profile`	the profile descriptor at `GET /api/profile` (`core` on a community server, `hosted` on the hosted service)

With the engine running

The same binary completes the full picture when a `swarmfile-engine` is running (the desktop app, or a seed node). It finds the engine through its cache-dir `control.json` and asks it over the local control socket for the checks that need it (peer-to-peer connectivity and local-network peer discovery, the mount, transfer stats, the host firewall, the engine socket) **against whatever hub the engine is configured for**, the hosted cloud or a community server.

Those checks come back real; with no engine running they appear with the same names as `pass` + "skipped (requires the engine)". A `--hub-url` still runs the engine-free probes against that hub and merges in the engine's engine-only checks, so you can check one hub while still getting local engine/network state.

Options and exit codes

Flags mirror the desktop doctor (`--json`, `--quiet`, `--report <path>`, `--hub-url`, `--no-iroh`, `--no-mdns`, `--iroh-peer`, `--cache-dir`, `--latency`, `--verbose`), plus the headless inputs `--token`, `--org-id`, `--repository-id` and `--timeout` (seconds). `--latency` adds the per-route timing probe above (three samples each, 2 s per sample, 6 s overall); the desktop tray's **Run doctor** sends the same flag. No report file is written unless you pass `--report <path>`.

`--repair` (with `--yes` to skip the prompt) reinstalls the Swarmfile desktop app from the update channel it was installed from, after verifying the download; on a machine without the desktop app it refuses and points you at the downloads page. See `swarmfile-doctor --repair`.

Exit codes match the desktop doctor: **0** every check pass, **1** at least one failure, **2** warnings only. With `--repair`: **0** reinstalled, **1** a step failed, **3** refused or canceled with nothing changed. A connection failure or a refused key prints a hint naming what to look at.

Uninstall & Clean Reinstall

Removing Swarmfile from a machine is straightforward, with one important distinction: **repairing** an install and **removing** one are different operations, and repair is almost always what you want.

If the app is misbehaving (a broken mount, a missing CLI tool, a half-applied update), use **Diagnostics → Reinstall...** or, from the command line, `sudo swarmfile-doctor --repair` instead (on Windows, from an administrator prompt; add `--yes` to skip its confirmation prompt). A repair reinstalls the full package and **keeps** your cache, queued uploads, config, and sign-in; removing the app and its data does not.

A repair stops the engine briefly while the installer runs (on Windows the drive is detached cleanly first) and it reconnects when the install finishes; queued work is unaffected.

Everything below is for a deliberate removal — decommissioning a laptop, handing it to someone else, or a truly clean slate.

Before you remove anything

Three things live only on this machine and are lost when you delete its data:

1. **Queued uploads.** Saves that haven't reached the hub yet live in the local database. Check `swarmfile status` (the `pending uploads` / `commits pending` counters) and `swarmfile uploads --wait`; wait for zero, or accept losing those changes.
2. **Stuck saves.** `swarmfile sync stuck` lists saves the hub keeps refusing. Export or resolve them before wiping the cache — they will not come back.
3. **Reservations and locks.** `swarmfile offline return` releases any Pack & Go reservations this machine holds; checkout locks you took expire on their own lease, but releasing them now is neighborly. If this machine runs a **seed node**, disable seed mode first (`swarmfile seed disable`) so the org's discovery list doesn't keep advertising it.

Anything staged in a changelist that has been **parked** is kept on the hub and can be resumed from another machine; unsent changelist work is local and goes with the cache.

One more piece of wiring is separate from the app install: if this machine used a repository as a git-LFS server, run `swarmfile-lfs uninstall` to remove the custom-transfer adapter from your git configuration. Uninstalling the app does not do that for you.

Windows

1. **Quit the Desktop App** from the tray (including any "quit engine" option so the mount dismounts). The uninstaller stops the engine and tray itself (with a force-kill backstop), but quitting first avoids holding the mount open. If uploads are still in flight, Swarmfile drains them before stopping; after about 30 seconds it asks whether to abandon the remaining saves or keep waiting.
2. **Uninstall from Settings → Apps → Swarmfile → Uninstall.** For scripted removal, find the product code and call `msiexec`:

   ```
   $key = Get-ChildItem 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall',
                        'HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Uninstall' |
     Where-Object { $_.GetValue('DisplayName') -match 'Swarmfile' } | Select-Object -First 1
   msiexec /x $key.PSChildName /qn /norestart
   ```

   If the install came from the bundled `Setup.exe`, its uninstaller lives in `ProgramData\Package Cache`; Settings → Apps invokes it with `/uninstall` for you.
3. The uninstaller removes the program files (`C:\Program Files\Swarmfile`), the `PATH` entry, the `swarmfile://` protocol registration, the inbound UDP firewall rules, the Explorer overlay handlers, the Start Menu shortcut, the per-machine `Run` entries, and the **Swarmfile Updater** scheduled task. WinFsp, the VC++ Redistributable, and WebView2 are chain prerequisites and deliberately stay installed.
4. **Per-user data is deliberately left behind** (uninstalling for one user shouldn't wipe another's cache):

   ```
   Remove-Item -Recurse -Force "$env:LOCALAPPDATA\swarmfile"
   ```

   This removes the cache, queued uploads, config, and sign-in for that user only. One leftover is harmless: `HKCU\Software\Swarmfile\MountPoint` (the remembered drive letter), which the MSI doesn't own.

To reinstall, run the bundled `Setup.exe` again — or a **repair** if you only wanted the app fixed. If you downloaded a fresh `Setup.exe`, expect the SmartScreen prompt again; Getting Started walks through it.

macOS

The `.pkg` install is machine-wide: the app in `/Applications`, LaunchAgents in `/Library/LaunchAgents`, and CLI symlinks in `/usr/local/bin`. Removing it takes a few commands; run the `launchctl` lines as the logged-in user and the `rm` lines with `sudo`.

1. Quit the Desktop App, then stop its background services:

   ```
   launchctl bootout gui/$UID/dev.swarmfile.tray 2>/dev/null
   launchctl bootout gui/$UID/dev.swarmfile.engine 2>/dev/null
   ```

2. Remove the app, the agents, the CLI links, and the bundled FUSE-T pieces:

```
sudo rm -rf /Applications/Swarmfile.app
sudo rm -f /Library/LaunchAgents/dev.swarmfile.engine.plist \
           /Library/LaunchAgents/dev.swarmfile.tray.plist
sudo rm -f /usr/local/bin/swarmfile /usr/local/bin/swarmfile-*
sudo rm -f /usr/local/lib/libfuse3.4.dylib /usr/local/lib/libfuse3.dylib \
           /usr/local/lib/libfuse-t-1.2.7.dylib /usr/local/lib/libfuse-t.dylib
sudo rm -rf "/Library/Application Support/fuse-t"
sudo pkgutil --forget dev.swarmfile
```

3. Remove the per-user data (cache, queue, config):

```
rm -rf ~/Library/Caches/swarmfile ~/.cache/swarmfile ~/.config/swarmfile
```

   The engine's cache and config are the `~/.cache/swarmfile` / `~/.config/swarmfile` pair; `~/Library/Caches/swarmfile` covers `.dmg` and other install layouts. Repeat per user profile.

4. Remove the engine's macOS Keychain items — the sign-in token and this device's end-to-end key live there, not in the files above. Each item's service is `com.swarmfile.engine`; open **Keychain Access**, search for that string, and delete every match (or `security delete-generic-password -s com.swarmfile.engine`, repeated for each item). Do this per user profile too; the login keychain is per user.

If Finder or an app still shows the drive, it is a stale mount from a killed helper: log out and back in, or reboot, before wiping. To fix an install you intend to keep, use **Diagnostics → Reinstall...** or `sudo swarmfile-doctor --repair` instead. On a Mac where the app was copied from the `.dmg`, `--repair` refuses (that copy is owned by your user, not root); install the `.pkg` from Downloads instead.

Linux

1. **Quit the Desktop App and the engine for each logged-in user first** — the package's pre-remove script disables the systemd units but does not stop services already running in your session, so a removed engine can keep serving its mount until you log out or kill it.
2. **Remove the package:**

```
sudo apt remove swarmfile      # remove the app and units
sudo apt purge swarmfile       # also remove the packaged /etc/swarmfile
```

   Removal disables the per-user `swarmfile-engine` and `swarmfile-tray` systemd units; nothing else (firewall, mounts) is touched. If a mount is still present after removal, unmount it (`fusermount -u <mount point>`) or log out.

3. **Remove the per-user data** for each user who signed in:

```
rm -rf ~/.cache/swarmfile ~/.config/swarmfile
```

   That covers the cache, queue, and config. If you ran a seed or a headless engine with its own `SWARMFILE_CACHE_DIR`, remove that directory too.

4. **Remove the keyring entries.** Where a desktop keyring is available (GNOME Keyring or KWallet), the sign-in token and this device's end-to-end key live there under the service name `com.swarmfile.engine` — delete those items in **Passwords and Keys** (Seahorse) or KWallet. On a headless/seed machine with no keyring, the engine falls back to a file in the cache directory, which step 3 already removed.

What lives where

Platform	Application	Per-user data (cache, queue, config, sign-in)
Windows	`C:\Program Files\Swarmfile`	`%LOCALAPPDATA%\swarmfile` (DPAPI secret store included)
macOS	`/Applications/Swarmfile.app`, `/Library/LaunchAgents`, `/usr/local/bin`, `/usr/local/lib/libfuse*`, `/Library/Application Support/fuse-t`	`~/.cache/swarmfile` (cache, queue, logs) + `~/.config/swarmfile` (config) + Keychain items (service `com.swarmfile.engine`)
Linux	package files, per-user systemd units	`~/.cache/swarmfile` + `~/.config/swarmfile`; `/etc/swarmfile` (purge only); keyring items (service `com.swarmfile.engine`) where a secret service runs

The cache directory is also where **log files** live, so if you're removing the machine as part of a support case, copy that directory aside first.

Clean reinstall

To start completely fresh on the same machine:

1. Follow the uninstall steps above, including deleting the per-user data.
2. Install the current package from swarmfile.com/downloads — or, on macOS, make sure it's the `.pkg`: the `.dmg` doesn't set up the engine service or the CLI on `PATH`.
3. Sign in again and re-open the repository. The mount comes back where the Drive Letter / mount-point settings say; nothing on the hub is affected by the reinstall.

Where to go next

- Release Channels & Updates — how updates work and what a repair does.

- Supported Platforms & System Requirements — installer names and per-platform layout.
- Deploying to Your Team — removing or re-provisioning machines at fleet scale.

Organizations, Repositories & Members

An organization is the top-level container for your team: it owns repositories, members, billing, and identity configuration. Everything in Swarmfile lives inside exactly one org.

Creating an organization

There's no separate "create your organization" step. When you sign up at `/start`, you register and an org is created for you in the same flow. Choosing **Starter or Pro** goes straight into Stripe Checkout to pick and pay for a plan; choosing **Free** creates the org without any card or checkout. The org gets a slug auto-derived from what you typed at signup; if that slug is already taken, it's auto-resolved to something unique so signup never blocks on a collision.

One account can create up to **25 organizations**; past that, creating another answers with a support pointer. The limit is an abuse bound (a real buyer has a handful of orgs), not a plan feature — contact support if a legitimate setup needs more.

Only **one** of those can be a Free organization: a second Free org is refused (`free_org_limit_reached`), and the new org can be created on Starter or Pro instead. Upgrading your Free org to a paid plan frees the slot; a deleted Free org counts until it is fully removed. See Billing & Plans.

The org slug matters beyond cosmetics — it appears in public share links (`/s/:orgSlug/:token`) and org URLs, though the share token itself is not bound to it.

Renaming your org

As owner, you can rename the org's display name or its slug from Settings at any time.

Renaming the name is cosmetic. Renaming the slug changes the visible URL, but old org URLs keep resolving: the previous slug is reserved as an alias for the life of the org (and a new org cannot take it). A share link's token is not tied to the slug, so old `/s/:orgSlug/:token` links keep working. There is no canonical 301, so a visitor using an old link may still see the old slug in the address bar. Changing the slug still requires retyping the current slug to confirm, because the visible URL changes.

Repositories

A repository is a mounted filesystem within an org. Owners and members with sufficient permissions can:

- **Create** a new repository
- **Rename** a repository (the old slug is reserved as an alias, so old URLs keep resolving; a repository API key should use the repository and org ids)
- **Archive** a repository (read-only, unmounts for day-to-day use, reversible)
- **Delete** a repository (removed from active use straight away; its files, history, git-LFS objects, and stored data are permanently purged 30 days after deletion, and there is no self-serve undelete. Recovery is not guaranteed — contact us before the purge window closes to discuss it. The 30-day window is fixed; a shorter per-user trash-retention setting does not shorten it. A very large entry tree is removed through a durable background job rather than the immediate cascade, so a big delete can take a while to finish; the CLI waits for it by default and `op status` shows its progress.) Deletion writes durable audit rows that outlive the purge, so it remains auditable after the data is gone.

A repository can also be cloned and pushed with git once an owner turns on git access (`swarmfile git enable`, the web repository menu, or the Desktop App's **Repository settings → Git access**) — see Cloning a Repository with Git.

Archived repositories are read-only

Archiving freezes a repository's content until someone unarchives it. Every change is refused with `403 repository_archived`, whether it comes from the desktop drive, the web, the CLI, an API key, git-LFS, or a share link:

- saving, uploading, creating, renaming, moving, or deleting files
- commits and changelists, branches (create, merge, archive, protection rules), tags, and merge requests
- publishing releases, RFIs, labels, and comments (adding or resolving, including through a share link)
- changing access: granting permissions, switching the repository's ACL mode, or creating new share links (revoking existing access still works)
- taking a lock: file locks, offline checkouts, folder reservations, byte-range locks, and `git lfs lock`

What keeps working:

- reading, browsing, downloading, and history
- **unarchiving** the repository
- taking a public release down, revoking access (removing an existing grant or revoking a share link), and releasing locks (including checking a file back in and `git lfs unlock`)
- canceling an upload that was still in progress when the repository was archived, so nobody is left waiting for a file that will never arrive

If someone still has the drive mounted, a save to an existing file is kept beside it as a separate copy instead of being lost, and a brand-new file's content stays on their machine with a message explaining why it wasn't uploaded.

Each repository runs in one of two modes:

- **Open mode** — the default for every new repository; more permissive, suited to small teams or early-stage repositories
- **Protected mode** — enforces explicit ACL grants on folders and files; opt-in at creation or by switching the repository later

Mode selection and the underlying ACL model are covered in Permissions — this page only covers repository lifecycle, not access control.

Repository configuration and templates

One options manifest is the single source for every create-time control — encryption tier, commit mode, retention, visibility, ACL mode, connection (the data plane), repository type, default-branch protection, the create-only default branch name, **Ensure Git Compatible**, data residency, and the per-repository external-collaborator policy. The web dashboard, the Desktop App, and the CLI all render the same set from it, so a choice looks and behaves the same wherever you create the repository.

Templates give a new repository a starting shape: **Blank**, **Media & post**, **Music production (Ableton Live)**, **Music production (DAW-agnostic)**, **Software (git repository)**, **Open source repository**, **Revit worksharing**, or **Geospatial**. Where a template defines one, creation seeds starter files (an ignore file, a folder skeleton, or — for **Open source repository** — README/CONTRIBUTING/SECURITY/CODEOWNERS skeletons, a publicignore starter, an inert commented-out CI starter, and a LICENSE placeholder that never guesses a license); an explicit create option always wins over the template's value.

After creation, the repository's settings show each effective value and where it came from. `swarmfile repository show` prints the same view for scripts, and `swarmfile repository set` changes the settings it can — see the CLI reference and Getting Started. The billing page's three-state entitlements matrix labels each setting **included**, **upgrade**, or **not built**, so a plan gate is visible before you hit it.

Switching the active organization or repository

An engine can host several mounts at once — one repository (and one branch) per mount — so a single machine can work in more than one repository without a second install. The common case is still one active workspace per engine, and switching it is a hot, in-place operation where possible:

```
swarmfile workspace status                      # which org/repository this engine is on
swarmfile workspace list                        # every org you can reach, plus the current org's repositories
swarmfile workspace switch --org <org-id>       # adopt that org's default repository
swarmfile workspace switch --org <org-id> --repository <repository-id>
```

To run more than one repository side by side instead of switching, open an additional mount — `swarmfile mounts open <path-or-letter> --label "..." --repository <repository-id>` (same org only) — and target commands at it with the global `--mount <id>` flag; `swarmfile mounts list` shows every open mount and which one is `current`. The per-agent version of this setup is documented in Coding Agents.

Switching is **not** a re-authentication: your session is per-user, not per-org, and already covers every org you belong to. Both same-org and org switches normally happen in place (no engine restart), and a target repository whose key has not been released yet is waited for rather than turned into a restart.

The engine drains and restarts only when the hot path cannot complete — typically when there is no live mount to re-scope — so the drive may then briefly unmount and return.

What follows the switch

A hot switch is workspace-wide for the repository you are leaving: every mount of that repository moves to the new repository's active branch (a per-mount branch pin doesn't survive a repository change), so open drives follow the switch. A mount on a *different* repository stays where it is — and because such a drive can't follow the engine to another **org**, an org switch is refused while one is open (`cross_repository_mount_open`). The Desktop App names both before you confirm: the drives that will follow and any that must be closed first.

A switch is refused while a changelist is open (submit, cancel, or `--park` it first).

Open files and in-flight work

A switch doesn't wait for uploads: saves still uploading in the repository you're leaving keep uploading in the background, and the Desktop App and `swarmfile status` show them as saving in other repositories until they land (see Reading the sync status).

While a switch is running, opening or saving a file on the drive can pause for up to a minute until it finishes, and the Desktop App shows that the drive is switching.

A file that was already open in an application before the switch still belongs to the repository it came from: its later saves are refused rather than written into the new repository, the Desktop App tells you straight away, and when the file closes its unsaved changes are kept on disk and the notification says where. If the application hadn't changed the file before the switch, its first change is refused before anything is written, so Swarmfile has nothing to keep: the changes are still in the application, and the notification says so. Save them under a new name, or switch back and save again.

The one exception is an org behind its own SSO — reaching it goes through that org's identity provider, not the built-in one (see Identity).

First sign-in on a new machine

A freshly-provisioned machine auto-adopts an org and repository on first sign-in, so there's often nothing to switch — see Deploying to Your Team for how that default is chosen.

Inviting members

Members are invited by email. An invite generates a link; the invitee accepts it to join the org, landing with the coarse `member` org role (below `admin` and `owner` — see the next section for what that hierarchy gates). Don't look for a role picker on the invite screen; there isn't one — you can't invite someone directly as an admin, only promote them afterward.

On a plan with ACLs (Pro and above) the invite form can also pre-grant whole-repository **Read** or **Write** access, listed on the pending invitation and applied when it is accepted; an invite sent without picks adds the org membership only, and repository access is granted afterward under that repository's **Permissions** tab.

The plan bounds who can be invited: **Free is a solo tier (one seat)** — the Team tab shows a one-seat notice with an upgrade link instead of the invite form — and a Starter/Pro **trial** is capped at two seats until it converts — or until an owner ends it early from the Billing tab, which the Team tab's seat-limit notice links to (admins see an "ask an owner" note instead). A signup that picks more than two seats skips the trial and starts paid. See Billing & Plans.

Paid Starter, Pro, and Enterprise orgs have no seat ceiling; each seat simply bills at the plan's per-seat rate.

What a member can see and do *within a repository* is governed by the ACL system described in Permissions; the org role is a separate, coarser layer that gates org-wide admin surfaces (billing, permissions administration, quarantine) rather than per-file access.

If your org configures SSO with an auto-provisioning group (see Identity), anyone in that IdP group is added to the org automatically on their first sign-in — no manual email invite needed.

SCIM on its own doesn't do this: it syncs your directory into Swarmfile, but org access still requires either that SSO group match or a manual invite.

On a public repository with access requests enabled, an approved **membership** request arrives through this same invitation flow — the requester accepts the standard email invite to join, and on a protected repository the invitation carries the repository grant they need to see it (Public access requests).

Removing members

An owner or admin removes a member from the **Team** tab (only an owner can remove another owner). The hub stops serving them the org's file content and keys within a few seconds. The web dashboard drops the org for them, and their desktop drive stops serving the org's files, including files it had already cached. A computer that's offline at the time does this when it next reaches the hub. A link to a single block that was issued just before the removal can keep working for up to two minutes until it expires.

Their engine also clears its local cache of the repository (file listings and cached content) and discards its copy of the repository's content key. The one exception is changes they saved that hadn't finished uploading: those stay on their computer, held, and upload if their access is restored. Their desktop app tells them so. The repository's key itself isn't changed.

For an end-to-end encrypted repository, removal also revokes their key wraps, but can't recall key material already on their device. Re-inviting the member restores their access without a reinstall — including the repository permissions they held — and their org role comes from the new invitation.

To cut a removed member off everywhere, Swarmfile visits each of the org's repositories in turn, closing their open connections and revoking their keys there; in a large org this can take a few minutes.

The same happens when a guest's access is revoked, your directory deprovisions someone, a device key is revoked, or a group's membership changes.

Owners and admins can follow it in **Settings → Access revocations**: what's still in progress, any repository Swarmfile couldn't reach (with the reason), and what each finished sweep did — including how many devices confirmed they deleted their cached copy, and a **Retry** for a repository whose sweep stopped.

Removal also changes the org's peer-to-peer key, the secret your team's computers use to connect directly to each other, so the removed member's computer can no longer join in. Everyone else's desktop app picks up the new key on its own within a couple of minutes, with no restart and no interruption to the drive.

If a computer seen in the last seven days runs an older desktop app that can't switch keys without restarting, the change waits until it updates; owners and admins see a **Peer-to-peer key change pending** notice in **Settings → Network** saying how many computers it's waiting for. Until then the removed member's computer may still connect directly to your other computers, but their access to files is already removed.

The org's activity feed records both the wait and the change.

Who can see plan and usage standing

Any org member — not just the owner — can see the org's current plan and usage standing: seats, storage, and how close the org is to any limits. This is surfaced in both the Desktop App's plan panel and the web dashboard.

This is a deliberate choice, not an oversight. Owner-only areas of the dashboard — billing and usage administration — stay restricted to owners.

Permissions administration and quarantine are open to admins as well as owners. The audit log itself is open to any member (ACL-filtered — you see events on entries you can read); only its CSV export, plus quarantine and org-policy events within the feed, stay owner-only.

But plan and usage *visibility* is open to everyone on the org, so a member isn't left guessing why an upload is being throttled or what tier the team is on. The dashboard's **Public streaming** card is member-visible; the Settings → Usage controls and other administration stay owner-only.

Changing request limits

Usage standing is read-only for members; the **request limits** themselves — the per-org and per-user request ceilings — are configurable from the web dashboard's **Settings → Organization** (owners only), the Desktop App's **Settings → Developer** panel (owners and admins), or `swarmfile admin rate-limits` (owners and admins).

`swarmfile admin rate-limits usage` shows the current minute's usage against them, and a raise can be time-boxed (`--for 6h`). A single API key can carry its own ceiling (`swarmfile api-key rate-limit`) for that key's bucket. A fleet's aggregate headroom comes from its active keys: each adds its per-key allowance to the org default, up to the plan org max, and the plan's key allowance (seats plus packs) caps how many keys count. An org override replaces the scaled window.

Seat count

Seat count itself isn't something you set manually — it's derived from live org membership and kept in sync automatically. Details are in Billing & Plans.

Identity

Every org needs a way to authenticate its members. Swarmfile gives you one out of the box and lets you layer on enterprise identity — SSO, SCIM, and on-prem LDAP sync — as your org grows.

Built-in identity provider

Every org gets Swarmfile's own OpenID Connect identity provider by default, with email/password authentication. There's no setup required — it's active from the moment your org exists, and for most Starter-plan teams it's all you need.

Bring your own SSO (OIDC)

Plan: Pro and above.

If your org already runs an identity provider, you can configure it in Swarmfile instead of — or alongside — the built-in one. Owners set it up under **Settings → Identity**: an OIDC issuer URL plus the audience/client id, validated against your IdP's standard `/.well-known/openid-configuration` discovery document and JWKS.

Entra ID and Okta are the two explicitly supported and tested patterns via their own OIDC endpoints. **SAML** SSO is offered on Enterprise through an adapter that sits between your SAML IdP and Swarmfile's sign-in, rather than as a self-serve option on this screen; if your IdP only exposes a SAML app, talk to us and we'll set it up with you.

SSO configuration is per-org, and each org's config is hard-bound: when a member signs in, the token they present is validated against the JWKS for that org's specific configured issuer. One org's SSO setup can't be used to forge identity into another org, even if both orgs use the same upstream IdP vendor.

> If an external IdP config is ever misconfigured badly enough to lock everyone in the org out, the owner has a break-glass path back to the built-in Swarmfile IdP. This is an owner-only safety net — don't rely on it as a routine way to bypass SSO.

> **Desktop app:** connecting the desktop app works from an external-IdP (SSO) sign-in too. Swarmfile records the external principal — scoped to that issuer and subject — the first time it sees a verified sign-in, and the device session it then mints is marked as externally originated, so your org's authentication policy exempts it exactly like the SSO browser session it came from. Sign in through SSO in the browser, then use **Connect app** as usual. A subject that collides with an existing Swarmfile account is refused with `identity_conflict` rather than being merged.

By default, SSO only authenticates people who are already members — someone who signs in successfully via your IdP but isn't yet in the org still needs a manual invite. To skip that step, set an auto-provisioning group in the org's SSO configuration: anyone whose token carries that group in its `groups` claim is added to the org automatically as a member the first time they sign in. Leave it unset and nothing changes from the default invite-based flow.

If your org's authentication policy requires MFA, you can also give the SSO config an **MFA authentication context**: Swarmfile sends that value as `acr_values` on the sign-in redirect so your IdP can ask for an MFA-strength authentication. Ask your IdP which value it accepts (for example `urn:mace:incommon:iap:silver`); leave it blank to send none — an IdP that validates `acr` strictly would reject a value it doesn't know, and enforcement at org access is unchanged either way.

SCIM provisioning

Plan: Pro and above.

SCIM 2.0 lets your identity provider push directory sync — creating and disabling principals, syncing group membership — into Swarmfile directly. SCIM by itself doesn't grant org access: a synced user still needs either an SSO auto-provisioning group match (above) or a manual email invite to actually join the org.

Owners manage it under **Settings → Directory Sync**: mint a bearer token for the integration (shown once — copy it into your IdP immediately), revoke one, and watch the recent sync log to confirm users and groups are actually arriving. Point your IdP at the hub's SCIM base URL for your org, `https://<hub>/orgs/<org-id>/scim/v2`; contact us if you need the exact URL for your deployment. Requests authenticate with the bearer token.

Entra and Okta each use slightly different PATCH conventions for SCIM group and user updates. Swarmfile handles both dialects, so you can point either one at the same endpoint without translation on your end.

On-prem LDAP sync

Plan: Pro and above.

If your org's directory lives on-prem — LDAP or Active Directory rather than a cloud IdP — you run a sync agent on your own infrastructure (for example, as a Docker container pointed at your AD) instead of connecting a hosted IdP. The agent syncs your directory into Swarmfile's principal directory on your schedule, without your directory ever needing to be reachable from outside your network. Like SCIM, this syncs identity — it doesn't by itself grant org access; members still need an SSO auto-provisioning group match or a manual invite.

Authentication policy (require MFA and/or a verified email)

Plan: Starter and above.

Individual members can always protect their own account with an authenticator app (TOTP), under **Settings → Account → Security**. An org can additionally *require* that protection for everyone. Owners configure this under **Settings → Identity → Authentication policy**:

- **Require multi-factor authentication** — members must sign in with an authenticator app. A session signed in before enrollment does not satisfy it; sign in again after enrolling.
- **Require a verified email address** — members must have verified their email before using the org.

You choose when enforcement begins: immediately, or after a grace window (7 days by default; the dashboard offers up to 30 days, and the API accepts up to 3650 days or an explicit enforcement date up to ten years out). During the window everyone keeps working; the web dashboard, the desktop tray, and `swarmfile status` all nudge members to enroll.

Once enforcement begins, any org-scoped request from a session — or a personal access token, which snapshots its session's factors at mint time — that doesn't meet the policy is refused with one clear message and a link to Account settings. Members signing in with a Swarmfile password at the org's login page (`/login/<slug>`) also see the requirement stated up front, so nobody starts a sign-in without their authenticator; members signing in through the org's own SSO provider are exempt from the policy, so no hint is shown on the SSO button.

Members who would fail also get an email: when the policy is enabled, again in the last 24 hours before enforcement begins, and once it takes effect. These are transactional security notices tied to the org's policy, not marketing — they aren't subject to notification quiet hours or unsubscribe settings.

The refusal is never a lockout: **enrolling and verifying are never gated by the policy**, and the owner can turn it off at any time from the same Identity page. That off-switch is a genuine break-glass — it stays reachable even for the owner's own non-compliant session.

One gate applies to turning it on: **the owner must have enrolled an authenticator app themselves before requiring MFA for everyone** — the same rule the org enforces, applied to the person setting it. (Signing in through an external IdP exempts you, since your IdP owns that factor.) Once a policy exists, the Identity card also shows an enrollment summary: how many members are enrolled, and which owners and admins aren't yet. **Download CSV** beside it exports the full roster — email, name, role, and enrollment status — for an IT review.

If a member loses both their authenticator app and their recovery codes, an operator can remove MFA for them after verifying who they are — the same effect as the member's own **Turn off** under Account settings: every session is signed out, and the change is recorded in the operator audit trail.

Who's covered:

- **Members and owners, including the owner who enabled it.** Owners are held to their own policy; only the off-switch is exempt.
- **Guests are exempt** — a guest's access is limited to the folder they were invited to.
- **Repository API keys are exempt** — they're org-provisioned machine identities, governed by key scoping and revocation.
- **Orgs that sign in through their own IdP are exempt from both flags** (see Bring your own SSO) — that IdP owns factors and email assertions.

Downgrades follow the same freeze rule as SSO: an enforced policy keeps working if the org drops below Starter, and can always be turned off, but a new requirement can't be turned on until the org is back on Starter or above.

Personal access tokens

Interactive sign-in is the right thing for a person at a keyboard. It's the wrong thing for a script, a scheduled job, or a CLI session that needs to keep working without a browser in the loop. A **personal access token** (PAT) is a long-lived bearer credential for exactly those cases — it stands in for *you*, non-interactively.

The key property is that a PAT carries no permissions of its own. It resolves to your identity, and every access check then runs against **your own current role**, exactly as if you'd signed in normally. A PAT can never do more than you can do right now, and it stops working the moment your access does: change your role and the token follows it; get removed from the org and the token is invalidated along with your membership. There's nothing separate to clean up.

Because it acts as you and dies with your access, minting one is **self-serve for any member** — it isn't owner- or admin-gated. Create and revoke tokens under **Settings → Access Tokens** in the web dashboard. Each token is shown in full exactly once, at creation; after that only its name and prefix are visible.

You can hold up to **20 active tokens per member, per org** at a time — the cap is counted against your own tokens in that org, so one member's tokens never consume another's allowance. Revoke one to make room if you hit the cap. Tokens can be given an expiry at creation or left non-expiring for unattended automation; the credential begins with an `sf_pat_` prefix and is sent as a normal bearer token.

If your org **mandates external SSO**, PATs are subject to that too — there's no bypass. A token that would authenticate as your built-in identity is rejected in an SSO-enforced org just as an interactive built-in sign-in would be. A PAT is a convenience for your own access, never a way around org policy.

If your org enforces an authentication policy, the same no-bypass rule applies, with one deliberate nuance: a PAT **snapshots the factors and email assertion of the session that minted it**. A token minted from a compliant session keeps working; a token minted before you enrolled (or before your email was verified) stops satisfying the policy when enforcement begins. Re-sign in with your second factor and mint a fresh token — the Access Tokens page flags any token that won't satisfy the org's current policy.

Security guidance. A PAT is a live credential — treat it like a password. Give each one a descriptive name so you can tell them apart, scope them to a single machine or job rather than reusing one everywhere, set an expiry where the workflow allows, and revoke a token the instant it's no longer needed or might have leaked. Revoking a token from the list ends its access; token material can only ever be created from an interactive sign-in, so revoking every token you can see genuinely ends non-interactive access under your identity.

PATs vs. repository API keys

A PAT is not the same thing as a **repository-scoped API key**, and the difference matters:

- A **PAT** is *you*, self-served, org-wide within your own access, and tied to your membership.
- An **API key** is a *separate service identity* scoped to a single repository, minted by an admin or owner rather than by the member who runs the automation. It has no role of its own beyond that repository and doesn't come and go with any one person's membership.
- A repository API key can be **branch-scoped** (one non-default branch; git/history/MRs stay on that branch) or **path-scoped** (engine/API-only, no git/history/commit walk/MRs). The scope is set at mint time and only narrows the key. Community servers reject scoped-key fields; scoped keys are hosted-only until P1. See branch- and path-scoped keys.
- **Keys are a capped plan resource; PATs aren't.** Each plan includes a fixed headless-seat allowance (Free 1; Starter 2 per seat, min 5; Pro 5 per seat, min 10; Studio 10 per seat, min 20) and extra seats are $2/month each (one repository-scoped key per seat); minting past the allowance is refused with `keys_cap` until a key is revoked or a seat is added. Use PATs for a person's own automation and keys for the org's machines — one key per machine or fleet is the recommended shape, for revocation radius and its own rate-limit bucket (see Billing & Plans).
- A PAT can also mint a **service-minted agent token** for an approved `agent/` service delegation (Starter+; hosted). The child is branch-scoped, expires within an hour, and is revoked with the PAT or delegation; it narrows the PAT rather than creating a new identity. In an SSO-enforced org the mint needs an owner or interactive session. See Service-minted agent tokens.

Revocation takes effect within seconds. Both credentials are checked on every request; revoking a PAT or an API key (or deleting its repository) ends its access immediately on the server that handles the revoke, and everywhere else within a few seconds.

Revoking a credential also rides Swarmfile's durable revocation sweep, which closes its live event connections — a repository API key's own; for a PAT, every live connection for that member, because a connection doesn't name the token that opened it (engines reconnect within seconds with any other valid credential).

An org can cap how long one connection may live at all — 5 minutes to 24 hours under **Settings → Organization** — so a connection can never outlast a missed sweep by more than that cap. A changed per-key rate limit applies on the same schedule.

Reach for a PAT when a job should run as a specific human; reach for a repository API key when a CI pipeline or service should have its own durable identity independent of any individual. API keys are covered in Runner as CI and the CLI reference.

Plan gating and what happens on downgrade

SSO, SCIM, and LDAP sync are all Pro+ features, and the restriction is enforced server-side — not just hidden in the UI. Calling an SSO, SCIM, or LDAP endpoint on a Starter-plan org returns a real `402` rejection, not a silently-ignored request.

Downgrades don't retroactively break things. If an org drops below Pro while SSO or SCIM is already configured, the existing configuration keeps working — turning it off automatically would strand people mid-login or silently break provisioning. The gate only applies going forward: on a Starter plan, you can't set up something new, but nothing already running gets switched off underneath you.

For plan comparisons and pricing, see Billing & Plans. For what an authenticated member can actually access once signed in, see Permissions. Setting up SSO with an auto-provisioning group before you push installers to a fleet removes the separate invite step for newly provisioned employees — see Deploying to Your Team.

LDAP Directory Sync

If your users live in on-premises **Active Directory**, this is how their accounts and groups reach Swarmfile. It is AD-specific — the agent reads attributes like `objectGUID`, `objectSid`, `sAMAccountName`, and `uSNChanged`, and generic RFC-2307 directories (`entryUUID` / `uid`) aren't supported today. Swarmfile's control plane can't open raw LDAP connections, so a small **customer-hosted agent** runs inside your network, reads AD, and pushes normalized changes to Swarmfile over HTTPS. Nothing needs inbound access from Swarmfile to your directory.

If your identity provider is cloud-based (Entra ID, Okta), you don't need this agent — those push SCIM directly to Swarmfile. See Identity for SSO and SCIM setup.

What you need

- A **Pro plan or above** (directory sync is a Pro entitlement).
- An **org owner** to mint the sync token under **Settings → Directory Sync**.
- A **read-only service account** in AD with permission to read users and groups. The agent never writes to your directory.
- A host inside your network to run the agent — Docker is the supported path — with line-of-sight to a domain controller and outbound HTTPS to your hub.

How it behaves

- **Delta sync every 5 minutes** by default (`SYNC_INTERVAL_SECONDS`): scans only entries whose `uSNChanged` is newer than the persisted cursor. Cheap, catches user joiner/mover/leaver churn. Group membership rides along with the full sync, not the delta.
- **Full sync daily** by default (`SYNC_FULL_INTERVAL_SECONDS`): rescans users **and groups**. This is how **deletions** propagate — delta scans can't see a deleted object, so any principal missing from a full scan is disabled (`user_disable` / `group_disable`).
- **Crash-safe:** the cursor is written atomically and a partial batch leaves it unchanged, so the next cycle replays the same window; ingest is idempotent on `(source, externalId)`, so replays don't duplicate anyone.
- **Disables, not deletes:** an account that disappears from AD is disabled in Swarmfile, preserving its history and audit trail.
- **Server limits:** the hub accepts at most 500 operations per push (batches are clamped to 1–500; default 100) and 1,000 group members per group. The sync token carries its own request-rate ceiling.

Set it up

1. **Mint the sync token.** As an owner: **Settings → Directory Sync → New sync token**. Copy the plaintext (`swarmfile_scim_...`) — it is shown once, and it authorizes principal changes for the org, so treat it like an API key.
2. **Create the AD service account** with read access to the users and groups OUs you'll sync.
3. **Build the agent.** The agent is published as source in the swarmfile-sync-agent public repository (dual MIT/Apache-2.0); there is no prebuilt registry image. Download or clone a release, then build the container image from the agent's directory:

   ```
   cd swarmfile-sync-agent
   docker build -t swarmfile-sync-agent packages/sync-agent
   ```

   To run it without Docker, build it with Node.js instead (`cd packages/sync-agent && npm install && npm run build && node dist/index.js`), setting the same variables shown below in the environment.
4. **Run the agent.** Production-safe Docker Swarm example with secrets, using the image you just built:

   ```
   printf 'changeme' | docker secret create ldap_bind_password -
   printf 'swarmfile_scim_xxxxxxxx' | docker secret create swarmfile_sync_token -

   docker service create --name swarmfile-sync \
     --secret ldap_bind_password \
     --secret swarmfile_sync_token \
     --mount type=volume,source=swarmfile-sync-cursor,target=/var/lib/swarmfile-sync \
     -e SWARMFILE_HUB_URL=https://hub.swarmfile.com \
     -e SWARMFILE_ORG_ID=<your-org-uuid> \
     -e SWARMFILE_SYNC_TOKEN_FILE=/run/secrets/swarmfile_sync_token \
     -e LDAP_URL=ldaps://dc01.acme.corp:636 \
     -e LDAP_BIND_DN='CN=svc_swarmfile,OU=ServiceAccounts,DC=acme,DC=corp' \
     -e LDAP_BIND_PASSWORD_FILE=/run/secrets/ldap_bind_password \
     -e LDAP_USERS_BASE_DN='OU=Users,DC=acme,DC=corp' \
     -e LDAP_GROUPS_BASE_DN='OU=Groups,DC=acme,DC=corp' \
     swarmfile-sync-agent
   ```

   Seven variables also accept a `<NAME>_FILE` variant pointing at a file with the value — `SWARMFILE_HUB_URL`, `SWARMFILE_ORG_ID`, `SWARMFILE_SYNC_TOKEN`, `LDAP_URL`, `LDAP_USERS_BASE_DN`, `LDAP_BIND_DN`, and `LDAP_BIND_PASSWORD` — which is the preferred way to keep secrets out of `docker inspect`.

   The rest (group base DN, filters, cadences, batch size, TLS/timeout, log level) are read from the environment only. If you bind-mount a host directory for the cursor instead of a named volume, `chown 10001:10001` it first — the container runs as UID 10001.
5. **Confirm in the dashboard. Settings → Directory Sync** shows the recent sync log; the agent's own logs print a summary per cycle (`users=… groups=… disables=… pushed=… failures=…`).

Configuration reference

Variable	Required	Default	Notes
`SWARMFILE_HUB_URL`	Yes	—	Hub base URL, no trailing slash (production `https://hub.swarmfile.com`; use your organization's hub host if different); `_FILE` variant supported
`SWARMFILE_ORG_ID`	Yes	—	UUID of the org this agent syncs into; `_FILE` variant supported
`SWARMFILE_SYNC_TOKEN`	Yes	—	Bearer token from **Settings → Directory Sync** (`swarmfile_scim_…`); `_FILE` variant supported
`LDAP_URL`	Yes	—	`ldap://host:389` or `ldaps://host:636`; `_FILE` variant supported
`LDAP_BIND_DN`	Yes	—	Service-account DN; `_FILE` variant supported
`LDAP_BIND_PASSWORD`	Yes	—	Service-account password; `_FILE` variant supported
`LDAP_USERS_BASE_DN`	Yes	—	Search base for users; `_FILE` variant supported
`LDAP_GROUPS_BASE_DN`	No	users base	Search base for groups
`LDAP_USER_FILTER`	No	`(&(objectClass=user)(!(objectClass=computer)))`	LDAP filter for users
`LDAP_GROUP_FILTER`	No	`(objectClass=group)`	LDAP filter for groups
`SYNC_INTERVAL_SECONDS`	No	`300`	Delta cadence
`SYNC_FULL_INTERVAL_SECONDS`	No	`86400`	Full cadence (governs deletion detection)
`SYNC_BATCH_SIZE`	No	`100`	Ops per push request (server caps at 500)
`SYNC_CURSOR_PATH`	No	`/var/lib/swarmfile-sync/cursor.json`	Persist this across restarts
`LDAP_TLS_REJECT_UNAUTHORIZED`	No	`true`	Only `false` for self-signed labs
`LDAP_TIMEOUT_MS`	No	`30000`	Connect/search timeout
`LOG_LEVEL`	No	`info`	`debug` / `info` / `warn` / `error`

How AD attributes map

AD attribute	Swarmfile field	Notes
`objectGUID`	`externalId`	Canonical mixed-endian GUID, matching PowerShell
`objectSid`	`adSid`	Used for Windows DACL projection
`displayName` → `cn` → `sAMAccountName`	Display name	First non-empty wins
`mail`	Email	Optional
`userPrincipalName` → `sAMAccountName`	UPN	First non-empty wins
`userAccountControl` bit 2	Active (negated)	The `ACCOUNTDISABLE` flag

For OIDC SSO, the agent pre-links each user's `objectGUID` as the JWT `sub` — the default behavior of AD FS and Entra Connect. If your IdP uses a different `sub` claim, rebind users manually in the dashboard.

Troubleshooting

Symptom	Likely cause
`config: required env var X is missing`	A required variable isn't set; the agent fails before connecting
`bind failed`	Wrong bind DN/password, or wrong `LDAP_URL`; test with `ldapsearch -x -H ldap://host -D '<bind dn>' -W -b '<base>'`
`/sync/push returned 401`	Token wrong, revoked, or minted for another org — mint a fresh one
`/sync/push returned 503: tenant provisioning in progress`	The org's tenant is still being provisioned; the agent retries next cycle
Users disappeared	The last full sync marked them disabled — check which OU the service account can actually read
Deletions lag	Lower `SYNC_FULL_INTERVAL_SECONDS`; note full scans cost O(directory size) per run

Security

- The sync token authorizes **any** principal operation on the org. Store it in a secret manager, never in a committed env file.
- The agent reads AD only; give the service account `Read` on the OUs it needs and nothing more.
- Use `ldaps://` in production — `ldap://` sends bind credentials in the clear. Certificate validation is standard (no skip option beyond the lab flag); there is no custom CA or client-certificate setting today, so your domain controllers need certificates the agent trusts.
- The cursor file holds GUIDs, no secrets; normal filesystem permissions are enough.

Where to go next

- Identity — SSO, SCIM, and how synced principals join the org.
- Permissions — grants and protected mode for the synced groups.
- Security — the broader admin security model.

Permissions

Swarmfile's permission model is built from access control lists (ACLs) applied to folders and files, layered on top of a per-repository mode that decides what happens when no grant exists at all.

Read, comment, comment + upload, write, admin

Every grant is one of five levels, and each level can be either an `allow` or a `deny`:

Level	Lets you
`read`	See and download the contents of the folder or file
`comment`	Everything `read` does, plus leave comments and @-mentions on files in the folder
`comment_upload`	Everything `comment` does, plus upload new files through the server-side upload route — a guest-reviewer tier, deliberately still below `write`: it does not grant overwrite, delete, rename, rollback, or restore on existing files
`write`	Modify contents — create, edit, delete within the folder tree
`admin`	Manage grants on the folder or file itself, in addition to read/write

Each level implies everything below it, and the levels are ordered exactly as listed above. `comment` and `comment_upload` exist for guest reviewers and external collaborators — see External Collaborators & Sharing — who need to leave feedback (and, for `comment_upload`, drop off new files) without the ability to touch what's already there.

A grant is set on a specific folder or file and is inheritable down the tree beneath it. You don't have to re-grant access at every nesting level — set it once on a parent folder and everything under it inherits the grant. Depth isn't a tiebreaker: a grant closer to the file doesn't override one set higher up the tree. See below for exactly how multiple grants combine.

A grant applies to the item you name, not to its surroundings. Access is resolved per entry, so granting someone `read` or `write` on a single file lets them open that file (for example, by path or link) without granting the rest of the folder it sits in.

The way to it stays walkable, though: a folder on the path to something a person holds a grant on still appears in listings and can be opened, and opening it returns only entries they can read and folders on the path to a grant — everything else stays filtered out.

Grant the containing folder when the recipient needs the folder itself — to browse its other contents, or to create entries in it. Inheritance only means something for folders: a file has nothing beneath it to inherit, so a file grant's inheritance flag has no effect, and grants store it off whenever they are written or changed.

How resolution works

When Swarmfile decides whether you can do something to a file, it walks up the ancestor chain from that file to the repository root, collecting every grant that applies to you along the way.

Allow grants add up — they don't override each other. If more than one `allow` grant applies along that chain (say, `read` on the repository root and `write` on a specific subfolder beneath it), the *highest* level wins, regardless of which grant is closer to the file. A grant lower in the tree can raise your access above what a parent folder gives you; nothing lower in the tree can reduce it. Only a `deny` restricts access below what an `allow` elsewhere in the chain provides.

Deny always wins. If a `deny` shows up anywhere in that ancestor chain, it overrides any `allow` found elsewhere in the chain — no matter which one is "closer" to the file. This is a deliberate, conservative default: a broad `deny` placed high up in a folder tree is a reliable way to lock something down, and nobody can accidentally punch a hole in it with a more specific `allow` lower down.

Where a deny doesn't reach today. Two known limits, so you can plan around them:

- **Read-deny during an identity-service outage.** Grant checks depend on resolving who someone is; during a rare outage of the org's identity/membership service, open-mode reads are served best-effort so a whole repository can't lock out, and publishing a public release fails closed in that state. Put folders that must stay restricted in a protected-mode repository.
- **Changes reach a mounted drive in two stages.** Granting or revoking access updates the hub immediately (the web dashboard, the CLI, and new mounts see it at once), and a mounted drive is pushed the change, so its cached permission answers refresh on the spot instead of waiting for a remount. A revoke goes further: the revoked folder is removed from that drive's listing and its cached file data is deleted, then confirmed back to the organization as a purge receipt. A newly granted folder is the remaining lag: it can still be missing from that drive's file listing — and writing into it may still need a reopen — until the mount is reopened.

Org owners bypass ACLs entirely. An owner can always get into any repository, folder, or file in their org, regardless of what grants exist. ACLs govern members; they don't govern the owner.

Open vs. protected repositories

Each repository runs in one of two modes, set at the repository level:

- **Open mode** — the default for new repositories on every plan. Access is permissive: ACLs function as additive grants layered on top of general access, useful for restricting specific sensitive folders without having to explicitly grant everything else.
- **Protected mode** — access is default-deny. Nothing in the repository is accessible without an explicit grant. Use this for repositories where you want to enumerate exactly who can see what, rather than opt specific things out. Protected is always an explicit choice: the web app and Desktop App offer it next to Open (both default Open), and an API or CLI create that omits the ACL mode gets Open — on every plan, including Pro and Enterprise. Protected mode itself needs Pro or above: creating a repository as protected, or switching an existing one **into** protected mode, is refused server-side on Starter. Switching back to open is always allowed.

Switching a repository's mode doesn't rewrite existing grants — it changes what happens when no grant matches at all. See Organizations, Repositories & Members for repository lifecycle and mode selection.

Access comes from grants, not membership. Joining an org through an invite adds the org role only, so a member with no grant on a protected repository sees an empty repository — the web files view says so explicitly ("you don't have access to any files here; ask an org admin") rather than pretending the repository is empty.

On a plan with ACLs (Pro and above), the Team tab's invite form can pre-grant whole-repository **Read** or **Write** access alongside the invite, and the Team tab lists every member's whole-repository grants next to their name; expanding a member's repository there — or picking a repository where the member holds no whole-repository grant, which is how folder-only access stays visible — loads that repository's folder- and file-level grants on demand, so an admin can inspect the full per-repository picture without opening each repository (a note appears when a page was capped).

Folder- and file-level access is also managed under each repository's **Permissions** tab.

Creating at the repository root. A protected repository's top level has no parent folder for a grant to inherit from, so a new file or folder created directly at the root — or an existing entry moved there — is allowed to an org owner, the repository's creator, or a member holding a repository-wide `write` (or higher) grant.

A folder-scoped grant — even `admin` — does not authorize a root create; work inside an existing folder, or ask an owner to create the top-level folder. Overwriting an existing root file is ordinary `write` access to that file and does not need root authority — it adds no new top-level name.

In an open repository this is moot: any member who can write creates at the root like anywhere else. The web's Upload / New Folder controls, the desktop drive's root writes, and a `git push` whose commit adds or moves a name at the root follow the same rule.

Publishing a release on a protected repository requires repository-wide admin access. Org owners and the repository's creator always pass, and a repository API key passes with a repository-wide `write` grant. A release publishes the tagged commit's entire tree to an anonymous URL, and the manifest walk doesn't filter per-entry ACLs, so a member holding only a folder-scoped grant could otherwise publish a tree containing folders they can't read.

On an open-mode repository, any member who can still write the repository as a whole may publish. **Deleting (unpublishing) a release requires the same access as publishing one.** A member made read-only by a deny grant can do neither, and external collaborators can do neither.

You can't publish a release that contains anything you're denied read on. In either mode, if you hold a `read` deny on any file in the tagged tree, or on a folder along its path, the publish is refused with `403`. The deny can be granted to you directly or to one of your groups, and it counts even when it doesn't inherit, because the release's paths would reveal the folder.

A deny on a folder the tag doesn't include doesn't block the publish, and neither does a deny below `read` (a `write` deny, say). Ask an org owner or the repository's creator to publish instead: a deny never limits what they can read. Unpublishing isn't subject to this check.

A publish fails safe when your permissions can't be checked. If Swarmfile can't look up your organization permissions at that moment, it can't tell whether you hold a deny, so the publish is refused with `503` and the error code `org_unavailable`. It's a temporary refusal, so retry shortly. If your read restrictions cover a very large part of the repository, the publish is refused with `403` rather than checked file by file. Neither refusal applies to org owners or the repository's creator, and unpublishing is never refused this way.

Creating or deleting a tag requires repository-wide write. A tag names the whole tree at a commit, and it's what a release is published from. On an open-mode repository that's every member by default. On a protected repository you need a repository-wide `write` (or higher) grant, since a folder-scoped grant isn't enough; org owners and the repository's creator always pass. A repository API key needs its own repository-wide `write` grant in either mode.

Scoped API keys stay inside these rules. A branch- or path-scoped repository API key is still an API key: the repository-wide grant and the ACL rules above apply, and the scope only narrows what that key may do — it never grants access to a branch or path the key's principal could not otherwise reach. Only an owner or admin mints or revokes one. A path-scoped key is engine/API-only; a branch-scoped key keeps git/history/MRs on its branch. Scope is set at mint time; rotate by revoking and minting a new key.

Windows DACL projection

On Windows, Swarmfile repositories the effective ACL permissions it's enforcing down onto the actual NTFS-style DACL that Windows and Explorer see on the mounted drive. Permissions aren't just an app-level restriction invisible to the rest of the OS; they show up as real Windows ACL entries on the mount — what you see depends on your own role, by design:

- **Org admins and owners, and anyone with an explicit `admin` grant on a folder,** see the full list of every principal with access to it — exactly what Explorer's Properties → Security tab is built to show.
- **Everyone else** sees just their own effective access level on that folder — read, write, or none — not who else has access. The full roster is deliberately not browsable by ordinary members, the same way it isn't exposed anywhere else in the product: seeing it would let anyone map out an org's whole access structure just by opening folder properties.

Either way, the DACL Windows shows is never stale or invented — it's a live read of what Swarmfile is actually enforcing for the account looking at it.

Branch protection

Folder ACLs govern who can touch *content*; branch protection is a separate governance control over *how changes land*. The day-to-day mechanics — protecting a branch, opening a merge request, the approval flow — are in Branches & Merging. What matters here is that protection is an **org-admin control, not a repository ACL**: creating, editing, and removing protection requires an org `admin` (or owner) role, and it's enforced server-side regardless of any grant a member holds inside the repository.

There are two ways to protect a branch, and they stack:

- **A per-branch flag** — mark one named branch (say `main`) protected, with its own required-approval count.
- **Glob protection rules** — a pattern like `release/*` protects every branch that matches it, now and in the future, with the rule's own required-approval count.

When both apply to the same branch, the **effective required-approval count is the maximum** of the per-branch flag's count and every matching rule's count (each configurable from 1 to 10). A rule can raise a branch's floor but never lower it below what the branch's own flag already requires.

Protection rules can also require CI checks to pass (`--require-checks-to-pass`), independently of the approval count; the per-branch flag has no checks toggle, so to gate one exact branch on CI use a no-wildcard rule (`swarmfile protection-rule create main --require-checks-to-pass`), as the CLI reference also says. A

branch counts as checks-gated if any matching rule says so, the same OR-based resolution the approval count uses.

Only checks that reported at the merge request's exact current head are judged: the latest run of each of those must be green. A check that never reported at the head isn't counted as failing, but if no check has reported at the head at all, the gate fails closed rather than passing vacuously.

A protection rule can additionally carry a list of **auto-reviewers** — a CODEOWNERS-style reviewer list scoped to whichever branches the rule matches. Every merge request opened against a matching branch automatically requests a review from each configured principal, on top of anyone requested by hand. This is bookkeeping, not enforcement: naming someone an auto-reviewer never makes their approval individually required — the required-approval count above is still the only lever for that.

On a protected branch, a direct `merge` is refused, and so is **every direct write** — saving, editing, renaming, moving, deleting, creating, uploading from the web, staging/committing, rolling back, resolving a conflict, auto-resolving one, or reflecting LFS content — with `409 protected_branch`; the change has to go through a **merge request** that collects the required number of approvals before it can land.

This is how protection forces review: there's no admin-only "just merge it" bypass on the protected branch itself, and a non-admin also can't archive a protected branch out from under the rule. Folder ACLs still govern who may attempt a write, but a grant cannot override protection.

Limitation. Self-review is blocked *per user* — the person who opened a merge request cannot approve it. That raises the cost of a unilateral change and creates an audit trail, but it is an identity check, not a cryptographic proof that two different humans reviewed the work: a second credential controlled by the same person can still approve. Treat the approval count as a control that raises that cost, not as a guarantee of independent review. Every protection change and every quarantine/ACL action lands in the activity feed — see the audit-log list in Operations.

Plan requirements and downgrades

Folder ACLs and Windows DACL projection are Pro plan and above. They're enforced server-side — the server validates the org's plan entitlement before honoring an ACL operation, so plan gating can't be worked around from the client.

If an org is later downgraded below the tier that supports ACLs, existing grants keep working and nothing is silently opened up. The only thing that changes is:

- **Revoking or narrowing a grant** — always available, on any plan. You can never be stranded unable to fix an over-broad permission just because the org downgraded.
- **Adding a new grant** — blocked on a downgraded plan, until the org upgrades again.

This mirrors the downgrade behavior for identity features — see Identity — and is intentional: a downgrade should never leave you less secure, only less able to add new access.

For the broader security model — org isolation and storage separation — see Security.

Public access requests

A public repository accepts **access requests** from its public page by default (**Repository settings** → Public access requests turns them off and on). Any registered Swarmfile user with a verified email — no organization membership — can then ask for one of three levels:

- **Comment access** grants `comment` on the repository's root `Contributions` folder — the same external collaborator tier, without a seat.
- **Contributor access** grants `comment_upload` on that folder: new files only, no overwrite or delete. It is not `write`, and it never grants access outside the folder.
- **Membership** mints the ordinary organization invitation (role `member`), which uses a seat; on a protected repository the invitation also carries a whole-repository `write` grant so the new member can see it.

Approvals run the same checks as an owner-minted guest invitation: membership consumes a seat (bounded by the plan's seat cap), and a guest or contributor grant must still land on a repository that is public. Public-repository guest and contributor grants are **free on every plan** — they consume no collaborator allowance and are never billed — so a Free organization's public page offers all three levels. A refusal (a full seat cap, or a repository that stopped being public) leaves the request pending with the reason, so fixing the cause and approving again works.

The repository's owners and admins decide each request from the **Repository settings** tab; the queue there and a badge on the Repository settings tab show pending work. Guests and contributors an approval creates appear in External Collaborators & Sharing like any other external collaborator, and revoking them there removes the grant. Approving a guest or contributor request after the repository stops being public (or is archived) is refused — use the sharing flow or restore the repository instead.

The requester sees the status on the repository page and in My requests, can withdraw a pending request, and gets an email on the decision. A declined requester can ask again, with a short message explaining what changed. Requests one person may have pending in an organization at once are capped; when a repository is deleted, its requests are removed with it, and old requests age out after 180 days.

Self-Hosted Seed Nodes

Swarmfile is LAN-first: peers on the same network exchange file data directly and only fall back to cloud storage when no local copy is available. A self-hosted seed node extends that further by letting a machine you own — a NAS, an old workstation, a dedicated on-prem box — participate permanently in the swarm as a warm cache tier.

What a seed node is

A seed node is just a machine running Swarmfile with seeding turned on. It joins the peer-to-peer swarm like any other client, but instead of only holding what a given user happens to have opened, it holds a persistent, shared cache that the rest of your office can pull from.

The practical benefit shows up for offices that already have a lot of repository data sitting on local infrastructure: rather than every workstation re-fetching the same large files from cloud storage over and over, they fetch from the seed node on the LAN instead. That's faster, and it takes load off your egress and off the cloud fallback path entirely for anything the seed already holds.

A seed keeps two things in sync. It walks the repository and fetches **every** block, mirroring the whole repository to cloud storage — so the cloud copy is always complete and no editor ever has to be the one who pulls a file from the cloud first.

Locally, it keeps a warm cache bounded by `SWARMFILE_CACHE_MAX_BYTES`: once full, the block store evicts the least-recently-used blocks, and an evicted block is simply re-fetched on demand (from a peer or the cloud) the next time someone reads it.

Size that budget to the working set you want resident — the template ships 100 GiB; a NAS serving a multi-TB production wants 1 TiB or more. A cache smaller than the repository still helps (hot data stays local), but the bigger the cache, the more of the office's reads never touch the WAN at all.

One thing worth being precise about: on the standard managed (encrypted) tier, what the seed node caches is ciphertext — reading it still needs a decryption key fetched live from the hosted hub. A seed node is a warm, on-premises cache that speeds up your office and reduces cloud round-trips, not an independently readable offline copy of your data. See Data Portability & Offboarding for what that does and doesn't mean if the hosted service is ever unreachable.

Plan requirement

Running a seed node requires the **Pro plan or above**.

It's also mutually exclusive with **cloud-only mode**: a mount can't be both a pure cloud-only client and a seed node at the same time. If a machine is configured for cloud-only operation, you'll need to turn that off before enabling seeding on it.

The hub also gates whether a self-hosted node is allowed to announce itself to the swarm at all, based on your org's plan entitlement. If a seed or NAS node is running under a plan that doesn't include the seeding entitlement, its announces are refused outright — a hard block rather than a warning you might miss in a log.

Against a self-hosted `swarmfile-server`, a seed node works the same and there is **no plan gate** — the community server has no plans, so any org key may run one. (The hosted **Office caches** feature around it — the enrollment wizard, the fleet dashboard, weekly reports — stays on the hosted service.) A server-side swarm posture (`SWARMFILE_SERVER_P2P`) reaches a running seed on its periodic check, so a `sealed` / `off` flip stops it serving on its own. See Self-hosting → Peer-to-peer.

Enabling seeding

For a machine that's already running the regular Swarmfile client, toggle seeding with:

```
swarmfile seed enable
```

and to turn it back off:

```
swarmfile seed disable
```

Both commands restart the engine to apply the change, so expect a brief interruption on that machine.

For a dedicated, headless machine — a NAS or a box that isn't otherwise being used as anyone's workstation — run the `swarmfile-engine` daemon directly instead of the Desktop App (the Desktop App is just a GUI wrapper around the same engine), then run `swarmfile seed enable` against it exactly as above.

> Don't confuse this with the standalone `swarmfile-seed` binary. Despite the name, it isn't a way to run a seed node — it's a one-shot admin tool that uploads a single local file straight into the hub and exits, useful for scripted bulk ingestion. See swarmfile-seed for its reference.

Enrolling an office cache from the dashboard

For a machine that has no Desktop App — the NAS, the edit-bay Mac mini, the small box next to the switch — set it up from an env file instead. The dashboard generates that file for you:

1. Open **Settings → Office caches** and click **Add an office cache**.
2. Name the office (a stable, lowercase id like `london`) and pick the repository the cache serves.
3. The wizard shows the complete `seed.env` once — it contains a freshly minted repository-scoped API key. Save it on the cache machine as `/etc/swarmfile/seed.env` (mode 0600), install the engine, and start the engine service with that env file; `SWARMFILE_SEED_MODE=true` is already in it, so there's no toggle to run.

The generated file carries every value a headless seed needs, including the scope (`SWARMFILE_ORG_ID` and `SWARMFILE_REPOSITORY_ID` alongside `SWARMFILE_OFFICE_ID`) — a seed whose scope is missing pins nothing. The credential is a repository-scoped API key: no interactive login on the box, and no refresh token to rotate away. It is an **organization** credential, not a personal one — it keeps working if the person who created it leaves the team, and it stops only when someone revokes it under Settings → API keys (at which point the cache shows offline here and the offline alert below fires).

One cache serves one repository; run one cache per office per repository. The same page lists every cache with its online/offline state, last announce, and the bytes it served this week; a weekly email report goes to the organization's owners and admins, and a cache that goes silent for a day triggers one alert email per outage. Both can be muted under **Settings → Notifications** ("Weekly office cache report", "Office cache offline for a day").

Installing on Windows

On Windows, run the seed as a scheduled task that loads `seed.env` and supervises the engine. Save the generated `seed.env` somewhere the SYSTEM account can read (e.g. `C:\ProgramData\Swarmfile\seed.env`), then create the wrapper and the task from an elevated PowerShell:

```
$wrapper = "$env:ProgramData\Swarmfile\run-seed.cmd"
@"
@echo off
setlocal
for /f "usebackq eol=# tokens=1,* delims==" %%A in ("C:\ProgramData\Swarmfile\seed.env") do set "%%A=%%B"
:loop
"C:\Program Files\Swarmfile\swarmfile-engine.exe"
timeout /t 5 /nobreak >nul
goto loop
"@ | Set-Content -Path $wrapper -Encoding ASCII

$action    = New-ScheduledTaskAction -Execute "$env:SystemRoot\System32\cmd.exe" -Argument "/c `"$wrapper`""
$trigger   = New-ScheduledTaskTrigger -AtStartup
$settings  = New-ScheduledTaskSettingsSet -AllowStartIfOnBatteries -DontStopIfGoingOnBatteries -ExecutionTimeLimit ([TimeSpan]::Zero)
$principal = New-ScheduledTaskPrincipal -UserId "SYSTEM" -LogonType ServiceAccount -RunLevel Highest
Register-ScheduledTask -TaskName "SwarmfileSeed" -Action $action -Trigger $trigger -Settings $settings -Principal $principal
Start-ScheduledTask -TaskName "SwarmfileSeed"
```

The task starts the engine at boot as SYSTEM — no sign-in needed — and the wrapper restarts it if it exits, so a crash recovers headlessly. To remove the seed, run `Stop-ScheduledTask -TaskName SwarmfileSeed` and `Unregister-ScheduledTask -TaskName SwarmfileSeed -Confirm:$false`, then delete the wrapper; the `seed.env` holds a credential, so remove it yourself when you mean to. After the first announce the cache appears on the Office caches page like any other.

Deliberate LAN shard placement

By default, which peer ends up holding which erasure-coded shard of a file is opportunistic — whoever happened to fetch or cache it holds it. That's fine for most teams, but it means fault tolerance is a byproduct of usage patterns rather than something you can rely on.

If you want guaranteed redundancy across your office instead — so that losing any single machine doesn't put a file's availability at risk — turn on deliberate placement:

```
swarmfile ec-placement enable
```

This rendezvous-hashes each shard across your office roster at a replication factor of 3, so every shard has three specific, deterministic homes on your network instead of landing wherever caching happened to put it. Like `swarmfile seed enable`, this restarts the engine.

Reach for this when you have a real office roster of machines and want predictable, guaranteed fault tolerance — not just "probably cached somewhere."

One thing worth being precise about: a seed node's own behavior doesn't depend on whether deliberate placement is on. A seed node always walks the full repository tree and fetches every block, unconditionally, so the cloud copy is guaranteed complete — that's a stronger, whole-repository guarantee than deliberate placement's replication-factor-of-3, not something deliberate placement adds to.

What stays on the seed's own disk is its warm cache (bounded by `SWARMFILE_CACHE_MAX_BYTES`, see above), not an unbounded mirror: "complete" here means every block has been fetched and mirrored by the seed, with recently used blocks resident for LAN reads. Deliberate placement is what upgrades your *regular* (non-seed) peers from opportunistic to guaranteed coverage; if every machine in your roster is already a seed node, the cloud always has everything and nothing is left to chance for recovery.

This doesn't require a literal office LAN. `office_id` is a grouping label, not a network check — any set of peers that share one `office_id` join the same placement roster together, whether or not they can reach each other on a local network at all.

That's the pattern for a cloud-hosted render farm: set a matching `office_id` on each node, and leave `--lan-from-office` off (that flag exists to fold regular peers on a real LAN into the roster when mDNS can't discover them; it's not relevant when there's no LAN). Leaving it off also keeps these nodes correctly classified as WAN traffic rather than wrongly exempted from the local bandwidth throttle — egress is never billed on any plan either way. See Deployment Topologies for the render-farm case end to end.

For context, files are erasure-coded Reed-Solomon 10+4 by default, and that scheme is adaptive: it exists primarily to mask WAN latency, so once two or more LAN peers are present for a repository it can automatically back off from the full redundancy it uses over a pure cloud connection. Deliberate placement overrides that automatic behavior — when you explicitly want guaranteed office-wide redundancy rather than whatever the adaptive default decides, `ec-placement` takes the decision out of the engine's hands. For more on the underlying security and isolation model, see Security.

Inspecting shard placement

To see exactly where a given file's shards actually live, run:

```
swarmfile shards <path>
```

This shows, per shard: which peer holds it, whether it's a data shard or a parity shard, and whether the machine you ran the command on currently holds a copy itself. It's the tool to reach for when you want to confirm deliberate placement actually took effect, or just to understand a file's current redundancy without guessing.

Security

This page is the deep-dive companion to the Security overview on the marketing site. It covers how encryption, integrity checking, and network isolation actually work, and it's explicit about what's available today versus what's still on the roadmap.

> Running a formal vendor security review? See Security Architecture — the threat model, a who-can-decrypt matrix across the managed and E2E tiers, key-custody specifics, and where each guarantee's limits are, written to hand to a security team.

Encryption

Every **private** repository on a paid plan (Starter and above) has its block contents encrypted at rest with AES-256-GCM by default. Each repository gets its own key, wrapped by a hub-held key-encryption-key (KEK). Desktop writes are encrypted on your machine before upload; small files uploaded from the web are sent over TLS and encrypted by the hub the moment they arrive — so stored private blocks are always ciphertext.

Because the hub holds the KEK for a managed repository, Swarmfile can decrypt when it needs to (for example, to generate a preview or thumbnail). So managed encryption protects your data at rest from the storage layer, not from Swarmfile itself — for that stronger guarantee, use the end-to-end tier below.

Two cases are stored unencrypted (plaintext). The **Free plan** is one: every repository it creates is plaintext at rest, public or private, the trade-off for no-card, no-cost storage. A private Free repository is private by access control, not by encryption — the hub can read it, and its blocks are never peer-served. The other case is a **public repository** on any plan — anonymous public releases can only be served from unencrypted blocks, so a repository has to be plaintext to be public.

Encryption tier and visibility are both fixed at repository creation and never change afterward, so a repository created on the Free plan or as public stays unencrypted even after the org upgrades; a new repository created with a paid encryption tier gets managed encryption normally. If encryption at rest matters for a given repository, create it on Starter or above with managed or end-to-end encryption.

Public exposure and the anonymous clone boundary

Public **exposure** is still scoped by the publisher: a committed `.swarmfile/publicignore.yml` keeps matched paths — and derived data such as CI logs — out of the public release view, enforced as each release is published (share links are not covered; see Security Architecture).

The same policy sets the anonymous git history boundary: while any active branch's committed tree carries a `.swarmfile/publicignore.yml` that hides a path in it, any tag's committed tree does, or a ready (published) release hid paths, anonymous clone is refused outright while that committed head, tag or ready release hides them, though a release already published keeps the contents it was built with. The boundary evaluates both committed trees a branch can advertise (a dirty-source fork's live head and its advertised commit tree can differ) and every tag's changeset tree.

A path that exists only in an older commit is not retroactively hidden. There is no filtered clone, because git history is content-addressed and hiding paths at the same commit ids would contradict the public history.

An anonymous clone serves commit messages and the committed `.swarmfile/publicignore.yml` verbatim (filtering them would change commit ids); the web commit list and federation commit/activity feeds redact path names the policy hides, but a clone does not — never put a secret in a public commit message.

An invalid or unreadable policy, or a boundary read/walk error, refuses anonymous clone (fail closed). A change takes effect within about a minute.

The boundary is anonymous-only; member ACLs still apply, so a restricted folder can still refuse a member's clone.

Publishing still requires anonymous cloning off while a release would hide paths, and the web wizard offers a one-click **Turn off anonymous clone**; once the release is ready, turning it back on does not re-expose the paths while the ready release or the policy still hides them.

Never store secrets in a public repository.

git-LFS on an encrypted repository

git-LFS on an encrypted repository uses the transfer agent. When a managed repository is used as a git-LFS server, uploads must go through the Swarmfile transfer agent, which stores the object as encrypted blocks like any other repository content (the Desktop App's built-in agent seals client-side; the standalone `swarmfile-lfs` agent's blocks are encrypted by the hub on write). The stock direct paths are refused with the typed `lfs_agent_required` error, so no new object accepted through the current paths is stored without application-layer encryption.

A presigned single PUT that was admitted before the gate landed can still complete, so it counts as an older object rather than a new one. A multipart upload session created before the gate cannot complete: its part and complete requests answer the same `lfs_agent_required` 409 and abort the session, so it never assembles a plaintext object.

git-LFS is refused on end-to-end-encrypted repositories. An unencrypted repository remains plaintext by definition on every path.

Objects a managed repository stored through the old direct paths before this rule are converted by an operator-run migration, which verifies the encrypted read-back before deleting each plaintext object; until it reports clean for that repository, don't assume every older object is encrypted. The git-LFS guide has the table and the older-objects note.

Opt-in end-to-end (customer-managed) encryption

For repositories that need a stronger guarantee than a hub-held key, you can opt into end-to-end encryption. Instead of wrapping the repository key with a hub-held KEK, each member's copy of the key is wrapped to their own device keypair (X25519). The key never reaches Swarmfile's servers in plaintext at all, and grants happen client-side.

This is a content-only guarantee:

- File and folder **contents** become unreadable to the server.
- Filenames, folder structure, and file sizes stay server-visible — search, ACLs, browsing, and quota accounting keep working normally.
- Stored blocks are named by a hash of their plaintext chunk (so identical content deduplicates). No whole-file hash is kept for E2E content — see Security Architecture for exactly what that does and doesn't reveal.

The trade-offs are real and worth knowing before you turn it on:

- Server-side preview and thumbnail generation don't work for E2E content, because the server can't decrypt it.
- New members are enrolled into E2E repositories automatically once an existing member's client has been online to complete the key wrap — no manual key ceremony. An optional always-on key-granter closes even that narrow window, delivering new-member and new-device keys automatically on a five-minute cadence at worst. See End-to-End Encryption Setup for the granter setup.

Losing every device with a wrapped copy of a repository key would mean losing the data permanently, so E2E repositories should be created under an org recovery key. That key is split using Shamir secret-sharing escrow (2-of-3 by default), so no single lost device or credential is catastrophic.

For the step-by-step — setting up the recovery key, creating an E2E repository, enrollment, and the recovery ceremony — see End-to-End Encryption Setup.

Content integrity

Every block is BLAKE3-verified on arrival. Corrupt data — from a misbehaving peer, a bad transport, or disk corruption in transit — never reaches the application layer.

Erasure coding

Blocks are protected with adaptive Reed-Solomon 10+4 erasure coding: any 10 of the 14 shards reconstruct the full data, so up to 4 slow or offline peers never block your read. This page keeps that summary brief — for the admin-facing detail on deliberate shard placement across your seed nodes, see Self-Hosted Seed Nodes.

Private P2P swarm

The peer-to-peer transport runs in a private swarm. Joining it requires a pre-shared key and a custom protocol identifier — a random host on the public internet can't dial into your blocks even if it somehow learned a peer's network address. Peers that connect directly do learn each other's network addresses, which is inherent to a direct connection; an org that can't accept that can turn on cloud-only mode (below), which disables peer-to-peer entirely.

Cloud-only (hub-only) data-plane mode

Some enterprises need to block UDP/QUIC entirely, or eliminate peer-IP exposure between employees' machines outright. An org-level policy — or an engine-local `SWARMFILE_HUB_ONLY` toggle — disables peer-to-peer entirely: QUIC never binds, there's no peer discovery, no gossip, and `swarmfile doctor` reports clean. All reads fall through to HTTPS against our cloud storage instead.

This is enforced at startup and re-checked every time a machine switches which repository or organization it's working in, so a person who works across multiple orgs never inherits the previous org's policy; a check that can't complete (a hub outage, for instance) fails closed rather than silently allowing peer traffic.

This is admin-gated (owner or admin), and toggling it is itself an audited event. See Operations for where that shows up in your activity log.

Removing access

Revoking access reaches every affected computer, not just the next sign-in. Remove someone from the org, revoke their access to a folder, or revoke a machine credential, and each affected drive drops the revoked content from its local cache and refuses further opens of it — including while offline, because the refusal is persisted locally the moment the notice arrives.

A machine that was offline takes the same step when it reconnects. Organizations can also bound how long a *disconnected* machine may keep serving already-downloaded content at all (default: seven days); past that window the drive stops serving cached bytes until it next reaches the hub.

Removed members' own unsaved work stays on their machine and uploads if their access is restored. Owners can see any revocation still pending under **Settings → Access revocations**.

Ransomware detection

The system watches for a suspicious pattern inline, at high frequency: a large number of *distinct entries* overwritten (by content-hash) in a short window. Writes are counted per user and machine, with a per-user total across all of that user's machines as a backstop, and a quarantine applies to the user, not just one machine. It counts distinct entries, not raw history rows, so re-saving the same file repeatedly doesn't inflate the signal.

The response is two-tier, and crossing the first line does **not** block anyone:

- **Alert bar** (a configurable number of distinct entries in the window) — the incident is recorded as alert-only and org owners are notified for review. Nothing is blocked.
- **Block bar** (a higher multiple of the alert bar) — this is what actually hard-quarantines the user, either on a genuinely large sweep or when an already-alerted user keeps going. Only past this bar do writes stop.

Writes are attributed to the machine that made them, whether they arrive as individual saves, a staged commit or a `git push`. Restores, checkouts, reverts and merges the hub applies from content the repository already held don't count as per-machine overwrites (their history rows carry no machine id), so a legitimate bulk recovery doesn't trip the per-machine bars. Both bars are configurable per organization (owners and admins, under **Settings → Quarantine → Detection thresholds**) and ship with sensible defaults; orgs whose workflows routinely rewrite many files in place can raise them.

What a quarantine blocks

When a user is quarantined, every content-write path — a save, a conflict resolve, a changeset commit or checkout, a `git push` apply, a branch or merge-request merge, LFS reflect/upload, and a direct web upload — is refused with the `quarantined` reason, and lock acquire and lock renew refuse too, so they can't keep working even in the middle of holding a lock. A repository or agent API key acting for that person is refused as well (the key inherits its owner's block); a human-triggered hosted-CI job inherits the actor's block; an actorless scheduled job must be stopped or revoked when containing an account.

Reads keep working. New saves stop on that computer: on macOS and Linux a new write-open (including an editor's save-by-rename) fails with the `quarantined` reason in **Why did my save fail?**, and on Windows the drive stays mounted while ordinary writes are accepted locally and held, though a rename is refused — an app

that saves by renaming reports a failed save. The drive is paused, not unmounted, and `swarmfile status` says `WRITES PAUSED`.

Clearing and auditing a quarantine

Work already queued on that machine is held and resumes when an owner or admin releases the block; an edit the app could not save is still the app's to retry. Alert-only incidents that no one acts on auto-expire after 7 days; blocks never auto-expire and require an admin or owner to clear.

Releases record whether the incident was a false positive, and the Quarantine view publishes the resulting block and alert false-positive rates with their sample sizes; manual quarantines are excluded from those rates because there was no detector decision to judge. A quarantine is scoped to the organization where it was raised; the same user in a different organization is unaffected.

See Operations for the release, rate, review-request and org-wide rollback workflow.

No kernel extension on macOS

The macOS installer ships a kext-free build using FUSE-T, which mounts entirely in user space. There's no kernel extension, no dropping into Recovery Mode to lower security settings, and no system-extension approval prompt on Apple Silicon.

Authentication and membership

- **Sign-in** runs on Swarmfile's own OIDC IdP, and every account can add TOTP multi-factor authentication under Account → Security.
- **Organizations can require MFA and/or a verified email** — see Authentication policy. Enforcement is checked on every org-scoped API request with a grace window and a typed refusal that names the remedy; the owner always keeps an off-switch. A personal access token snapshots its session's factors and email assertion at mint time, so a token can never carry more than the session that created it.
- **SSO**: per-org OIDC (Entra/Okta), SCIM 2.0 provisioning, and an on-prem LDAP sync agent; an org can require that members sign in through its own identity provider, and machine credentials are held to the same binding.
- **Machine credentials**: repository-scoped API keys for headless engines and personal access tokens for scripts; both are revocable and audited. API keys are walled to one repository, can be branch/path-scoped at mint time, and can carry their own budget; a personal access token acts as its owner and can never carry more than the session that minted it.

On the roadmap

- **SOC 2** is on our roadmap; we have not yet begun a formal audit. In the meantime, the team answers security questionnaires directly and walks customers through the architecture on request.
- **SIEM forwarding** — a managed Splunk, Datadog, or S3 stream of audit, access, and quarantine events — is planned. Today, a Pro-plan owner can export the activity feed to CSV (up to 100k rows per export), and outbound webhooks push events to an endpoint you control when they are recorded, typically within seconds (the hourly tick is a backstop).
- **Legal hold & retention immutability** — a per-entry legal-hold flag that overrides retention and trash purge — is planned.
- **DLP & egress controls** — watermarking, share-link domain allowlists, and export policy gating — are planned.

If any of these are a hard requirement for you today, talk to us directly — for details on the permission model that is live today, see Permissions.

Security Architecture

This is the deep technical companion to Security — written to be handed directly to a security team running a vendor review. It states plainly what each layer protects against, who can decrypt what, and where the limits of that protection are. If your review needs something not covered here, contact us and we'll answer specifics.

The one-paragraph version

On a **paid plan** (Starter and above), **private** repository content is encrypted with AES-256-GCM before it is stored — on the machine that writes it for anything saved through the drive, and by the hub for browser uploads to a managed repository (the hub holds a managed repository's key; an end-to-end repository's key never reaches it).

Peer-to-peer nodes and our cloud object storage see only ciphertext addressed by a BLAKE3 content hash, with a few documented exceptions: the git-LFS direct-upload path (see below); server-side **video proxies** and the **whole-file copies** assembled to serve browser downloads (both retained short-term — see "Worth knowing about server-side previews" below); and any **public** repository, which is stored unencrypted by design — see Unencrypted tier.

Free-plan repositories use that tier because they have no other, whether they are public or private: a private Free repository is private by access control, not encryption, and its plaintext blocks are never peer-served.

A paying org can still choose the tier deliberately for a public repository that needs the anonymous-download paths (which read unencrypted content only). Because a public repository's content is public, its plaintext blocks may also be served to peers on the org's private swarm — peers reach only what the anonymous read routes already serve, peer serving is read-only, and every fetched block is BLAKE3-verified before it is used.

A paid org's **private** repositories are encrypted — managed by default, E2E opt-in — which is also what makes them eligible for peer-to-peer transfer.

The **only** party that can turn ciphertext back into your files is whoever holds the repository key — and *you choose who that is*. On the default **managed** tier that party is the Swarmfile hub, so we can give you server-side previews, search, and proxies — managed means *encrypted at rest with hub-held keys*, not zero-knowledge. On the opt-in **end-to-end (E2E)** tier the key never reaches our servers in a form we can open, so we structurally cannot read your content. Both tiers are fully supported; the difference is a deliberate trade you make per repository, not a weakness in one of them.

The **Free** plan is the one plan with no at-rest encryption at all: its repositories — public or private — are stored **unencrypted**, with no repository key — the trade-off for no-card, no-cost storage. See Unencrypted tier below. On Starter and above, a **private** repository gets managed encryption by default (E2E opt-in); a **public** repository is unencrypted-only on every plan, because public releases and archives still read unencrypted content only. The unencrypted tier is therefore every Free-plan repository plus a paying org's deliberately-public repositories.

Threat model — what each tier defends against

Adversary	Managed tier	E2E tier	Unencrypted tier (every Free repository; a paid org's explicit public-repository opt-out)
A malicious/curious **peer** on your swarm	✅ sees only ciphertext + BLAKE3 CIDs	✅ same	❌ sees plaintext of a **public** repository (peers see only what anonymous readers see); a private Free repository is never peer-served
The **storage provider** or anyone who exfiltrates a storage bucket	✅ ciphertext at rest, no keys co-located	✅ same	❌ plaintext at rest
Someone who **subpoenas or compromises the Swarmfile hub**	⚠️ hub holds the wrapping key and *can* decrypt	✅ hub holds only un-openable wrapped blobs	❌ nothing to compromise — already plaintext
A **Swarmfile employee** with production access	⚠️ same as above — possible, audited, not prevented	✅ prevented by construction	❌ same as above
A **non-member of the repository** (incl. other orgs, other members)	✅ blocked by the block ACL on every hub fetch	✅ blocked by ACL *and* has no wrapped key	✅ nothing confidential to reach (public by design); hub fetches stay ACL-gated, and swarm peers can only fetch blocks the anonymous read routes already serve
A **lost password / lost device**	✅ recoverable (hub re-wraps)	⚠️ recoverable only via the org recovery key	✅ no key to lose

git-LFS objects. The block properties above describe repository content written through the drive/engine. A managed repository used as a git-LFS server now follows the same rule: uploads must use the Swarmfile transfer agent's chunked block path, and those blocks are encrypted at rest (the desktop agent seals them client-side; the standalone agent's plaintext blocks are encrypted by the hub on write).

The stock direct paths — a plain `git-lfs push`, a direct object PUT, or presigned multipart — are refused on a managed repository with the typed `lfs_agent_required` error instead of storing one plaintext object. An unencrypted repository remains plaintext by definition on every LFS path, and E2E repositories refuse LFS outright.

Older git-LFS objects. A managed repository that accepted direct uploads before that rule can still hold the old plaintext shape. An operator-run migration reads each object, writes it through the encrypted block path, verifies the read-back bytes, records the oid mapping, and only then deletes the plaintext object; it is dry-run/resumable and reports when the repository has no plaintext objects left. Until that report is clean for a repository, this page's "ciphertext at rest" property describes new objects and the migrated set, not necessarily every historical one.

The two ⚠️ rows on the managed tier are the entire reason the E2E tier exists. If "the vendor can technically decrypt our footage" fails your review, the answer is the E2E tier — not a promise about the managed one. The unencrypted tier is a different trade entirely: it is not a weaker version of managed encryption, it is the deliberate absence of at-rest encryption in exchange for a free, no-card tier — access is still gated by the same block ACL every other tier uses, but there is no encryption layer to defend the *storage* boundary at all.

How block encryption works (both tiers)

Encryption of the bytes is identical on both tiers; only *custody of the repository key* differs.

- Each repository has one 32-byte **repository master key**.
- Every content block gets its own key, derived from the repository key and bound to the block's content (the CID is the BLAKE3 hash of the *plaintext* chunk), so a key is never reused across blocks.
- The block is sealed with **AES-256-GCM** and keyed in object storage by its CID. (The exact block format is in the Storage Format spec.)
- On read, the reader re-derives the key, decrypts, and re-verifies the BLAKE3 CID before any byte reaches an application — so a tampered or corrupted block is rejected, not served.

Content-defined chunking means identical content across file revisions shares a CID and deduplicates, and the per-repository key scoping means dedup never crosses a repository boundary.

Managed tier (default)

The repository master key is generated once, **wrapped by a hub-held key-encryption-key (KEK)**, and stored as ciphertext in the org's metadata store. The hub unwraps it only when it needs to act on your behalf — serving a download, generating a thumbnail, or transcoding a preview proxy.

What this protects: your data at rest against the storage layer, a leaked bucket, a lost disk, or a peer on your swarm. What it does **not** protect against: the hub itself, since the hub custodies the KEK. That is the standard posture of essentially every managed B2B file service (it is what lets us offer previews and search) — and it is why, between two members of the same org, the confidentiality boundary is the **block ACL**, not cryptography. Every block fetch is authorized per CID against the repository's permission model; encryption is the at-rest guarantee, the ACL is the member-to-member guarantee.

Web uploads (small files sent from the browser over TLS) are encrypted by the hub the instant they arrive, before anything is written to storage — so managed data is always ciphertext at rest, matching what the desktop engine writes client-side.

Worth knowing about server-side previews. Because the hub *can* decrypt managed content, it renders previews on the server: thumbnails, video poster frames, point-cloud images, image/PDF previews, and scrubbable video proxies. The rendering happens in memory. What gets stored depends on the kind of preview:

- **Thumbnails, poster frames and point-cloud preview images are stored encrypted** under the repository's key, with the same AES-256-GCM block encryption as your files, and decrypted only to serve a request that already passed the file's read check. The hub can still decrypt them, exactly as it can decrypt the source files. That is what managed encryption means.
- **Still stored unencrypted:** scrubbable **video proxies** (a 720p transcode, kept up to 30 days), the **whole-file copies** assembled to serve browser downloads and previews (kept up to 24 hours), and the short-lived inputs and outputs of a preview render, deleted when the render finishes.

So a compromise of the storage layer could still expose a video proxy or a recently downloaded file, even though the source blocks stay sealed. The **E2E tier** removes this residual entirely: on E2E the hub can't decrypt, so no server-side previews or derived copies exist. If preview confidentiality at rest matters for your content, choose E2E.

Key rotation

Two things can rotate: the KEK and the repository key. Rotating the KEK only re-wraps the stored repository keys. The keys themselves don't change, so no block needs re-encrypting.

A managed repository's key can also be rotated — by its owners or admins in **Repository settings → General**, or by the repository's creator through the API. This adds a new key *generation* and makes it current.

Earlier generations are kept, so every block written before the rotation stays readable, and each block records which generation sealed it (see Storage Format). New content is sealed under the new generation by every client that knows about key generations — the Desktop App, the hub itself, and the git-LFS agent.

A rotation doesn't re-encrypt content that's already stored, though, and an old generation is never destroyed. So after a suspected key compromise, content written before the rotation is still protected only by the old key.

An **E2E** repository rotates the same way — the generation model is identical — but the hub never holds or serves the key, so a member's own device performs the rotation: it mints a new generation and re-wraps it to every remaining device plus the org recovery key, and the hub only stores the envelopes the device produced. The same caveat applies: existing content is not re-encrypted, so it stays readable under the old generation. Revoking a single device is separate — the hub revokes that device's wrap(s), and the wrapped-key route then answers it a typed `key_access_revoked`.

Key-encryption keys rotate without making stored data unreadable: new values are wrapped under the active key and reads accept every retained key, and a rotation re-seals stored values under the new key before the old one is retired. The same discipline covers the managed-tier repository key and every credential the hub stores for a customer.

Unencrypted tier

The Free plan — no card required — is restricted to a third tier that is not a variant of managed encryption: it has **no repository key at all**. Blocks are written and stored exactly as they arrive, no AES-256-GCM sealing, no KEK, nothing to wrap or unwrap. A **public repository on any plan** also has to be on this tier, for the same underlying reason: anonymous public serving only works from unencrypted blocks. This is the tier's design, chosen deliberately so a free tier can exist without a per-repository key — and, for public repositories, because there is nothing to hide from anonymous visitors anyway.

What stays the same as every other tier: the **block ACL**. An unencrypted repository's content is not encrypted, but it is not *unauthenticated* either — every hub block fetch is still authorized per CID against the repository's permission model, the same gate managed and E2E repositories use. Plaintext is peer-served only for **public** repositories, and then only with content the anonymous read routes already serve, while a private repository's cleartext is never served there. Plaintext removes the storage-at-rest guarantee; it does not remove access control on the hub.

Encryption tier is fixed at repository creation and immutable afterward. A repository created on the Free plan stays unencrypted even after the org upgrades to a paid plan — upgrading does not retroactively encrypt existing content. A new **private** repository created after upgrading gets managed encryption normally, the same as any Starter/Pro repository. If a specific repository needs encryption at rest, create it on a paid plan; don't rely on a later upgrade to change an existing Free-plan repository's guarantee.

The Free plan can create public repositories and private unencrypted repositories, but it cannot encrypt either: a private Free repository is access-controlled plaintext, and its blocks are never peer-served.

On a paid plan a repository is private and managed-encrypted by default, and only enters this tier if it is created as **public** — visibility is fixed at creation, never flipped, so this is a create-time choice: the CLI's `swarmfile repository create <name> --public` (which pins the unencrypted tier with it) or the create API's `"visibility": "public"` + `"encryptionTier": "plaintext"`. The Desktop App's New Repository dialog and the web dashboard both create private repositories by default, with a **Publish publicly** card for a repository meant to be open (unencrypted repositories remain the Free-plan shape there).

See Permissions for how repository visibility works; it's a separate setting from, but always paired with the unencrypted tier.

Keeping paths out of the public view

An unencrypted public repository's content is public by design, but a publisher chooses *which* content actually goes public. A release reads a `.swarmfile/publicignore.yml` committed at the tag's tree and leaves matched paths out of the public release entirely — no browse entry, raw stream, download, archive, file search or Explore — while the policy hides nothing from members; ACLs still apply.

The same policy can mark platform-derived data (currently CI-run logs) member-only. The policy file itself is never published, an invalid file fails the publish instead of publishing everything, and a change applies to future releases only, so an already-published release needs a new tag to pick it up.

The full syntax and the enforcement points are in Publishing Releases.

Two other boundaries worth stating plainly: **share links are not covered**, so a link still serves what it points at, and an org policy can only *tighten* this exposure, never loosen it. It is a publisher control, not a retroactive takedown: copies already downloaded stay where they are.

Anonymous git history has the same boundary, as a refusal. Git history is content-addressed: a filtered clone at the same commit ids cannot exist, because the filtered tree has different OIDs and would contradict the public history the project page shows. So anonymous clone is **refused** — not filtered — while any active branch's committed tree carries a `.swarmfile/publicignore.yml` that hides a path in it, or while any ready (published) release hid paths (`public_ignore_hidden_count > 0`); an already-published older release keeps the contents it was built with.

The same gate covers every anonymous git read (`/public/git/state`, refs, commits, trees, blobs and LFS), so dispatch's clone-ticket mint stops minting and an in-flight ticket is re-checked on its next object read; the verdict is cached for about a minute.

The git refs themselves are answered `Cache-Control: no-store` and served live, not from an edge or browser cache. A policy that hides nothing does not block. An invalid or unreadable policy, or a read/walk error while checking, refuses anonymous clone (fail closed).

The boundary is anonymous-only: member git reads are ACL-scoped regardless, so a folder restricted for a member can still refuse their clone. Publishing still requires anonymous cloning off while a release would hide paths — `409`, code `publicignore_clone_exposure` — with the web publish wizard's one-click **Turn off anonymous clone**; once the release is ready, turning cloning back on does not re-expose the paths while the ready release, an active branch's committed trees or a tag's committed tree still hides them.

The check covers both committed trees a branch can advertise, every tag's changeset tree, and ready releases; a path that exists only in an older commit is not retroactively hidden. The boundary evaluates up to 5,000 active branches; its expensive-work caps are 100 unique policy-bearing trees, 500 tree objects under policy work, and 200,000 files (policy-free root probes are liveness-bounded separately), and exceeding one fails closed.

An allowlisted federator reads the same public surface through the origin's `/public` routes; it is public-plaintext only and cannot reach share links, private/E2E content or ACL-restricted files.

End-to-end (E2E) tier (opt-in, per repository)

On an E2E repository the repository key is **never wrapped with a hub-held KEK**. Instead:

- Each member has an **X25519 keypair**, generated and enrolled per device; the private key never leaves the member's machine. (Deriving the same keypair from a password via Argon2id — so a member could recover access on a new device without a per-device re-enrollment step — is on the roadmap, not implemented today: every device holds its own independently-generated keypair.)
- The repository key is **wrapped to each member's public key**. Granting a new member = an existing member's client wraps the repository key to the newcomer's public key and uploads the (un-openable) wrapped blob. The hub distributes wrapped blobs but never sees the plaintext repository key.

This is a **content-only** guarantee, and we say so up front: file and folder *contents* become unreadable to the server, but **filenames, folder structure, and file sizes stay server-visible** so that search, ACLs, browsing, and quota accounting keep working. Encrypting metadata too would break all of those and is a separate, much larger effort — nearly every B2B "E2E" product draws this same line.

To be precise about which metadata: what the server sees is the *entry-level* metadata it keeps in its database — the name, its place in the folder tree, and the file's total size. It does **not** see the file's internal structure.

The per-file block **manifest** (the ordered list of content-chunk hashes and their sizes) is itself encrypted alongside the content — the engine runs it through the same block-encryption as the chunks — so on the E2E tier the server holds the total size but the chunk-level layout is opaque to it. (This is why an E2E share is fetched and reassembled entirely in the recipient's browser: the server can't read the manifest to hand out a chunk list, let alone assemble the file.)

One more thing the server does see: **block names**. Each stored block is named by the BLAKE3 hash of its *plaintext* chunk — that is what lets identical content deduplicate. So the server can tell when two stored blocks are the same, and someone who already holds a copy of a file could chunk it the way the engine does and check whether those blocks exist in a repository. What the server does **not** keep for E2E content is any whole-file hash: the SHA-256 that git-LFS uses to address a file is never stored for an E2E repository, so E2E files can't be matched against published lists of file hashes.

Share links on an E2E repository

A public share link has no account and no device keypair, so it can't receive the repository key wrapped the member way. Instead the link carries the repository key wrapped to a **per-share random "link key"** that lives only in the URL `#fragment` — browsers never send a fragment to a server, so the hub distributes the wrapped blob but never sees the link key or the repository key. The recipient's browser unwraps it and decrypts the file locally.

The link carries the **repository master key**, not a per-file key — the block cipher derives every block's key from the repository key, so there is no narrower key to hand out. Two consequences follow:

- **The share's block endpoint is scoped to the shared file.** The hub serves a share's raw ciphertext blocks through one endpoint, restricted to the shared file's own block set: its manifest plus the chunk hashes the creator's browser enumerated at share time (the hub can't read the E2E manifest to derive them itself).

- **That boundary is enforced at the serving layer.** Because the recipient genuinely holds the repository key, the hub's scoping of the link to the shared file's blocks is an access control, not cryptographic isolation. Per-file share keys — which would make that isolation cryptographic — are a planned change. Treat a single-file share as granting access to that file and nothing else: scope links with expiry and revocation, and share narrowly. Sharing is an explicit trust decision.

The trades E2E makes (know these before you turn it on)

- **Server-side previews/proxies are disabled** for E2E content — the server can't decrypt to render them. Image/PDF preview can move to in-browser decryption; server-generated video posters and point-cloud rasters cannot. This is a deliberate, headline trade-off.
- **SSO/SCIM grants authentication, not key access.** A provisioned user can sign in but sees nothing until an existing member's client (or the optional key-granter, below) wraps the repository key to them; when no grant was queued at all (a group-added member, a failed fan-out), the member's own engine asks the hub to open one.
- **Recovery is via an org recovery key, or not at all.** Losing every device holding a wrapped copy means losing the data. The enterprise-viable answer is an **org recovery key**: repository keys are *also* wrapped to a recovery public key whose private half is escrowed via **Shamir secret sharing (2-of-3 by default)** across your admins. This reintroduces a party that *can* decrypt, so "E2E with recovery" is a point on a spectrum — customers who choose no-recovery accept unrecoverable loss, and we describe it that way at the point of choice. Rotating the recovery key does not re-wrap existing repositories: a repository stays sealed to the key in force when it was created, recovery accepts shares from any recovery key the org has created, and the previous shares must be kept or that repository becomes unrecoverable.

Optional always-on key-granter

For orgs that add many members while their existing members are offline, an opt-in **key-granter** service can hold an escrowed recovery share and fulfill pending grants on a tight cadence. It is off by default and, like the recovery key, is a decrypting party you are choosing to introduce for operational convenience — enrolled explicitly per org, never automatically.

Integrity, availability, and network isolation

- **Content integrity.** Every block is BLAKE3-verified on arrival; corrupt data from a bad peer, transport, or disk never reaches the application layer.
- **Erasure coding.** Blocks are protected with adaptive Reed-Solomon 10+4 — any 10 of 14 shards reconstruct the data, so up to 4 slow or offline peers never block a read.
- **Private P2P swarm.** The peer transport requires a pre-shared key and a custom protocol identifier; a random internet host cannot dial your blocks even if it learns a peer's address. Peers that connect directly do learn each other's network addresses, which is inherent to peer-to-peer transfer; the cloud-only mode below removes that exposure.
- **Cloud-only (hub-only) data-plane mode.** An org policy (or engine-local `SWARMFILE_HUB_ONLY`) disables P2P entirely — QUIC never binds, no peer discovery, no gossip, no peer-IP exposure — and all reads fall through to HTTPS against cloud storage. Re-verified fail-closed on every switch between repositories or orgs, not just at startup, so a policy check that can't complete never leaves P2P silently enabled. Toggling it is itself an audited event.
- **Ransomware detection.** The system watches for the signature of encryption malware: a large number of *distinct entries* overwritten (by content hash) in a short window, counting distinct entries rather than raw history rows. Writes are counted per user and machine, with a per-user total across all of that user's machines as a backstop. The response is two-tier: crossing the **alert bar** only records an alert-only incident and notifies org owners (it does *not* block), while crossing the higher **block bar** hard-quarantines the user on every machine. The block is enforced on the write itself — saves, creates/deletes, conflict resolves, changeset commits/checkouts, `git push` applies, branch/MR merges, LFS reflect/uploads and direct uploads are refused, not only lock acquisition — and a repository/agent API key or a human-triggered CI job inherits its owner's block (an actorless scheduled CI job must be stopped or revoked). Reads keep working and the drive stays mounted: Windows accepts ordinary writes locally and holds them (a rename is refused), while macOS/Linux refuse a new write-open; `swarmfile status` reports `WRITES PAUSED`. Per-machine overwrite counts exclude restores, checkouts, reverts and merges the hub applies from content the repository already held, so a legitimate bulk recovery doesn't trip the per-machine bars. Both bars are configurable per organization under **Settings → Quarantine → Detection thresholds**, with sensible defaults. Alert-only incidents auto-expire after 7 days; blocks require an owner or admin to clear, and a release records whether it was a false positive (the Quarantine view publishes the block/alert false-positive rates with their sample sizes; manual quarantines are excluded). An active incident also offers an owner-only org-wide rollback to 60 seconds before the recorded burst point as a new commit. See Operations for the release, review-request and rollback workflow.
- **Storage blast-radius containment.** Every **paid** org can get **dedicated storage**: its own bucket, reached through a credential scoped to that bucket alone, so a credential minted for your org cannot reach another org's bucket. New paid orgs get it by default; an org that started on shared storage can switch at any time, including after it has uploaded files — the existing objects are copied into the dedicated bucket in the background while reads fall back to the shared prefix. See Dedicated Storage Isolation.
- **Data residency (jurisdiction pinning).** An org's metadata and dedicated block storage can be pinned to a real, infrastructure-enforced jurisdiction — EU or US — chosen once at signup and permanent thereafter; free and self-serve on every paid plan, not an Enterprise upsell. An individual repository can also pin its **own** region at creation, independent of the org's: it gets its own metadata namespace and its own region bucket, so its content is never served from the org's storage. FedRAMP / FedRAMP-High storage regions are available on Enterprise, provisioned through sales (Swarmfile is not FedRAMP-authorized). This is a platform-level restriction, not an application-level convention: jurisdiction-pinned storage lives in a namespace invisible to requests that don't carry the matching jurisdiction. See Data Residency.

For the strictest buyers: self-hosted control plane

Where "the vendor's infrastructure touches our data at all" is disqualifying (some government, defense, and critical-infrastructure work), the E2E tier may still not be enough because metadata is server-visible.

For those cases a **self-hosted control plane** is the Enterprise option: the same code running on infrastructure you control, on a self-hostable runtime compatible with the platform the hosted service is built on. It is available on Enterprise today; a fully air-gapped deployment is on the roadmap — contact us to scope what it would involve for your environment. It is the top of the spectrum; the E2E tier is the middle; managed is the default.

Certifications and controls on the roadmap

- **SOC 2** is on our roadmap (Type I first, Type II thereafter); we have not yet begun a formal audit. In the meantime we answer security questionnaires directly and walk teams through this architecture on request.

- **SIEM forwarding** — a managed Splunk/Datadog/S3 stream of audit, access, and quarantine events — is planned. Today, a Pro-plan owner can export the activity feed to CSV (up to 100k rows per export), and outbound webhooks can push events to an endpoint you control when they are recorded — typically within seconds; the hourly tick is a backstop.
- **Legal hold & retention immutability** — a per-entry legal-hold flag overriding retention/trash purge — is planned.
- **DLP & egress controls** — watermarking, share-link domain allowlists, and export-policy gating — are planned.

If any of these is a hard requirement for you today, contact us to talk through sequencing. For the permission model that *is* live today, see Permissions; for the E2E setup walkthrough, see End-to-End Encryption Setup.

Trust & Compliance

A map for security reviews and procurement. Everything here links to the details; we claim no certification we don't hold.

Compliance posture today

Ask	Status
SOC 2	On the roadmap (Type I first, then Type II); **no formal audit has begun**. We answer security questionnaires directly and walk teams through the architecture on request.
FedRAMP	FedRAMP / FedRAMP-High storage regions are available on Enterprise, provisioned through sales rather than the self-serve picker. Swarmfile is **not** FedRAMP-authorized — talk to us before planning around it.
DPA	There's no published, pre-signed DPA today. Start the conversation via contact us and we'll tell you exactly what we can sign and work through your paper.
Subprocessor list	Below, plus the service-provider disclosure in the Privacy Policy.
Security questionnaire / security package	Answered directly, alongside the Security Architecture whitepaper.
EU / US data residency	Available today, free on every paid plan, chosen at org creation (or pinned per repository at creation) — see Data Residency.
Bring-your-own storage	Available on Enterprise: your own S3-compatible bucket becomes the org's block storage — see Bring Your Own Storage.
Self-hosted control plane	Available on Enterprise: the same control-plane code on infrastructure you operate, on a self-hostable runtime compatible with the platform it's built on — see Deployment Topologies. A fully air-gapped deployment is on the roadmap.
SAML SSO	Available on Enterprise via an adapter; OIDC SSO is self-serve on Pro and above — see Identity.

Subprocessors

These are the third parties that process data for the hosted service:

Subprocessor	Role	What it can access
Cloudflare (Workers, R2, Pages, KV/D1)	Hosting for the hub, identity service, dashboard, and default block storage; optional Turnstile bot check at signup/contact; Workers Containers for video previews and hosted CI jobs	Encrypted blocks for encrypted private repositories; unencrypted for Free-plan repositories (public or private) and public repositories by design; git-LFS objects on a managed repository stored through the transfer agent's encrypted blocks (the stock direct paths are refused; older direct-upload objects are converted by an operator-run migration); hosted CI jobs with their secrets and logs; request metadata
Postmark	Notification and account email delivery	Recipient addresses and the notification content (event metadata, never file contents)
Stripe	Billing and payments	Account and payment details; never file content
iroh / N0 relay network	NAT traversal for peer-to-peer connections when a direct path fails	Connection metadata; relayed traffic is end-to-end encrypted between peers
Google	Google Analytics on the public marketing/docs pages only, after the visitor explicitly allows analytics (not loaded before that choice or on public repository pages, Explore, the signed-in dashboard, or app routes); Google Fonts site-wide	Page-view metadata (URLs, referrer, approximate location) from analytics; font requests carry the page URL as referrer. No repository or file content
GitHub	Distribution of the git-LFS agent installer and release artifacts	Download metadata only

With Bring Your Own Storage, block storage moves out of our hosted storage and into your own cloud account; with Dedicated Storage, it moves into a bucket exclusively yours on our infrastructure. Cloud-only mode removes the peer-to-peer path entirely.

Data lifecycle commitments

- **What's stored:** file content (encrypted according to the repository's tier), block metadata, repository history, and the activity/audit events described in Telemetry & Data Collection.
- **Retention:** history, trash, and audit rows default to **90 days** and are configurable 1–3650 days per org. Per-user preferences cover history and trash; the activity/ACL/audit rows follow the org window. Deletion audit rows for repository and org purges are the exception: they are durable and outlive the purge, so a deletion stays auditable after the data is gone. See Operations.
- **Deletion:** canceling or an unresolved payment failure starts a **28-day grace period** with access blocked and reminder emails, after which the org's data and the owner's account (if it has no other org) are permanently deleted. The org owner can also delete an organization immediately from the dashboard. See Data Portability & Offboarding.
- **Export:** there is no lock-in format — the mount is a drive, so teams copy files out with normal tools; the activity feed exports to CSV; Branch Mirror continuously exports real files to a bucket you control. There is no indefinite post-cancellation read state, so export before you cancel.

Encryption you can point at

- **At rest:** AES-256-GCM for private repositories on paid plans, with per-repository keys; Free-plan repositories (public or private) and public repositories are plaintext by design. On a managed repository, git-LFS uploads must use the transfer agent and are encrypted at rest — the stock direct paths are refused with `lfs_agent_required`; objects an older managed repository stored before that rule are converted by an operator-run migration. The full tier comparison and exceptions are in Security Architecture.

- **End-to-end (opt-in, Pro+):** the key never reaches our servers; recovery is the customer's responsibility via device transfer and recovery keys.
- **In transit:** TLS to the hub and storage; QUIC with end-to-end encryption between peers.
- **Public release exposure:** a publisher's committed `.swarmfile/publicignore.yml` keeps matched paths and platform-derived data (currently CI-run logs) out of the anonymous release view, enforced as each release is published. The same policy sets the anonymous git history boundary: while any active branch's committed tree carries a policy that hides a path in it, or a ready release hid paths, anonymous clone is refused outright (git history is content-addressed, so there is no filtered clone at the same commit ids). Publishing still requires anonymous cloning off while a release would hide paths (`409 publicignore_clone_exposure`), and the web wizard disables **Continue** and offers one-click **Turn off anonymous clone**; turning cloning back on later does not re-expose those paths while the ready release or policy still hides them. A change takes effect within about a minute. The policy hides nothing from members; ACLs still apply, so a restricted folder can still refuse a member's clone. An invalid or unreadable policy (or a boundary read/walk error) refuses anonymous clone (fail closed), and a release already published keeps the contents it was built with. Share links are not covered. See Security Architecture.

Security contact

- **Vulnerability reports:** the security page directs you to the contact form — put "Security vulnerability" in your message so it reaches the right people. Please include reproduction details and give us reasonable time before public disclosure.
- **Everything else (DPA, questionnaires, incident questions):** contact us and mark the request as a security review.

What is not available today

Procurement checklists often include these; we don't claim them today:

- SOC 2 report, penetration-test summary, or a public trust portal.
- A managed SIEM/streaming export (CSV export and webhooks exist — Operations).
- Legal hold / retention immutability, DLP watermarking, and share-link domain allowlists (roadmap).
- A fully air-gapped deployment (the self-hosted control plane exists on Enterprise; air-gapped operation is roadmap).

If one of these is a hard requirement, tell us during the review and we'll talk sequencing honestly rather than work around it.

Where to start a security review

1. Security Architecture — tiers, threat model, what each party can see.
2. Security — operational controls (encryption, integrity, quarantine, cloud-only mode).
3. Network Requirements — every host, port, and direction.
4. Telemetry & Data Collection — what is measured and what never leaves unencrypted.
5. Data Portability & Offboarding — exit path and deletion timing.

End-to-End Encryption Setup

Private repositories on Starter and above are encrypted by default with a hub-held key (Security covers the model — Free-plan repositories (public or private) and public repositories are stored unencrypted). End-to-end (E2E) encryption is a stronger, opt-in tier: the repository key is wrapped to each member's own device keypair and never reaches Swarmfile's servers in plaintext, so the service cannot read your file contents at all.

Plan: Pro and above. E2E is enforced server-side — creating an E2E repository on a plan without it is refused with a `402` (upgrade required), so pick the tier on a Pro or Enterprise org.

This page is the operational how-to — how to turn it on, set up recovery, and what to know before you do. For the cryptographic model and what E2E does and doesn't protect, read the E2E section of Security first; the short version is that it's **content-only** (filenames, folder structure, and sizes stay server-visible so search, ACLs, browse, and quota keep working).

Before you turn it on

Three things are worth internalizing before you create your first E2E repository, because two of them are irreversible:

- **The tier is immutable per repository.** A repository is either managed-key or E2E, chosen at creation and never changed. You can't convert an existing managed repository to E2E (the hub has already seen its key) or an E2E repository back. To move existing data under E2E, create a new E2E repository and migrate into it.
- **git-LFS is unsupported on E2E repositories.** An LFS request returns a per-object `422` with a clear message: a stock `git-lfs` client can't hold a per-user key and the hub structurally can't read E2E content, so there's no one to encrypt or decrypt for. Use a managed or unencrypted repository for LFS, or keep the asset on the mounted drive.
- **Hosted CI does not run on E2E content; a self-hosted runner does.** The hosted service holds no E2E key, so a hosted job is refused. A self-hosted CI runner can run the repository's jobs from an owner-maintained plaintext workflow sidecar; the repository key is delivered to the granted runner's own device, never to the hub in plaintext.
- **Server-side previews are disabled on E2E content.** Thumbnails, video posters, scrubbable proxies, and point-cloud previews are all generated server-side, which an E2E repository's server can't do because it can't decrypt. E2E files show a generic icon in the dashboard. This is the single biggest day-to-day cost — see File Previews.
- **"E2E with recovery" is not zero-knowledge-absolute.** If you set up an org recovery key (strongly recommended, below), a quorum of your admins *can* reconstruct a repository key. That's a deliberate, enterprise-grade tradeoff — the alternative is that a lost password means permanently lost data. Choose consciously and tell your security reviewer plainly: recovery-enabled E2E means "Swarmfile can't read it, but your own admin quorum can."

We'd suggest piloting on a single non-critical repository first: the tier is immutable per repository, so a hasty rollout is expensive to walk back.

Step 1 — Set up the org recovery key *first*

Do this before creating any E2E repository. When you create an E2E repository, its key is wrapped to your org recovery key *if one already exists at that moment*. A repository created before the recovery key exists has no recovery wrap — losing every member's device would then mean losing that repository's data with no way back. Order matters.

As owner, open the **Recovery Key** tab in the dashboard (owner-only). Creating a recovery key:

- Generates an org recovery keypair and splits the private half with **Shamir secret sharing** into *N* shares with a *t*-of-*N* threshold (the default is 2-of-3). Any `t` shares can reconstruct it; any `t−1` reveal nothing.
- Returns the share strings **exactly once**, on creation. They are never retrievable again. The view is print-friendly on purpose.

Distribute the shares to **different people and different physical locations** — a safe, a vault, an HSM, separate offices. The whole point of the threshold is that no single lost or compromised share exposes anything, and no single person can unilaterally decrypt. Store them the way you'd store the master keys to anything else that matters.

Rotating (creating a new key) or revoking is done from the same tab. Note that rotation applies to repositories created *afterward* — it does not retroactively re-wrap repositories created under a previous recovery key. **Keep the previous shares**: recovery accepts shares from any recovery key the org has created, active or previous, so those old shares are the only way to recover a repository that was bootstrapped before the rotation. The dashboard warns about this at the rotate button.

Step 2 — Create an E2E repository

Create a repository as usual (**Repositories → New**), and choose the **E2E** encryption tier instead of the default managed tier. The creation dialog states the tradeoff at the point of choice — previews disabled, recovery via the org key only, immutable after creation. E2E repositories carry an **E2E** badge in the repository list so the tier is never ambiguous.

Step 3 — Members enroll automatically

There's no manual key ceremony for ordinary members. When a member signs in on a machine, the engine enrolls that device: it registers the device's **public** key with the hub, while the private key is generated on the device and never leaves it. No admin action, no per-device setup screen.

Granting a member access to an E2E repository happens **client-side**: an existing member's client wraps the repository key to the new member's public key.

The consequence worth knowing is timing — a newly added member sees an E2E repository's contents only once *another* member's client has been online to complete that wrap. Add someone while the rest of the team is offline and their access is pending until someone with a key comes online.

A running engine watches for the grant on its own and mounts the drive as soon as it lands — there is nothing to restart — and if that member switches to the repository before the wrap is done, the switch waits for it rather than failing. If no grant was ever queued for them (for example a member added through a group), the engine asks the hub to open one the first time it tries to mount, so that wait heals without any manual step.

The optional granter

An always-on **granter** closes this gap so grants don't wait on a human being online — enroll it yourself from the **Recovery Key** tab's "Granter" section (it needs a recovery key set up first, and reuses the same recovery shares): paste your threshold of shares once, and from then on new members and new devices get their keys automatically, on a five-minute cadence at worst.

If you'd rather not run a granter, or a grant is stuck waiting on one, any member whose own device already has the repository's key can clear the queue by hand: open the repository's **Permissions** tab in the web dashboard — a banner lists every pending device grant (who it's for) with a **Grant access now** button that wraps the key to each waiting device on the spot, and a **Dismiss** button per grant if it was created in error.

Multiple devices for one person

A member who works on more than one machine transfers their key between devices directly — a one-time, ten-minute code links a new device to an already-enrolled one and moves the wrapped repository keys across, without the private key ever touching the hub. The code is bound to your account: a device signed in as a different account can neither link it nor fetch the moved keys.

From the CLI: on the device that already has the repository, run `swarmfile device-transfer start`; on the new device, run `swarmfile device-transfer link <code>` with the code it printed (see the CLI reference for the full details).

The Desktop App's **Settings → Add another device** panel does the same thing without a terminal — generate a code on one device, type it into the other, and both sides poll and finish automatically once linked. Enrollment on the new device is otherwise the same automatic flow as the first.

Recovering a repository key

If a member is locked out and their access can't be re-granted the ordinary way, an owner recovers the repository key from the **Recovery Key** tab's recover flow: gather a quorum of shares from wherever you distributed them (for a repository created before a recovery-key rotation, that is the *previous* key's share set — its threshold may differ from the active key's), enter them together with the repository, and the dashboard reconstructs that repository's plaintext key.

This is a break-glass procedure — it requires the physical cooperation of your threshold of share-holders by design, and it's the reason the shares are stored apart in the first place.

Sharing E2E content externally

A share link on an E2E repository still works for an external reviewer with no account: the key needed to decrypt travels in the link's URL fragment, which browsers never send to the server, so the hub serves only ciphertext and the recipient's browser decrypts locally. Through the link the hub limits raw-block access to the shared file's own blocks, so a recipient fetches only the file you shared — not the rest of the repository.

That limit is hub access control, not cryptographic separation: the link carries the *repository* key, so treat sharing as a real trust decision and use expiry/revocation; revocation and per-email blocks stop block retrieval on the recipient's very next request. An access-count cap set on an E2E share through the API is enforced per distinct verified email — each new verified visitor consumes one access.

(The deeper key-custody detail is in Security Architecture.) Image and PDF previews work this way; video proxies don't (the server can't transcode what it can't decrypt), consistent with the in-app preview limits above.

When a member is removed

Removing someone from the org cuts off their access to E2E content within a few seconds: the hub stops serving them blocks and their wrapped repository keys, revokes their key wraps, and cancels any key grants still pending for them. The revocation reaches every repository in the org, including ones in the trash; a repository that doesn't answer is retried until it does, and **Settings → Access revocations** shows any that are still pending.

Their desktop drive stops serving files, clears its local cache of the repository, and discards the repository key it held. Changes they saved that hadn't finished uploading stay on their computer, still encrypted, and upload if their access is restored.

A computer that's offline at the time does this when it next reaches the hub — the org's offline access window, on by default at 7 days, bounds how long that can take.

What removal **can't** do is recall key material that's already on their device. A member who had access held a repository key, and the hub can't reach into a machine to erase anything copied from it. If a departing member's copy of the key is a real concern, **rotate the repository key** (below) so future writes stop using it.

Rotating an E2E repository key

A repository owner or admin can rotate an E2E repository's key from the Desktop App's **Settings → My Devices** panel — the repository's key-access card — and from the CLI with `swarmfile repository rotate-key` (run it with the repository open; it reads the active repository off the mount). Rotation mints a fresh key generation and re-wraps it to every current device **and the org recovery key**, all client-side; the hub only ever stores the wrapped envelopes and never sees the key.

Know what it does and doesn't protect:

- **New content is protected.** Every write after the rotation is sealed under the new generation.
- **Existing content is not re-encrypted.** Blocks written before the rotation stay under the old generation, so a device that already unwrapped the old key can still decrypt content it already has or can still fetch from the swarm. Treat rotation as "stop using the old key going forward", not "make the old key useless everywhere".
- **Rotation is client-driven.** The hub never holds the key, so one of your devices performs it; if an org recovery key is configured, the new generation is wrapped to it too, so recovery keeps working. Only owners and admins can rotate.
- **Other devices keep reading.** A device that doesn't yet hold the new generation fetches it on its next check and keeps its old-generation key so existing files still open.

Seeing and revoking a repository's devices

Owners and admins can see which devices hold an E2E repository's key. Open the repository's **Permissions** tab in the web dashboard and look under **Devices with access**. Each device is listed with its name, the person it belongs to, its platform, app version and when it was last seen. It also shows whether it still has access and, if not, why: removed from the organization, revoked by an admin, revoked by its owner, or waiting for a new grant.

Revoke cuts one device off from that one repository. The device stops opening the repository's files, and the copy cached on it is cleared when it next connects. It doesn't get access back automatically, and the person's other devices aren't affected.

You can also revoke a device that has lost access but would be given it again when it next asks (removed from the organization, or waiting for a new grant), so a lost device in those states stays locked out. As with any revocation, anything already copied off the device can't be recalled. To cut a device off from every repository at once, its owner can revoke it under **Settings → My Devices**.

What lives where

For a regular member, enrollment and key-granting are **automatic**: they happen in the engine without user action, and there's no Desktop App UI for them (nor anything to configure there). The owner-facing admin — the recovery key, its shares, and the recover flow — lives in the web dashboard only.

A member's own device keys, though, are self-service from the web dashboard, the CLI, and the Desktop App — distinct from the owner-only recovery-key/quorum machinery above, which stays web-only. In the dashboard, **Settings → My Devices** lists every device you've enrolled and lets you revoke a lost one yourself, no admin needed. From the CLI, `swarmfile e2e-key status` shows this device's own enrolled key (id, public key) straight from its local cache, no hub round trip; `swarmfile e2e-key list` audits every device enrolled on your account, and `swarmfile e2e-key revoke <id>` kills a lost one's access the same way the dashboard does.

Revoking a device takes effect on every E2E repository in the org. If the device is online, it's cut off within seconds: its drive stops serving the org's E2E repositories, and it discards the repository keys it held along with its local cache of their files. If it's offline, the same happens as soon as it reconnects. It's also never granted a repository key again, even from a request made on one of your other devices. Revoking a device key doesn't sign that device out, though. Revoking the key can't recall a repository key that was copied off the device before you revoked it.

The Desktop App's **Settings → Add another device** panel walks through the same transfer `swarmfile device-transfer` does below, without touching a terminal.

Headless/CI identities work differently again: `swarmfile api-key create` mints a repository-scoped API key, and for an E2E-tier repository that command *is* a key-management operation — it generates real E2E key material (a device keypair, repository-key wrap) for that key to use, printed once alongside the key itself. `swarmfile api-key revoke` reverses it, tearing down that identity's key material along with the key.

The dashboard's API keys page and the tray's API keys panel mint these too (the CLI remains the scripted path). A path-scoped key cannot introduce new content on an E2E repository, and E2E git with a scoped key is not supported — use an owner credential.

The bottom line for a security review

- Contents are unreadable to Swarmfile; metadata (names, structure, sizes) is not.
- Recovery is your own admin quorum, not the vendor — and only if you set the recovery key up before creating repositories.
- Previews and server-side proxies are off for E2E content, by necessity.
- Set-up is irreversible per repository; pilot before you commit a flagship repository to it.
- Removing a member revokes their key wraps and stops their access, but can't recall a repository key already on their device.
- An E2E repository key can be **rotated** (client-side, owner/admin) so future writes stop using an old key — but that does **not** re-encrypt existing content, so a device holding the old key can still read what was written before the rotation.

If E2E is a hard procurement requirement, raise it early and we'll walk your reviewer through the envelope format and the recovery threat model directly.

Dedicated Storage Isolation

Dedicated storage is the default for every new paid organization, and every paid organization still on shared storage can switch to it: your org gets its own bucket, reached through a credential scoped to that one bucket. The credential your org's traffic uses cannot touch any other org's data, even if that credential is compromised or misused. Free-plan orgs, and paid orgs that explicitly opted out at creation time, use shared storage: one shared bucket in which every org's objects are scoped under its own prefix.

Switching an organization that already has files is supported. The bucket is provisioned first, then every existing object is copied into it in the background and verified; until each object is copied, reads fall back to the organization's previous shared storage, so there is no read outage and no data-loss window.

This page is the operational how-to. For the underlying model — what "blast radius" means here and why the boundary is the org rather than the repository — see Security Architecture.

What this does and doesn't change

- **Bucket isolation, not encryption.** Dedicated storage is about *where the bytes live*, not what protects them once there — it gives your org a storage credential scoped to its own bucket alone, so it can't reach another org's data. Encryption at rest is a separate question with its own tiers: private repositories on paid plans are encrypted; Free-plan repositories (public or private) and public repositories are stored unencrypted (see Security).
- **Still our infrastructure, not yours.** This is **not** bring-your-own-storage — your data stays in a bucket we create and operate on our own cloud infrastructure, just a bucket that's exclusively yours instead of shared. If your compliance requirement is specifically "our data must live in infrastructure we control," this isn't the right tier: for that, Bring-your-own-storage (Enterprise) makes a bucket on *your* own account the org's primary block storage, and a self-hosted Enterprise control plane (see Deployment Topologies) puts the control plane and storage both on infrastructure you control. Talk to us about which fits.
- **The org boundary, not the repository boundary.** One dedicated bucket covers every repository in your org. Repositories within it are separated by a repository-scoped prefix inside the bucket.

Requesting it

New paid orgs select this automatically — there's nothing to request. Org creation records the storage choice but does not provision a bucket. The dedicated bucket is created only when the user deliberately creates the org's first repository. API callers can also explicitly opt out at creation time with `POST /orgs` and `blockIsolation: "shared"`.

If your org was created with the shared opt-out — or upgraded to a paid plan after it already had files — an owner can turn dedicated storage on from **Settings → Storage**. It is a one-way switch from that page; turning it back into a shared-storage org is an operator action (see below).

Starter plan or above only. Free-plan orgs always use shared storage and aren't offered the switch at all; upgrading to Starter or Pro is the path onto dedicated storage, and the dedicated bucket is then created with the org's first repository (or by the switch itself). The same rule is enforced server-side, not just hidden in the UI.

Existing files are copied in the background. Switching is not rejected when the organization already has files. The dedicated bucket is created and the organization is flipped onto it first; a background job then copies every object under the organization's previous shared prefix into the dedicated bucket and verifies each copy (size plus the object's ETag).

Reads keep working throughout: an object that has not been copied yet is served from the previous shared storage. New writes go to the dedicated bucket immediately.

The previous shared copy is retained for a safety window after verification; removing it is a separate, explicit operator action once that window elapses.

The same page also carries your bucket's public-access switches when your org is on Bring-your-own-storage — anonymous streaming and gated release downloads served from a customer-supplied bucket. Shared and Swarmfile-managed dedicated buckets always stream and serve downloads, so those switches only appear in BYOS mode.

What happens during provisioning

Provisioning isn't instant — creating the bucket and minting a credential scoped to it both involve real calls to our storage platform, and a freshly minted credential needs a few seconds to become valid before we'll rely on it. This begins when the first repository is explicitly created, or on-demand when an existing eligible org turns the switch on from Settings → Storage.

While provisioning runs, existing repositories keep reading and writing from the organization's previous shared storage — the not-ready mode is not pushed to them. Creating a *new* repository is briefly refused with a retryable `storage_provisioning` error until provisioning completes, so a new repository is never bootstrapped against a half-ready bucket; retrying the create once provisioning finishes is the normal path.

If provisioning hits a transient failure (an upstream API hiccup, for instance), it retries automatically with backoff, so nothing needs to be re-requested by hand.

Once the org is on its dedicated bucket, the background copy is the only remaining work. It runs on the organization's own background alarm, in small bounded pages, and is safe to pause and resume. The Storage page shows its progress; an operator can also inspect it with the `storage-migration` ops command. A copy that hits an unexpected failure stops and reports `failed` (reads still work through the shared fallback) rather than silently claiming completion.

Known limitations

- **The copy is asynchronous.** An organization with existing files reads from two places until the copy completes: copied objects from the dedicated bucket, not-yet-copied objects from the previous shared storage. That is transparent at the API, but it means the previous shared copy must not be deleted until the copy is verified and its retention window has elapsed. Cleanup is an operator action, never automatic.
- **One-way from the Storage page.** There's no self-serve path to move an org that's already on dedicated storage back to the shared tier. An operator can roll it back safely: a reverse copy moves post-switch objects back into the shared prefix and verifies it before the organization is re-pointed at shared.

- **No self-service credential rotation.** If you have a specific reason to believe your org's storage credential needs rotating, contact us directly — there is no self-service flow.
- **Owner-only.** Admins and members don't see the Storage tab; only the org owner can turn it on — and a Free-plan org's owner can't turn it on at all (see above).

For API consumers

`GET /orgs/:orgId/block-bucket` reports `hasContent` — whether the org already has files. It is informational (it no longer gates the switch, and an org with files can enable dedicated storage) and is only meaningful for a `shared` org; the hub skips the cross-repository lookup and reports `false` for `dedicated`, `byos`, `provisioning`, and `failed` orgs.

`mode` is the field to branch on everywhere. The response also carries `migration` for a dedicated org that flipped with existing files: its `status` (`copying`, `verifying`, `paused`, `complete`, `failed`, or a rollback phase), cumulative counters, and the verification timestamp.

A `complete` migration with a `verifiedAt` older than the retention window is eligible for operator-driven shared-copy cleanup.

Bring Your Own Storage

Bring-your-own-storage (BYOS) makes a customer-supplied S3-compatible bucket the **primary** block storage for an organization. Instead of your data's blocks living in a bucket on our storage platform, they live in a bucket on *your* account, on *your* provider (AWS S3, Wasabi, Backblaze B2, MinIO, and other S3-compatible targets) — with our hub still running the control plane, metadata store, and coordination as usual.

BYOS is an **Enterprise-only** option (the `byos_storage` entitlement).

How it's different from the alternatives

- **Not Dedicated Storage.** Dedicated Storage gives your org its own bucket *inside our own infrastructure* — physically isolated, but still a bucket we create and operate on our cloud account. BYOS moves the bucket out to infrastructure *you* own and control.
- **Not a branch mirror.** A branch mirror is a read-only export *copy* of one branch into your bucket, alongside the authoritative data that still lives on our platform. BYOS is the opposite relationship: your bucket *is* where the live blocks are read from and written to — there's no second copy on our side.
- **Not full self-hosting.** BYOS relocates only the block storage. The hub, metadata store, and coordination still run on our infrastructure. If your requirement is that *everything*, control plane included, runs on infrastructure you operate, that's the self-hosted Enterprise control plane (talk to us; fully air-gapped operation is on the roadmap), not BYOS.

Think of it as a spectrum of "where do the bytes live": shared bucket (Free-plan and shared-storage orgs) → your dedicated bucket on our infra (Dedicated Storage, the default for new paid orgs) → **your bucket on your infra (BYOS)** → your entire deployment on your infra (self-hosted).

Requesting and provisioning it

BYOS is provisioned by the **org owner** through Enterprise onboarding. Because it makes an external bucket the org's primary storage, it's a deliberate, owner-gated action — not a self-serve toggle a member could flip.

You supply a bucket that **already exists** plus a credential that can read and write it; Swarmfile validates the credential with a real write-read-delete round-trip against your bucket before switching anything over, so a bad endpoint, region, bucket, or key is caught immediately rather than surfacing on the org's first real read or write.

Configuration fields

Field	Notes
Endpoint URL	Your provider's S3-compatible endpoint.
Region	The bucket's region.
Bucket name	The target bucket. It must already exist.
Key prefix	Optional sub-path inside the bucket to confine Swarmfile's objects to (e.g. `swarmfile/`).
Access key ID	The S3 access key ID for a credential with read/write access to the bucket.
Secret access key	Stored **encrypted at rest** (AES-256) under a dedicated key, and never returned.

The endpoint must be an `https://` URL on a real remote host. A cleartext `http://` endpoint would put your secret key on the wire, so it's refused unless your deployment operator has explicitly opted in for a trusted on-prem target; loopback, link-local, and cloud-metadata addresses are always refused. A self-hosted operator can additionally pin which hosts are allowed.

Content is encrypted at rest exactly as it is on our own storage for **private, managed** repositories (see Security) — the objects that land in your bucket are the same encrypted blocks, just hosted by you. **Public repositories are unencrypted by design** (they have to be, to be served anonymously), so a public repository's blocks land in your bucket as plaintext.

Public repositories: streaming and downloads from your bucket

If your org publishes a public repository, visitors can play its video and audio and view large images straight away, without downloading the whole file first. For a BYOS org that streaming is served **from your bucket**, so the requests and the data transferred out are billed to **your** storage provider account, not ours.

Two independent owner switches control this, both on **Settings, Storage**, and both take effect on the next request:

- **Anonymous streaming** (**Allow anonymous streaming from my bucket**). Because anyone on the internet can trigger those reads, this is **off by default**.
  - **While it's off:** public visitors can't stream media from your bucket. On the public repository page the audio, video, or image falls back to the proof-of-work gated download (no account needed). Thumbnails, small previews, and readmes are not affected, and neither is anything for signed-in members of your org.
  - **While it's on:** anyone who can see one of your public repositories can stream its media directly from your bucket. Expect your provider to bill for the requests and egress. Swarmfile applies per-IP rate limits and byte budgets to public streaming, but it cannot cap what your provider ultimately charges you.
- **Gated release downloads** (**Allow anonymous release downloads from my bucket**). Release files and a release's combined archive, open to everyone behind a solved proof-of-work challenge and rate limits (no account needed). For a **new** BYOS org this is **off by default**, and gated release downloads are refused until you turn it on — visitors see the same "not found" a private release shows. An org with existing public releases defaults to on so those releases keep working.

Shared and Swarmfile-managed dedicated buckets stream and serve downloads by default; the switches only appear in BYOS mode.

Known limitations

- **New/empty orgs only.** BYOS is set at provisioning time, before the org has uploaded any content. There is **no live migration** of an existing org's blocks into a BYOS bucket in this version — switching an org that already holds data is rejected. Decide on BYOS before adding content, not after.

- **No fallback if your bucket is unavailable.** This is the real operational tradeoff to understand. Your bucket is the *only* home for the blocks — there's no shadow copy on our side to fall back to (a silent fallback would defeat the entire point of BYOS). If your bucket has an outage, or its credential is rotated/revoked out from under Swarmfile, reads and writes for the org **fail synchronously** until you restore access. Storage availability and credential lifecycle become your responsibility.
- **Credential rotation is re-provisioning.** When you rotate the access key, re-run provisioning with the new credential for the *same* bucket. (Re-pointing a non-empty org at a *different* bucket is refused for the same reason a live migration is — it would orphan the existing content.)
- **Set the incomplete-multipart lifecycle rule on your bucket.** On a plaintext-tier repository, a large git-LFS push (≥ 64 MiB) can be one S3-compatible multipart upload that goes **straight to your bucket**; a client killed mid-push can't abort its own upload, and your provider bills the uploaded parts until they're aborted or expired. (On a managed repository, multipart is refused and the transfer agent uploads encrypted blocks instead, so only the block objects land in your bucket.) Add an `AbortIncompleteMultipartUpload` rule — 7 days is the value we use. Swarmfile applies this automatically to buckets it creates; a BYOS bucket is yours to configure.
- **Data residency is yours to enforce.** Swarmfile can't verify the physical region of an arbitrary customer endpoint, so BYOS is refused for an org with a pinned jurisdiction — and for an org that has any repository pinned to its own region. If you have a residency requirement, choosing a bucket in the right region — and keeping it there — is on you.
- **Enterprise-only, owner-gated.** BYOS requires the Enterprise plan, and only the org owner can provision it.

Where to go next

- Dedicated Storage Isolation — physical isolation inside our own infrastructure.
- Branch Mirror — a read-only export copy into a bucket you control.
- Deployment Topologies — the fully self-hosted Enterprise option.

Data Residency

Data residency lets you pin an organization's metadata and block storage to a real, infrastructure-enforced jurisdiction — **EU** or **US** — chosen once at signup.

It's free and self-serve on every paid plan, Starter through Enterprise (Free-plan orgs use shared storage and can't pin a region); there's no price gate and no sales conversation needed to turn it on.

An individual repository can additionally pin its **own** region at creation, independent of the org's — see Per-repository residency.

FedRAMP and FedRAMP-High storage regions are available on Enterprise, provisioned by us through sales rather than the self-serve picker. Swarmfile itself is not FedRAMP-authorized — the regions are available, the authorization is not — so if you have that requirement, talk to us before you plan around it.

This page is the operational how-to. For the underlying mechanism — how jurisdiction pinning actually works at the infrastructure level — see Security Architecture.

What this does and doesn't change

- **Platform-enforced, not a convention.** This isn't an application-level rule we promise to follow — it's a real, platform-level restriction on where your organization's metadata and block storage physically live. Jurisdiction-pinned storage exists in a namespace that's invisible to any request that doesn't carry the matching jurisdiction; there is no shared code path that could accidentally route around it.
- **Covers our system of record, not your own peer-to-peer transfers.** Your own team's machines transferring files to each other over the peer-to-peer engine are already authorized devices on your own network — that's not something jurisdiction pinning governs, the same way a cloud region guarantee doesn't follow your own laptop once you've downloaded a file. If you have your own downstream residency commitment to your own clients, you can additionally turn on cloud-only mode (Settings → Network) to disable peer-to-peer entirely for that org.
- **Global by default.** If you don't choose a jurisdiction at signup, your org runs on our standard global infrastructure — this is a purely additive choice, not a change to what already exists.
- **Immutable, on purpose.** Jurisdiction is fixed at creation by design. Both your metadata store and your block storage have their jurisdiction fixed at creation by the platform itself; moving one means migrating to entirely new infrastructure, not flipping a setting. Choose carefully at signup. The same applies to a per-repository pin (below).

Choosing it

Pick **Data residency** on the signup form (or the organization-creation screen if you're joining as an existing user with no organization yet) — the choice is right next to the organization name field. Leaving it on **Global (default)** is the same as not having this feature at all; picking **European Union** or **United States** pins the org immediately upon creation, and a confirm dialog gates the choice specifically because it can't be undone.

FedRAMP and FedRAMP-High aren't in the self-serve picker, but they are available on Enterprise: we provision them through sales. If you have a FedRAMP requirement, talk to us directly.

Per-repository residency

A paid org can pin an **individual repository** to its own EU/US region when the repository is created — independent of the org's own choice, so a repository on a Global org can be pinned, and a repository on an EU org can be pinned to US. A pinned repository gets its own metadata namespace and its own region bucket, and is never served from the org's storage.

- **Chosen at creation, immutable.** Like the org-wide choice, a repository's region is fixed when the repository is created; there is no way to move or un-pin it later. The create form states this before you commit.
- **Entitlement.** Per-repository EU/US pinning is available on every paid plan; FedRAMP regions are Enterprise, provisioned through sales (not the self-serve picker), and Swarmfile itself is not FedRAMP-authorized.
- **Provisioning.** The region's bucket is provisioned on first use. Until it's ready the repository's reads and writes fail closed with a `storage_provisioning` refusal — it never silently falls back to the org's bucket — and the creator is notified once it's ready.
- **Rules out customer-managed storage.** A pinned repository can't use a branch mirror, and an org with any pinned repository can't switch to BYOS primary storage: Swarmfile can't verify the region of a customer-supplied endpoint.
- **Web, desktop app and CLI.** `swarmfile repository create --residency eu` pins at creation, and `swarmfile repository show` reports the repository's region and storage status (see the CLI reference).

Known limitations today

- **No self-serve upgrade path.** If your org is already running on global infrastructure, there's no way to move it to a pinned jurisdiction after the fact. This is the same "provisioning is a one-time, creation-time decision" constraint as Dedicated Storage, for the same underlying reason — neither your metadata store nor your block storage can be relocated once created.
- **FedRAMP is Enterprise, not self-serve.** There's no self-serve picker for it — we provision FedRAMP/FedRAMP-High regions through sales — and Swarmfile is not FedRAMP-authorized; ask us before planning around it.

Billing & Plans

How billing works

Signing up for **Starter, Pro or Studio** runs through Stripe Checkout. Once you're on a plan, you can **switch between the paid plans right from the Billing tab** — "Upgrade to Pro", "Upgrade to Studio", or "Switch to…" — without leaving the app. The switch retargets your existing subscription in place (it never creates a second one), takes effect immediately, keeps any trial in progress, and confirms the projected cost first.

Canceling, updating a payment method, and viewing invoice history still happen in a self-serve Stripe Customer Portal, available to the org owner (which also offers the same plan switch as a fallback). The Billing tab shows a **"What happens if you cancel?"** summary next to the portal link — 24 hours, then the 28-day grace period with access blocked, then permanent deletion — so the consequences are visible before the cancellation button, not only after. See Data Portability & Offboarding for the full detail.

Free is different: it's a no-card tier you choose at signup, not something you switch into or out of later. There's no Stripe subscription behind it at all — nothing to charge, cancel, or retarget. Upgrading from Free runs the same org through Checkout for the first time, the same as any Starter/Pro signup; there's no separate downgrade path back into Free afterward.

One thing neither the app nor the portal lets you edit **once you're subscribed**: seat *quantity*. (At sign-up you can pre-pick a starting seat count — that's the number the first invoice uses — but from then on it's live.) Seat count is derived from org membership, not a number you type — **you change your seat count by adding or removing people in the Team tab**, and your billed seats follow automatically, reconciled daily. Editing a seat count by hand would just get overwritten by the next reconcile sweep; the Billing tab links you to Team instead.

Starter, Pro and Studio carry the same 7-day free trial. Your card is collected upfront at signup, and the first charge happens on day 7. **During the trial, seats are capped at 2 and add-ons aren't available** — growing past 2 people, or adding extra storage/collaborator capacity, means ending the trial and paying first (you can do that any time, see below). Trials are also one-time per org: once a subscription has gone active, canceling and resubscribing doesn't grant a new trial period.

The trial is optional, and you can leave it early:

- **Skip it.** At sign-up, tick **Skip the 7-day trial and start paying today**, or pick more than 2 seats (a trial can't cover them, so that starts paid automatically). From the Billing tab, each paid plan offers **Subscribe now, skip the trial** next to the trial button. Either way you get a paid subscription straight away, with no trial and no 2-seat cap.
- **End it early.** While trialing, the Billing tab's **End trial and subscribe now** charges your card on file for about the first month's seats right away (the confirmation shows the amount) and lifts the 2-seat cap immediately; add-on packs unlock at the same moment. If the card is declined, nothing changes: you stay on the trial, and you can update the card in the Customer Portal and try again. If you canceled during the trial, resume the subscription in the Customer Portal first; the Billing tab points you there instead of offering to end a trial that's already set to stop. The Team tab's seat-limit notice and the desktop app's trial notice both link there, and the Billing history records the change as **Trial ended early**.

Plans

Plan	Price	Storage	Seats	External collaborators (private)	Headless seats
Free (`community` in the API)	Free, no card	5 GiB total	1 seat only	1 included (public collaborators free)	1
Starter	$4/seat/mo	100 GiB/seat	No plan ceiling; self-serve checkout up to 5,000 seats (contact us above that)	1 per seat (private repositories); public repositories free	2 per seat (min 5)
Pro	$25/seat/mo	500 GiB/seat	No plan ceiling; self-serve checkout up to 5,000 seats (contact us above that)	5 per seat (private repositories); public repositories free	5 per seat (min 10)
Studio	$50/seat/mo	1 TiB (1,024 GiB)/seat	No plan ceiling; self-serve checkout up to 5,000 seats (contact us above that)	10 per seat (private repositories); public repositories free	10 per seat (min 20)

Billing cadence

Paid plans bill **monthly or annually**. The annual discount applies only to what is **prepaid**: seats and committed packs charge ten months for twelve — **two months free**: Starter $40/seat/year, Pro $250/seat/year, Studio $500/seat/year, with annual pack Prices at ten months. Metered usage — storage above the allowance, external collaborators, and hosted-CI compute — stays on a **monthly** line on every plan and is charged as it is used; it is never accrued for a year. Stripe accepts the mixed annual-seat / monthly-usage intervals in one subscription. The monthly spend cap therefore applies to every plan the same way. Pick the cadence at signup; a plan switch in the Billing tab **preserves** the current seat cadence, and changing cadence on an existing subscription goes through the Customer Portal or support (it re-anchors the billing period).

The seat figures above describe an active, paid subscription — the 7-day trial itself is capped at 2 seats regardless of plan (see above). The Free plan has no trial (there's nothing to trial) and no add-ons; its storage cap is a single all-in allowance of 5 GiB from day one, rather than the per-seat allowance the paid plans use.

Each account can have **one Free organization**. Creating a second Free org is refused with a `409` (`code: free_org_limit_reached`); choose a paid plan for the new org instead, or keep working in the Free org you have. The limit counts orgs you created that are on the Free plan right now, so upgrading your Free org to a paid plan frees the slot, and a deleted Free org keeps counting until it is fully removed. Paid orgs are unaffected.

The Free plan can create **public** repositories and **private** repositories, and every repository it creates is stored **unencrypted** at rest — managed and end-to-end encryption require Starter or above, and end-to-end requires Pro/Studio or above. A private Free repository is therefore private by access control, not encrypted: the hub can read its content, and its blocks are never peer-served.

The same unencrypted rule applies to any **public** repository on a paid plan, because public repositories must be unencrypted so they can be served anonymously.

Encryption is the thing the Free plan restricts by *kind*, not just by size: a Free-plan org that wants it has to upgrade first, it can't simply run out of room the way storage or collaborators can. Encryption tier is also fixed at repository creation — upgrading doesn't retroactively encrypt a repository created while on Free; only repositories created with a paid encryption tier get managed encryption. See Security Architecture for the full detail.

Each plan also caps how many **live repositories** an organization can have: 10 on Free, 100 on Starter, 250 on Pro, 1,000 on Studio (Enterprise is contract-negotiated). Deleting a repository frees its slot; the cap bounds per-org resources and never bills anything.

Storage and bandwidth

Everything stored at rest counts toward your storage allowance on the same basis — repository files, and the objects held for a repository used as a git-LFS server (billed on distinct content, so deduplication applies). Storage starts with the first repository: an org with no repositories has no billed storage and no dedicated bucket yet.

Bandwidth is unlimited and free on every plan, for both cloud and P2P transfer. Every plan's included bandwidth allowance is unlimited: egress is never billed and no plan caps it, and egress bytes are still counted per org and shown on the Usage tab.

Anonymous public streaming is separate: byte-range reads of public pages and release media draw on a per-plan org pool that refills continuously, shown to members on the dashboard's **Public streaming** card (owners also see it under **Settings → Usage**) and covered by the streaming allowance; a platform-wide anonymous-read backstop can additionally refuse reads at extreme volume (see Telemetry & Data Collection).

Unlike storage and collaborators, egress has no billed meter at all, so "unlimited" is a policy choice on top of real counters, not an absence of counting.

Self-hosted seed/NAS nodes require Pro/Studio or above; see Self-Hosted Seed Nodes for setup.

Enterprise plans

Need more than Pro's or Studio's limits — unbounded external collaborators and headless seats, a custom seat arrangement, or different billing terms? Enterprise plans are available; contact us.

Verified-OSS grants

An open-source repository can apply for a **Verified-OSS grant**, which puts its org on the **Pro** plan for free for a fixed term (a year by default), capped at **10 seats**. Pro's included allowances and hard ceilings apply as for any Pro org.

An org owner or admin applies from the repository's settings (**Public page → Verified OSS grant → Apply for a Verified OSS grant**), optionally with the repository's SPDX license id and a short motivation, and a person on our side reviews each application.

To be eligible, the repository must be public, its latest release must have been published between 7 days and a year ago, and the org must be at least 30 days old. The checks run when you apply; if the repository doesn't qualify yet, the card lists the checks it failed.

The Billing tab shows the grant and its expiry, and the repository's public page carries a Verified OSS badge. A grant is a plan, not a payment method: hosted CI still needs a card on file, and the grant doesn't include Enterprise-only features. If an application is denied or withdrawn you can reapply after 14 days.

Overage rates

Storage that exceeds your plan's included allowance is billed at $5/100 GiB on Starter and $4/100 GiB on Pro and Studio. External collaborators beyond your plan's included allowance are billed at $1/month per collaborator on **paid plans**, up to the 3x hard ceiling described below — **private-repository grants only**. Free includes one private-repository guest and has no overage.

Collaborators on public repositories are free and consume nothing: neither the included allowance, the overage meter, nor the ceiling.

The Free plan bills no storage or collaborator overage — there's nothing to charge a card that doesn't exist — so its storage allowance is a hard wall and its one private-repository guest is a hard slot (see Enforcement below).

Hosted-CI compute is the separate metered dimension described below. **Headless seats** never bill overage — they're a hard cap, not a metered dimension: Free 1, Starter 2 per seat (minimum 5), Pro 5 per seat (minimum 10) and Studio 10 per seat (minimum 20), and additional seats are **$2/month each** on any paid plan (one repository-scoped key per seat; see Enforcement below).

Prefer committed capacity to overage? **Storage packs** pre-buy it at a discount: **$3/100 GiB on Starter, Pro and Studio** per month — below the matching overage rate. A pack adds 100 GiB to your included allowance for as long as it's on the subscription, and because the allowance is raised first, pack capacity is never also billed as overage (and never counts against the spend cap below).

Manage them in the **Add-ons** section of the Billing tab; a plan caps pack counts per kind — 100 storage or collaborator packs and 1,000 headless seats on Starter; on Pro and Studio, 10,000 storage or headless-seat packs or 1,000 collaborator packs — because storage and collaborator packs also lift the applicable 3x hard ceiling.

Collaborator packs work the same way: $0.80/month per extra private-repository external collaborator (public-repository guests never need one), versus $1 in overage. Changes are also rate-limited (about ten per organization per day), and an increase must clear its prorated charge immediately rather than waiting for the next invoice.

Hosted CI compute is a third metered dimension, separate from the allowances above: Hosted CI jobs bill per second of container time at our underlying provider's at-cost rate × 1.2, rounded up to the next whole second (a started job bills at least ten seconds), with no included allowance on any plan — every cent counts against the spend cap from the first job.

It is deliberately not plan-gated: every plan can enable it per repository. A Free-plan org has no card by default, so it attaches one for compute only (a zero-price subscription, not a plan upgrade) before its first job, and then runs under a Free-plan compute ceiling that owners can't change: it scales with account standing (the org's age, and whether its creator has a verified email, a verified card or a seat in a paid org), from about $2.50 up to $15 of compute a month.

Hosted agent sessions (`hosted-agents`) meter the same way — session container time at the same rate, billed per second (a started session bills at least one second), with a 60-second minimum session budget — and appear as a distinct **agent compute** line on the invoice. A per-session or per-agent budget narrows admission, and every cent counts against the org spend cap.

Enforcement

Quota enforcement is always on. There's no setting to disable it and no feature flag gating it off — it's a structural part of the product, not an opt-in.

Plan storage counts the size of your files. Each repository also has a separate, unbilled limit on its file information and history (names, versions, and so on), which only very large repositories approach — see Repository storage limit.

Storage, seats, and external collaborators are enforced differently, on purpose:

- **Seats** have no plan ceiling; self-serve checkout supports up to 5,000 seats (contact us above that). There is no minimum on any paid plan once you're subscribed, beyond the universal 1-seat floor every org has (you always have at least an owner). Add or remove people in the Team tab any time; billed seats simply follow. The one exception is the 7-day trial itself, which is capped at 2 seats until you commit to a plan (see Plans above). The Team tab reflects whichever cap applies — it replaces the invite form with the plan's limit and an upgrade link at the Free plan's single seat, and a **Subscribe now** link (to end the trial) at the trial's two, so the refusal is visible before anyone types an address.
- **Storage and private-repository external collaborators** both bill metered overage before hard-blocking, rather than blocking right at the included amount: storage hard-blocks at 3x the included allowance, and private-repository external collaborators hard-block at 3x the included per-seat allowance. (Public-repository collaborators sit outside this meter entirely — free, no slot, no ceiling. Free includes exactly one private-repository guest.)
- **Headless seats** are a hard cap, not a soft one: minting is refused with a `402` (`code: keys_cap`) the moment the org is at its allowance — Free 1, Starter 2 per seat with a floor of 5, Pro 5 per seat with a floor of 10, Studio 10 per seat with a floor of 20 — and additional seats are **$2/month each** (one repository-scoped key per seat). A seat is a machine identity, not just a secret: each gets its own 60-second request bucket (300/500/750 per minute by tier) and adds that allowance to the shared org ceiling (5,000/50,000/100,000 per minute), so the cap bounds how many rate buckets a fleet can buy. Per-key spend budgets and the org spend cap stack on top; the org window counts only the current seat allowance, so retained over-cap keys keep working but do not enlarge it. User workstations and personal access tokens are free; one key per machine is the recommended shape for revocation radius. Revoking a key frees a slot immediately, and a downgrade never revokes keys you already have. A key is walled off to exactly one repository, and deleting a repository revokes its keys with it. Owners see the standard soft-cap warning banner in Billing once the org reaches 80% of its seat allowance, before minting ever refuses.
- **Service-minted agent tokens** (the `agent_service_tokens` entitlement) are a separate **Starter+** hosted capability: a service user's PAT mints branch-scoped keys inside an approved `agent/` delegation, at most one hour each. They are not a metered dimension and are not the headless-seat allowance; they are revoked with the token, the delegation or the PAT. See Service-minted agent tokens.
- **The Free plan's storage** is the one exception to the above: its included allowance and its hard block are the same number. There's no overage zone to sell, so there's nothing to leave room for — the cap you see is the cap that's enforced.

The reasoning: the pricing page sells metered overage as a real capability for both dimensions, so hard-blocking the moment you cross the included allowance would refuse to sell what's advertised. The backstop exists to catch runaway or abusive usage, not to cap normal overage billing. The Free plan isn't selling anything, so this reasoning doesn't apply to it.

Spend cap

Every paid plan also carries a **spend cap**: a monthly dollar ceiling on metered charges (storage above your allowance, extra private-repository external collaborators, and hosted-CI compute — metered from the first ten seconds, with no included allowance). Seats and add-on packs aren't counted against it, and downloads are always free. The cap is monthly on every plan, including annual seats, because metered usage bills monthly.

By default the cap equals your plan's seat charge — your seat count times the per-seat price — and an owner or admin can change it under **Billing → Spend cap**, either permanently or for 24 hours, 7 days or 30 days before it reverts to the default. Leaving the field blank, or choosing **Reset to plan default**, clears your own figure. Resetting to a default that sits below metered charges already accrued this month pauses new uploads immediately, exactly like lowering the cap by hand — the form says so before you click.

There's no upper limit on the figure. Metered compute counts against the same cap: when any metered dimension reaches it, growth pauses and overage bills up to the cap, never past it, exactly as the Billing form describes.

The cap is a hard ceiling on the overage line itself, not just a pause button: overage bills up to it and never past it. Because metering follows your *standing* usage for the period, the ceiling applies to the whole month — lowering it below overage you've already accrued drops that month's charged overage to the new figure and pauses further growth immediately, while raising it lets accrual continue. It never changes what's included: plan seats and committed packs still raise your allowance before any overage is counted.

When metered charges reach the cap, uploads that would add billable storage (and invites that would add a billed collaborator) pause, and new hosted-CI jobs are refused with a blocked `spend_cap_reached` run — running jobs finish.

Reading, streaming, deleting, renaming and everything else keep working, and nothing is lost: the desktop app holds paused saves and uploads them once the cap is raised or storage is freed. A refusal is a `402` with `code: spend_cap_reached` carrying `capCents`, `usedCents`, `remainingCents` and the figure the request would have reached; the desktop app shows "Spend cap reached" and keeps retrying, so a raise resumes the backlog on its own.

The Free plan and Enterprise have no spend cap on the storage/collaborator line — a Free-plan org's hosted CI runs under the Free-plan compute ceiling described above (after a card attach), and Enterprise compute is contract-priced.

A payment **dispute** (a chargeback on the subscription) is a different, deliberately sticky lockout rather than a cap: while it is open, the org's file reads and writes are refused with a `402` (`code: grace_period_inaccessible`) until the dispute is resolved with support. Nothing is deleted, and saves that were paused upload as soon as it is cleared.

Per-key budgets. A repository-scoped API key can also carry its own ceiling, set per key on the **API Keys** page (blank means no key budget). Enforcement uses the lower of the two — `min(org spend cap, key budget)` — and a key-budget refusal affects only that key's writes: other keys, user workstations, and reads are untouched. A key refusal names the key in the `402` (`keyLimitCents`, `keyId`).

Notifications. Owners and admins get an in-app notification when the cap is approached (50%, 80%, 100%), on the first refusal of a billing period, when a raise or a cleanup lets writes resume, and whenever the cap or a key budget changes. Billing's "approaching plan limits" banner shows the same levels.

Plan-gated features

By plan tier

Three features are **Starter and above**: encrypted private repositories, a dedicated per-org storage bucket (the default for a new paid org; a one-way enable in **Settings → Storage** for an org still on shared storage, including one that already has files — they are copied into the new bucket in the background), and an organization-enforced authentication policy (require MFA and/or a verified email for members).

A Free-plan org can create public repositories and private unencrypted repositories, and can't turn a policy requirement on; requesting managed or end-to-end encryption from the Free plan, or enabling a policy requirement, is rejected the same way any other plan-gated feature is.

A number of others are **Pro and Studio**: SSO (OIDC), SCIM/LDAP sync, folder ACLs with Windows DACL projection, audit-log export, the end-to-end-encryption tier, self-hosted seed nodes, and branch-mirror-to-S3.

A few are **Enterprise-only**: bring-your-own primary storage, SAML SSO (via an adapter — see Identity), and a self-hosted control plane. (EU/US data residency is the exception — available at no extra cost on every paid plan, not an Enterprise upsell; a Free-plan org uses shared storage and can't pin a region.)

These are enforced server-side with a real `402` rejection if you try to configure them on a plan that doesn't include them — not just hidden behind UI. Downgrading never disables something you've already configured; it only blocks configuring something new.

For the details on those features themselves, see Identity and Permissions.

Usage visibility

Plan standing, current usage, and your org's resolved entitlement set are visible to any org member, not just the owner. See Organizations, Repositories & Members for where that lives. The **Usage** tab (owners) breaks usage down by dimension — storage, egress, collaborators, hosted-CI and hosted-agent compute — and includes a **headless seats** line showing how many of the plan's included seats the org is using; Billing carries the 80% warning banner before minting is refused.

Operations

This page covers the tools you'll reach for as the person operating an org day to day: the audit log, retention policy, and connectivity diagnostics.

Audit log

The Activity tab in the web dashboard is a unified activity feed, open to any org member, covering (among others):

- ACL changes
- History events
- Quarantine incidents — both applied and released
- Branch-protection changes — per-branch flag toggles and glob protection-rule edits
- Org-level audit events (including member role changes and data-plane policy changes)
- Notifications
- Unlock requests
- Changeset commits
- Branch creates, archives, and switches
- Tag creates
- Comments added, resolved, and mentions
- RFI / submittal / approval workflow events (created, assigned, due-soon, overdue, response posted, revisions requested, closed, voided)
- Webhook changes (created, edited, rotated, re-enabled, or removed)

The feed is ACL-filtered server-side: owners see everything, members see events on entries they can read, and quarantine incidents and org-policy events repository to empty for non-owners. Everything lands in one feed instead of being scattered across separate logs. It refreshes automatically (within about a minute while the tab is open, paused in background tabs) so new events show up without reloading the page.

You can export it to CSV, capped at 100k rows per export. The export endpoint is owner-only *and* gated on the `audit_log` entitlement (Pro plan and above): a non-owner gets a 403, and an owner on a plan without the entitlement is told to upgrade. The live in-dashboard feed and the underlying event capture stay ungated on every plan — only the bulk CSV export is entitlement-gated.

CSV is the bulk-export path. An outbound webhook also forwards the same underlying events to your own endpoint as they are recorded — typically within seconds, with an hourly sweep as a backstop — which covers some of the same ground for a system willing to receive and store them itself. See Security for the full list of what's still on the roadmap.

What the CSV export contains

Each export starts with a few `#` provenance rows (format version, generation time, requested window, row cap) that spreadsheet apps skip, then one row per event with these columns: `occurred_at_iso`, `kind`, `actor_user_id`, `actor_display_name`, `actor_email`, `repository_id`, `repository_name`, `entry_id`, `entry_name`, `summary`, `detail`.

The rows are a merged, deliberately un-sorted-across-sources view of every source in the requested window; sort by `occurred_at_iso` after import if you need a single timeline, and check the `X-Activity-Export-Truncated` response header in the web export. The CLI prints the CSV without surfacing that header, so a CLI export is silently capped at 100,000 rows — narrow `--since` / `--until` if you need a complete window.

Audit-event retention. Activity, ACL, and audit rows are pruned on the org's retention window — the same 1–3650 day setting that governs history and trash, defaulting to 90 days when nothing is set. Per-user retention preferences apply to history and trash only; these activity/ACL/audit rows follow the org window. Plan for exports or webhook ingestion if your compliance position needs records kept longer than the window in force, and confirm the window on **Settings → Organization** before relying on it.

Ransomware quarantine

Swarmfile watches for the signature of encryption malware: a large number of *distinct entries* overwritten in a short window, counted per user and machine, with a per-user total across that user's machines as a backstop. It responds in two tiers. The detection mechanics are covered in Security; this section is about the admin side of an incident.

- **Alert bar** (a configurable number of distinct entries in the window): crossing it records an *alert-only* incident and notifies org owners for review. Nothing is blocked.
- **Block bar** (a higher multiple of the alert bar): crossing it hard-quarantines the user: their writes stop reaching the hub (on Windows the drive stays mounted and ordinary writes are accepted locally and held (a rename is refused, so an app that saves by renaming reports a failed save); on macOS and Linux the write is refused), and lock acquire/renew both refuse, so they can't keep working even mid-lock. A quarantine applies to the user on every machine. A single large operation (one big commit, for example) can reach either bar on its own. If your team routinely rewrites hundreds of files in one operation, raise the thresholds (below) rather than leaving them to be quarantined.

A quarantine is enforced on the writes themselves, not only on locks: a content save, a conflict resolve, a changeset commit or checkout, a `git push` apply, a branch or merge-request merge, LFS reflect/upload and a web upload are all refused with the `quarantined` reason while the block is active.

Reads keep working. The computer is **paused**, not unmounted: `swarmfile status` prints `quarantine: WRITES PAUSED` with the start time, the drive's write-refusals panel names the files it refused or held, and queued saves resume on their own when an owner or admin releases the block.

A repository/agent API key and a human-triggered CI job inherit the block; an actorless scheduled job must be stopped or revoked.

Owners and admins can tune either bar per organization under **Settings → Quarantine → Detection thresholds**; a blank field keeps the default, and the card shows the effective value and where it comes from.

Raising the alert bar moves the inherited block bar with it (an explicit block bar doesn't move). The org's daily container-preview budget — with today's org-wide spend against it — lives on **Settings → Usage → Preview generation budgets**: previews/day default to 500 per repository and 2,000 per org, a blank field keeps the default, and `0` means no limit.

The source caps are fixed — video proxies accept sources up to 16 MiB (poster extraction reads the first ~16 MiB), and point-cloud (`.las` / `.laz`) previews up to 32 MiB; File Previews covers the rest.

Alert-only incidents that no one acts on auto-expire after 7 days, so a legitimate bulk edit doesn't linger as a flag (and doesn't suppress future alerts for that user). Blocks never auto-expire — they require a human to clear.

Manage incidents from the **Quarantine** view in the web dashboard. It lists active and historical incidents; each row is one click to **Release** (lift a block) or **Dismiss** (clear an alert-only flag), and the release records the reviewer's judgment: **false positive** (legitimate activity) or **confirmed incident** (contained real attack), or no judgment. Admins can also manually quarantine a user out-of-band — for example, in response to an incident report that didn't trip the detector. Quarantine management is available to admins and owners.

Review requests. A blocked user can send one request-only appeal per 24 hours at `/dashboard/quarantine-appeal` (2,000 characters, no files). It never lifts the block and never changes access. Owners and admins see the request under **Quarantine review requests** in the Quarantine view and can **Acknowledge** or **Close request** — those are records only; use **Release** below to lift the block.

Published false-positive rates. The same view shows the org's measured detection quality for incidents detected in the selected window (30, 90 or 365 days; manual quarantines are excluded — there was no detector decision): the **block false-positive rate** and the **alert false-positive rate**, each with its sample size.

The definition is deliberately narrow, because a rate without one is marketing: the numerator is incidents released WITH a `false_positive` judgment, and the denominator is incidents released WITH any judgment — unreviewed incidents are excluded, and an empty sample reads "No data", never 0%. The unit is one detector decision in one repository, so a burst seen by several repositories counts as several decisions.

Only counts are published; no file names or content enter the metric.

Deployment-wide rate. A rate covering every organization is published only as a quarterly report over a rolling 12-month window, and only when each rate has at least 100 judged incidents; otherwise we publish the sample size and say it is insufficient. It is never a live badge. The in-product rate above is your organization's.

Roll back the organization in one click. An incident row offers **Roll back all repositories** (one action, with a typed confirmation): every repository in the org has its default branch restored to the incident's point — 60 seconds before the recorded burst start when the detector saw one, otherwise the review window (24 hours before detection, or the detector window if longer) — as a **new commit**.

Nothing is rewritten — the damaged versions stay in history, so the rollback is itself undoable — and each repository is restored independently. The result names every repository it refused (with the reason), with counts for those it restored or started as a background job (very large trees); a repository created after the point is left alone.

A repository relay that fails is retried automatically up to three times; after that, **Retry unfinished** re-runs only unfinished or relay-failed repositories when a run is partial (restored repositories are never re-run). **Recover to branch** remains the review-first alternative: it puts the content as it stood at the incident's review point on a recovery branch and leaves every default branch untouched.

The org-wide rollback is owner-only; restoring one repository stays available from that repository's own history. Owners and admins get a bell notice when a rollback finishes or partially fails; the web notice opens that job's result, which lists every repository's outcome (the desktop tray notice opens the Quarantine view). Terminal job and appeal rows are pruned after 90 days; the audit log keeps the outcome, and each restore is a new commit you can undo from that repository's history.

Access revocations

Removing a member, revoking a guest, deprovisioning a user through SCIM or LDAP, revoking a device key, or invalidating a directory group membership doesn't just delete one row: Swarmfile sweeps every repository in the org so cached peer credentials, wrapped keys, and access grants stop working there too.

That sweep is durable — it runs across repositories in the background, retries failures with backoff, and parks on a repository it can't reach instead of skipping it silently.

Settings → Access revocations (owners and admins) shows where every sweep stands: in progress, parked repositories with the reason, and finished sweeps. A parked repository carries a **Retry** — fix the cause and re-run the sweep from the same view; each attempt is recorded in the audit log. Revoking a repository API key or a personal access token rides this same sweep to close that credential's live connections.

Because a sweep reaching a client is what makes it delete its cached copy, each engine reports a **purge receipt** when it has applied a revocation: the view shows how many devices confirmed it, how much cached data they removed, and which device each confirmation came from (attributed to the signed-in identity — a machine label is self-declared). So "dispatched" and "applied everywhere" are separate, visible facts.

`GET /orgs/:org/admin/security-fanout` returns the sweep view for scripts, and `GET /orgs/:org/admin/revocation-receipts` the receipts. See Organizations, Repositories & Members for member removal and End-to-End Encryption for device-key revocation.

Retention & garbage collection

Trash and soft-delete retention is configurable, from 1 to 3650 days — owners set the org-wide window in **Settings → Organization**, and an individual user can set their own window in account settings, which overrides the org default for them. The end-user side of this — restoring a deleted file, browsing history, rolling back a change — is covered in Trash, History & Rollback; this page is about the org-level policy knob that governs how long deleted data is kept before garbage collection reclaims it.

Set the retention window based on your org's compliance and recovery needs — a longer window costs more storage but gives you (and your users) a longer recovery runway. The default when nothing is set is 90 days.

When file content is reclaimed. Removing the last reference to a file's content frees that content in a background sweep. That happens when you purge it from Trash, when the daily cleanup expires trash and old versions, or when an abandoned upload is cleared. Sweeps run:

- **At the next daily pass:** after a purge from Trash, or after an abandoned upload is cleared.
- **At the next daily pass, for any amount:** on a plan with a hard storage cap, or an org close to its spend cap, whatever was released.

- **Otherwise:** once enough content has accumulated, with large files counting for more, and at least once a week.

Billed storage counts only files you still have, so this timing does not change your bill. On a plan with a hard storage cap, released content counts against the cap until its sweep has run.

New-repository history retention. Independently of the window, every repository carries a history-retention choice: **Keep every saved version** (the default) or **Named commits only**. It decides whether individual per-save versions survive the window at all — on a keep-everything repository the window controls when aged history moves to cheaper cold storage, not when it is deleted, so on such a repository a short window does not delete history.

Owners set the default for newly created repositories with **New repositories keep every saved version** in **Settings → Organization**; a creator's explicit choice in a creation form still wins, so this is a default, not an enforcement.

An org whose data-minimization policy wants saves to expire should turn it off (or clear the flag on existing repositories with `PATCH /orgs/:orgId/repositories/:repositoryId` / `swarmfile repository set-keep-full-history false`).

Repository storage limit

Separate from your plan's storage allowance (the size of the files themselves — see Billing & Plans), each repository keeps its file information and history — names, folders, versions, locks, comments — in its own store with a fixed limit. It takes a very large repository to approach it: roughly ten million files and versions combined, since a 1 TiB file costs the same here as a 1 KB one. It isn't billed.

- **Where to see it.** A repository's **Storage** tab (owners and admins) shows how much of the limit is used, whether new writes are being accepted, what the last daily cleanup removed, and — when any exist — access revocations still waiting to reach a device (the history window in force lives on the repository's **Repository settings** tab). **Settings → Repository storage** lists every repository in the org that is nearing or at its limit, worst first.
- **Early warning.** At the early-warning threshold (5 GiB for a standard repository, 2 GiB when it keeps every version), the org's owners and admins get a "Repository storage" notification (in the inbox, and by email unless muted per kind in Settings). History retention may also start shortening automatically — see Trash, History & Rollback.
- **At the limit.** Adding files, saving new versions, copying, merging, staging, and git-LFS pushes are paused with an "insufficient storage" (`507`, `repository_storage_full`) error. Nothing is lost: the Desktop App keeps changes on each person's computer and syncs them once space is available. Everything needed to recover keeps working — deleting, purging trash, restoring, renaming, moving, locks and commits.
- **Freeing space.** Purge deleted files from Trash, delete old branches, or shorten the history retention window (**Settings → Organization**, owner-only). On a keep-every-version repository the window only controls cold archiving; if history genuinely isn't needed, switch the repository to **Named commits only** as well. Space is reclaimed as soon as rows are removed, so writes resume without waiting for the next daily pass. A repository that keeps growing past this is usually better split into several repositories.

Swarmfile also watches these stores from the platform side and pages its own operators when a repository approaches its ceiling, so growth is caught before it pauses writes — there is nothing for you to configure.

Write pause

Each repository also has a daily budget for how much it writes to its own file and history records (file content doesn't count). Normal work stays far below it. A repository that goes well past it in one day, usually because a script, agent or integration is stuck repeating the same work, is paused in two steps until midnight UTC:

- **Background maintenance pauses first.** Cleanup sweeps and background merge planning stop for the rest of the day. Saves, reads, commits, merges, pushes and jobs carry on as normal.
- **If writing keeps going, to several times the budget, new work is refused.** Starting a background job (a large commit, revert, checkout or restore), starting a merge, or starting a `git push` answers `503` with the code `do_write_budget_exceeded`, a `retryAtMs` time in the body and a `Retry-After` header; retry after that time. Work already in progress keeps going, slowly, so nothing is left half-applied. Deletes and purges keep running at normal speed, so the data causing the problem can still be removed. Saves and reads are not affected. The Desktop App holds a commit refused this way and retries it when the pause lifts, without counting it as a failed commit. A background job waiting out the pause reports when it will resume: `GET /jobs/:id` includes `parkedUntilMs`.

The pause lifts on its own at midnight UTC. An org owner or admin can release it sooner from the repository's **Repository settings** tab, which shows **Writes paused** with a **Release write pause** button while a pause is in force (API: `POST /orgs/:orgId/repositories/:repositoryId/write-budget/release`).

A release is allowed once per repository per UTC day; a second one is refused with `write_budget_release_used`. Each release is recorded in the org audit log and the activity feed. After that, only Swarmfile support can release the repository again that day, so contact us if it is still paused and the cause is fixed.

Swarmfile's operators are alerted to every pause and look for the cause from their side as well.

Long-running background jobs

Large cascades don't run inline. Above the hub's **5,000-entry** cap, folder delete, restore, purge, revert and checkout — plus `repository delete` — run as durable background jobs: the request answers `202 { jobId }`, and the engine, web dashboard and tray follow the job and synthesize the original outcome (the root change lands before the `202`, so the operation's effect is already visible). Jobs are durable and survive a dropped connection, a closed laptop, or an engine restart.

From the CLI, `swarmfile op` manages them:

- `op list` shows recent operations.
- `op status <id>` reports state and live `done` / `total` progress (exit `0` succeeded, `1` failed, `2` canceled, `3` still running).
- `op wait <id>` waits and exits as the original command would.
- `op cancel <id>` stops the operations that can safely stop part-way (`materialize`, `hydrate free-space`, `git backfill-oids`, `git index`, and `checkout` / `revert` while still staging) — a cancel that lands after the staged apply has begun is declined (`too_late`) and the operation ends with the commit's real outcome, while commits, merges, deletes and `undo` / `redo` complete or fail on their own and refuse (`not_cancellable`).

A command that starts a long operation and doesn't follow it prints the job id and exits `3` (with `--no-wait`, or when you press Ctrl-C to detach), and the operation carries on in the engine.

Office caches

A self-hosted seed node runs headless; its operational surface is **Settings → Office caches**. The page lists the fleet with liveness (a cache reads offline after roughly 2.5 minutes without an announce, or **Awaiting approval** when your organization approves nodes), the bytes each cache served this week, and the enrollment wizard that mints a cache's credential and generates its `seed.env`. Owners and admins receive a weekly report email and one alert per day-long outage; each is muteable per-kind under **Settings → Notifications**. Setup, sizing and hardware guidance: Self-Hosted Seed Nodes.

Connectivity diagnostics

Swarmfile ships two ways to run the same connectivity probe suite:

- **`swarmfile doctor`**, proxied through a running engine.
- **`swarmfile-doctor`**, a standalone binary that doesn't need a running engine at all — useful for a machine that isn't mounting anything yet, or for isolating whether a problem is the engine or the network.

Both run the same checks: DNS, hub reachability, cloud storage, the peer-to-peer layer, mDNS discovery, and OIDC/auth. Output is colorized pass/warn/fail for a human to read, or `--json` for attaching straight to a support ticket.

Diagnostics also run automatically in the background on a periodic basis, not just when you invoke them by hand, and server-side deep-health probes run from the other direction — so a problem can surface from either side. Always-public liveness endpoints exist separately, for basic uptime monitoring.

One probe worth knowing about specifically: the **plan** probe. If a self-hosted seed or NAS node is running under a plan that doesn't include the seed-node entitlement, this probe fails clearly and explicitly, instead of the node just silently misbehaving or repeating the same log line forever. If you're setting up a seed node, see Self-Hosted Seed Nodes for the plan requirement.

Reach for `swarmfile doctor` any time something feels wrong and you're not sure whether it's your network, your account, or the service — it's the fastest way to narrow that down, and its output is what support will ask for first. Full flag reference is at swarmfile-doctor.

Webhooks

Webhooks let an external system react to what happens in your org — a file changed, a comment was posted, a branch was created — without polling. Swarmfile sends a signed HTTP POST to a URL you control every time something you've subscribed to happens. There are two independent kinds: an **org-wide webhook**, managed by an owner and shared by the whole org, and a **personal webhook**, a private notification channel for just your own account. Both use the same signing and delivery mechanics, described once below.

It's free and self-serve on every plan, including Free — there's no price gate and no sales conversation needed to turn it on.

How a delivery works

Every delivery is a single `POST` request with a JSON body:

```
{
  "apiVersion": 1,
  "id": "evt_...",
  "kind": "file_created",
  "data": { }
}
```

- **`apiVersion`** is a schema-version tag for the envelope shape itself (currently `1`). It's separate from `kind` — new event kinds get added over time without bumping this number; it only changes if the envelope's own top-level shape ever does.
- **`id`** is a stable, unique id for this event.
- **`kind`** names what happened — see Event kinds below.
- **`data`** is the event's own payload, shaped differently per kind.

Two headers travel with every request:

- **`Swarmfile-Event`** — the same value as the body's `kind`, so you can route on the header without parsing JSON first.
- **`Swarmfile-Signature`** — `t=<unix timestamp>,v1=<hex-encoded HMAC-SHA256>`, the same convention Stripe uses for its own outbound webhooks. Verify it before trusting anything in the body.

Verifying the signature

Compute an HMAC-SHA256 over the string `<timestamp>.<raw request body>` using your webhook's signing secret, and compare it (in constant time) to the `v1` value from the header. Use the **raw, unparsed** request body for this — not a re-serialized version of the parsed JSON, which can differ in whitespace or key order and would make the signature not match.

```
expected = hex(hmac_sha256(secret, `${timestamp}.${rawBody}`))
```

We also recommend rejecting a request if `t` is too far in the past (five minutes is a reasonable window) — this protects against a captured request being replayed later.

Node.js

```
import { createHmac, timingSafeEqual } from "node:crypto";

function verifySwarmfileSignature(secret, rawBody, signatureHeader) {
  const parts = Object.fromEntries(signatureHeader.split(",").map((kv) => kv.split("=")));
  const expected = createHmac("sha256", secret).update(`${parts.t}.${rawBody}`).digest("hex");
  const expectedBuf = Buffer.from(expected, "hex");
  const givenBuf = Buffer.from(parts.v1, "hex");
  return expectedBuf.length === givenBuf.length && timingSafeEqual(expectedBuf, givenBuf);
}

// Express example — mount with a raw-body parser so `req.body` is the
// exact bytes Swarmfile sent. Re-parsing and re-serializing the JSON
// before verifying is the single most common way to make this fail.
app.post("/webhooks/swarmfile", express.raw({ type: "application/json" }), (req, res) => {
  const signature = req.header("Swarmfile-Signature");
  if (!signature || !verifySwarmfileSignature(process.env.WEBHOOK_SECRET, req.body, signature)) {
    return res.status(401).send("invalid signature");
  }
  const event = JSON.parse(req.body.toString("utf8"));
  // ... handle event.kind / event.data
  res.status(200).end();
});
```

Python

```
import hashlib
import hmac

def verify_swarmfile_signature(secret: str, raw_body: bytes, signature_header: str) -> bool:
    parts = dict(kv.split("=", 1) for kv in signature_header.split(","))
    payload = f"{parts['t']}.".encode() + raw_body
    expected = hmac.new(secret.encode(), payload, hashlib.sha256).hexdigest()
    return hmac.compare_digest(expected, parts["v1"])

# Flask example — request.data is the raw, unparsed body.
@app.route("/webhooks/swarmfile", methods=["POST"])
def swarmfile_webhook():
    signature = request.headers.get("Swarmfile-Signature", "")
    if not verify_swarmfile_signature(WEBHOOK_SECRET, request.data, signature):
        return "invalid signature", 401
    event = request.get_json()
    # ... handle event["kind"] / event["data"]
    return "", 200
```

Manual verification (curl / openssl)

Useful for confirming your receiver's math against a real delivery while debugging, without writing any code — save the raw request body Swarmfile sent to `body.json`, then:

```
TIMESTAMP="1700000000"   # the "t" value from the Swarmfile-Signature header
SECRET="whsec_..."       # your webhook's signing secret

printf '%s.' "$TIMESTAMP" | cat - body.json | openssl dgst -sha256 -hmac "$SECRET" -hex
```

Compare the resulting hex digest to the `v1` value from the header — they should match exactly.

Your endpoint must be reachable on the public internet. Localhost, private network ranges (`10.x`, `172.16–31.x`, `192.168.x`), and cloud metadata addresses (`169.254.169.254` and similar) are rejected at creation and re-checked immediately before every delivery, so pointing a webhook at internal infrastructure never works — this is a deliberate security boundary. Your endpoint must also respond directly: HTTP redirects are not followed.

Each delivery attempt times out after 10 seconds. Respond with any `2xx` status to acknowledge success; anything else counts as a failure for that attempt.

Org-wide webhooks

Manage these from **Settings → Webhooks**. Only owners can create, edit, or remove them — a webhook is a data-egress control, the same trust tier as an API key or an SSO configuration, not a viewing convenience open to every member.

Automating webhook management

A repository-scoped API key (see Runner (Headless CI)) **cannot** manage webhooks, or reach any other org-wide admin surface (org settings, ACLs, directory sync) — that's deliberate: an API key is confined to one repository and is never treated as an owner, no matter who minted it.

If you need to manage webhooks from a script rather than the dashboard — provisioning one as part of an infra pipeline, say — mint a **personal access token** instead, from **Settings → Access Tokens**. Unlike an API key, a personal access token isn't a separate service identity: it acts as *you*, with your own live role, so it can do anything your own account can do here and nothing more. It stops working the moment your own access does (role change, removal from the org), with no separate cleanup needed. It's a general-purpose credential, not specific to webhooks — the same token also works against any other route your role can reach.

Creating one

You need an endpoint URL. Everything else is optional:

- **Description** — a free-text label to tell webhooks apart in the list.
- **Event kinds** — leave this unset to receive every kind of event, or pick specific ones (file changes, comments, branches and tags, sharing/ACL changes, unlocks, RFIs, merge requests, uploads) to only receive what you actually care about.
- **Custom headers** — up to 5 extra `name: value` pairs sent on every delivery, alongside the standard `Swarmfile-Signature` / `Swarmfile-Event` pair. The most common use: an `Authorization` header carrying a bearer token your own receiver's gateway expects, so it can gate on that in addition to (or instead of) verifying the signature.

Once created, the signing secret is shown **exactly once** — copy it immediately. Swarmfile never displays it again; if you lose it, rotate to a new one (below).

Custom headers

A header value is exactly as sensitive as the signing secret itself — it's frequently going to *be* a credential — so it gets the same write-once treatment: once saved, Swarmfile can show you which header **names** are configured (so you can tell what's set without guessing), but never the values again. Editing headers means re-entering the full set you want, not patching one value in place; leaving the editor empty when editing a webhook keeps whatever's already configured, untouched.

A handful of header names are reserved and can't be overridden: `Content-Type`, `Content-Length`, `Swarmfile-Signature`, `Swarmfile-Event`, `Host`, `Connection`, `Transfer-Encoding`, and `Upgrade` — these either carry the protocol's own meaning or aren't things a webhook delivery can meaningfully override.

Delivery timing

Deliveries are dispatched when the event is recorded — typically within seconds. The hourly tick is a backstop that catches anything a live dispatch missed, so in the rare case a delivery is delayed it can take up to an hour.

A failed delivery is retried automatically with increasing delays (roughly 30 seconds, 2 minutes, 10 minutes, 30 minutes, then 1 hour) for up to 6 attempts total before it's given up on.

Auto-disable

If 5 events **in a row** each exhaust all of their retry attempts, the webhook is automatically disabled — Swarmfile stops trying to deliver to it rather than silently failing forever. You'll get a notification (bell, email, and your own personal webhook if you have one configured) when this happens. Fix whatever was wrong with your endpoint, then click **Re-enable** — delivery resumes from exactly where it left off, catching up on everything that happened while it was disabled, rather than skipping that window.

Editing, rotating, and removing

- **Edit** changes the URL, description, or event-kind filter without losing the webhook's id, delivery history, or audit trail.
- **Rotate secret** mints a new signing secret for an existing webhook. This is a **hard cutover** — the old secret stops verifying the instant you rotate, with no overlap window. Update your receiver's stored secret promptly; deliveries sent before you do will look unsigned to it.
- **Delete** removes the webhook and its delivery history permanently.

Testing and retrying

- **Send test** fires one synthetic event at your endpoint immediately, signed with the real secret, so you can confirm your receiver and signature verification actually work without waiting for a real event. It shows up in the delivery log as kind `webhook_test`, but never counts toward auto-disable.
- **Retry** re-sends one specific failed delivery from the log, using the exact payload that attempt originally sent.

Both actions have a short cooldown per webhook (a few seconds) to stop a rapid click (or a script) from hammering your endpoint faster than intended.

Limits

An org can have at most 20 webhooks. Every create, edit, delete, rotate, and re-enable is recorded in the org's Activity feed under the "Security" filter, so there's an audit trail of who changed what and when.

Personal webhooks

A personal webhook is a private notification channel for your own account, alongside the bell and email — independent of any org-wide webhook an owner may have set up. Configure it under **Settings → Notifications** (the Personal webhook panel): a URL, an on/off toggle, and the same per-kind mute controls email already has (see Notifications & Inbox) — you can send comment mentions to your webhook while muting the noisier "file changed" kind, for example.

It uses the exact same envelope, headers, and signature scheme described above, with its own independently-generated secret (**Send test** and **Rotate secret** work the same way as the org-wide version).

The one real difference: a personal webhook is fire-and-forget. There's no delivery log, no automatic retry, and no auto-disable — it's held to the same bar as email, not the fuller governance treatment an org-wide webhook gets. If your endpoint is down when an event fires, that event is simply not redelivered later.

Clearing the URL turns it off. It's also cleared automatically if you're removed from the org, so a departed member's endpoint doesn't keep receiving signed events after they've lost access.

Event kinds

Event kinds match the org Activity feed's own categories — file changes, commits, sharing/ACL changes, branches and tags, comments, unlocks, uploads, RFIs, and merge requests, plus a few webhook-governance-specific kinds. The exact, current list is what you see in the event-kind picker when creating or editing a webhook — that list is generated from the same source the Activity feed itself uses, so it can never drift out of date the way a hardcoded list on this page eventually would.

A `git push` through the git remote helper counts as ordinary activity too: pushed commits, files, branches and tags land the same events a drive-side change produces, so a git workflow needs no separate hook.

Merge request events

`mr_created`, `mr_reviewed`, `mr_merged`, `mr_closed`, `mr_reopened`, `mr_assigned`, and `mr_review_requested` fire as a merge request moves through review. Every one of these carries `merge_request_id`, `mr_number`, `mr_title`, `mr_status`, `repository_id`, and `by_user_id` in `data`.

`mr_reviewed` additionally carries `verdict`, one of `"approved"`, `"changes_requested"`, or the non-blocking `"commented"` — so you can route on the outcome of a review without a follow-up API call. `mr_assigned` additionally carries `assigned_principal_id` and `assigned_principal_name` (the user or group that was assigned to land it); `mr_review_requested` carries the same pair under `requested_reviewer_principal_id` / `requested_reviewer_principal_name` (the user or group whose review was requested — distinct from being assigned to land it).

Every one of these also carries `assignee_names`, `reviewer_names`, and `label_names` — the merge request's *current* assignees, requested reviewers, and labels at the moment the event fired (all plain string arrays, empty if none), not just what changed. A "someone approved this" webhook consumer can tell who's on the hook for it without a follow-up API call.

A few fields are best-effort and can be absent: `source_branch_name` / `target_branch_name` are `null` if the branch itself has since been deleted past recovery, and `url` (a deep link straight to the merge request's page) is only present when the hub deployment has a site URL configured — an Enterprise on-prem deployment that hasn't set one won't include it.

`mr_created` does not fire while a merge request is a draft — on the theory that a draft isn't something you want a webhook waking anyone up for yet.

That suppression is permanent, not deferred: marking a draft ready (`mr ready`) does not retroactively fire the `mr_created` a subscriber missed at creation time, so a draft-opened merge request may reach `merged` / `closed` with no `mr_created` ever delivered for it at all.

`mr_assigned` and a manually-triggered `mr_review_requested` (someone explicitly requesting a review by hand) are unaffected and fire normally even on a draft — those are the events to rely on if you need a webhook signal that a specific draft exists.

The one exception is `mr_review_requested` fired automatically by a branch-protection auto-reviewer at creation time: that one IS suppressed on a draft, same reasoning as `mr_created` — the reviewer is still attached, the webhook just doesn't fire for it until something later (marking the MR ready, then a subsequent manual action) generates a notifying event.

Where to go next

- Automating Swarmfile — the credentials and surfaces for driving Swarmfile from scripts and CI.
- Notifications & Inbox — the bell, email, and quiet hours, which a personal webhook sits alongside.
- Operations — the org-wide Activity feed a webhook's own governance actions appear in.
- Permissions — how ACLs shape what an org member can see and do, for context on why webhook management is owner-only.
- OpenAPI contract — the hub's machine-readable API contract (this webhook reference included) as an open-source repository, for generating clients or validating payloads.

Deploying to Your Team

For rolling Swarmfile out to a fleet rather than having each person install it by hand — silent install, GPO/MDM push, and what a freshly-provisioned machine needs (or doesn't) before someone signs in.

Windows

Prerequisites first, then the app

The download most people should use is the bundled installer (`Swarmfile-Setup.exe` / `Swarmfile-x64-Setup.exe`), which chains WinFsp, the VC++ Redistributable, and WebView2 ahead of the app itself, skipping anything already present. For a scripted mass deployment, install those prerequisites through your own tooling (GPO, SCCM, Intune) first, then push the **bare** `.msi` — it's published under its own stable name specifically for this, and unlike the bundle it does not install those prerequisites itself:

```
msiexec /i Swarmfile-x64.msi /qn /norestart
```

If a probe-backed prerequisite is missing, the MSI refuses during install with a message naming it — WinFsp or the VC++ Redistributable (instead of applying and failing later with a side-by-side or mount error). WebView2 is required at first launch but is not part of that gate; the bundle installs it, so stage it in the same task sequence when you use the bare MSI. The checks are `Installed OR …`-guarded, so repair and upgrade are never blocked by a missing prerequisite.

`/qn` (fully silent) and `/norestart` are the same flags this repository's own CI uses to verify the installer end to end, not a guess — that's a meaningfully stronger guarantee than "should work" for an MSI. Uninstall is the mirror image: `msiexec /x <ProductCode> /qn /norestart`.

The app install itself doesn't require a reboot — it closes and relaunches Explorer to pick up the shell integration, not the OS.

A scripted rollout never sees Windows' SmartScreen prompt. If someone runs a freshly downloaded `Swarmfile-Setup.exe` by hand, Windows shows **More info → Run anyway** until the signing identity builds up download reputation — expected, not a failed download. The `Setup.exe` / `.msi` are Authenticode-signed; the `-bin.zip` payload is not. See Getting Started.

It installs per-machine, which is what makes GPO work

The MSI installs per-machine (not per-user), so a **Computer Configuration** software-installation policy is the right kind of GPO push — you don't need a per-user policy for it to reach everyone who logs into that machine. Auto-launch for the Desktop App and engine is likewise registered machine-wide, so every user account on a pushed machine gets Swarmfile running automatically at login, not just whichever account happened to run the installer.

It also installs the command-line tools (`swarmfile`, `swarmfile-lfs`, `swarmfile-doctor`, `swarmfile-migrate`, `swarmfile-search`, `swarmfile-verify-history`, `swarmfile-seed`, `swarmfile-runner`, `git-remote-swarmfile`) into `C:\Program Files\Swarmfile` and appends that folder to the machine `PATH`, so a pushed machine gets the CLI with no per-user setup.

The MSI also adds an inbound Windows Firewall rule for the engine: UDP, local subnet only, Domain and Private profiles. Without it, a silently installed machine can't discover or be reached by colleagues' machines on the office LAN, and every transfer goes through the cloud. The rule is removed on uninstall. If your GPO ignores locally added firewall rules, push the equivalent rule yourself. See Network Requirements for the exact rule and a `New-NetFirewallRule` command.

First sign-in is still interactive

There's no way around this today: an employee's first launch requires them to sign in through the Desktop App. For password sign-in, that's a plain email/password form inside the Desktop App itself — no browser involved at all. For SSO, it's a real browser-based flow: the Desktop App hands off to your system's default browser for the IdP's login page via PKCE, then returns control to the Desktop App.

Nothing in the product supports a fully unattended, no-user-interaction first login for a regular desktop seat, regardless of which method you use. If you need machines that never have a human sign in — a render farm node, a CI runner, a headless seed node — that's a different mechanism entirely: see Headless and service machines below.

What happens after sign-in

What *is* automatic: once someone signs in, the engine adopts an org and repository on its own, and it's more permissive than you might expect — it doesn't wait for an unambiguous single choice. It picks the *first* org the account can reach (even for a multi-org account — there's no picker on first run), and within that org it picks the `default`-slugged repository if one exists, else just the first repository in the org.

Orgs start with **zero** repositories: creating the first one is a deliberate action in the dashboard (or `swarmfile repository create`), and until an org has one there is nothing for the drive to mount. Once every org you're rolling out has a repository, "just sign in and it works" is accurate for nearly every org shape — the drive mounts to *some* repository on first launch, not necessarily the one you'd have picked for them.

If a newly-provisioned employee needs a specific non-default repository, that's a one-click switch in the Desktop App's org/repository picker after first sign-in, not something to pre-plan around. The same switch is scriptable without the Desktop App — `swarmfile workspace list`, then `swarmfile workspace switch --org <id> [--repository <id>]` — and, like the picker, it needs no re-authentication: the session already covers every org the account can reach. See Organizations, Repositories & Members.

If your org has SSO configured with an auto-provisioning group (see Identity), that removes the manual-invite step for anyone in that IdP group — they can sign in the moment their machine has Swarmfile installed, with no separate Swarmfile-side invite to send. SCIM alone doesn't do this on its own: it syncs your directory into Swarmfile, but org access still requires that SSO group match or a manual invite.

If you're self-hosting the control plane

Everything above assumes the standard hosted service, where a fresh install already knows which hub and identity provider to talk to. A self-hosted control plane — the same software running entirely on infrastructure you operate — is available on Enterprise.

With it, the deployment needs to know which hub and identity server are yours *before* first launch: pre-stage a `config.json` next to the install with your `hub_url`, `oidc_issuer`, and `oidc_client_id` set, and push that file alongside the installer through the same GPO/MDM channel. See Engine Config File for the full key reference, and talk to us to scope a self-hosted deployment.

macOS

The installer is a single signed and notarized universal `.pkg` (Intel and Apple Silicon), built to work with standard MDM package-deployment tooling (Jamf, Kandji, Mosyle, Apple Business Manager) — its own setup step recognizes when it's running under an MDM-driven install session rather than a human at the keyboard, and logs accordingly so it shows up sensibly in your MDM's policy logs rather than as a silent failure. If you hit anything MDM-specific that doesn't behave the way you'd expect, let us know.

Linux

The `.deb` pulls its runtime dependencies (`libwebkit2gtk-4.1-0`, `libgtk-3-0`, `libfuse2`) in automatically through normal `apt` / `dpkg` dependency resolution — nothing extra to stage on a normal online install, though an offline or mirrored image needs all three available in the mirror.

It enables the engine and Desktop App as per-user systemd units globally, so **new logins** pick them up with no further action; a user already logged in at install time won't have them start automatically until their next login (or you start them by hand). The Desktop App unit is conditioned on a graphical session and quietly does nothing on a headless box, which is the correct behavior for a Linux seed/NAS node.

One thing the package does *not* handle: some distributions require `/etc/fuse.conf`'s `user_allow_other` set, or the installing user added to a `fuse` / `plugdev` group, before a non-root user can mount via FUSE. If mounting fails with a permissions error on a specific distro, check that first — it's a FUSE/distro configuration matter, not something the `.deb` sets up for you.

The `.deb` doesn't touch your firewall either. If machines run ufw or firewalld with inbound traffic denied, allow UDP from the local subnet so LAN peers can find each other. Network Requirements has ready-made rules.

Headless and service machines

Render-farm nodes, CI runners, and self-hosted seed nodes don't have anyone signing in interactively, so they use a different credential entirely: a repository-scoped API key (`swarmfile api-key create <name> --repository <id>`, admin/owner-only), set as `SWARMFILE_API_KEY` on the target machine. An engine started with that env var skips the browser sign-in flow completely — no human interaction, ever — but is confined to exactly the one repository the key was minted for.

This is the right tool for provisioning unattended fleet machines as part of a rollout; it's a separate concern from getting Swarmfile onto employees' desktops, and the two shouldn't be conflated — a desktop seat always goes through interactive sign-in (above), a service machine never does.

Self-hosted seed nodes have a guided path aimed at exactly this: **Settings → Office caches** generates a cache's complete `seed.env` for a chosen office and repository (scope, hub URL, and a freshly minted repository-scoped key, shown once), then lists the whole fleet with liveness and the bytes each cache served this week. Owners and admins get a weekly report email and one alert per day-long outage, both muteable under **Settings → Notifications**. See Self-Hosted Seed Nodes for setup and sizing.

Each key can also be given its own request ceiling (`swarmfile api-key rate-limit <id> <max>`, admin/owner-only), capped at the org's per-user limit — the lever for letting a fleet burst without raising a person's window. See Organizations, Repositories & Members.

Repairing or reinstalling at scale

`swarmfile-doctor --repair` reinstalls the desktop app from your update channel (it needs the desktop app installed on that machine). Add `--yes` to skip the confirmation prompt so it can run non-interactively from MDM/SCCM/Ansible.

It needs the same elevated privileges an interactive install would (SYSTEM on Windows, root on macOS/Linux): a scripted push job running as SYSTEM/root can drive it directly, but it can't be triggered remotely from an unprivileged context.

As SYSTEM on Windows it installs silently (no desktop to show progress on). On macOS it repairs only `.pkg` installs, which are root-owned; it refuses an app copied from the `.dmg`, so deploy the `.pkg` (macOS above) if you plan to repair from MDM. See swarmfile-doctor.

Where to go next

- Identity — set up SSO and SCIM before you push installers, so provisioned employees can sign in and start on first launch with no separate invite to send.
- Network Requirements — the firewall allowlist to have in place before rollout.
- Engine Config File — every `config.json` key, for self-hosted or otherwise customized deployments.
- Self-Hosted Seed Nodes — turning existing on-prem hardware into a warm cache tier as part of the same rollout.

Data Portability & Offboarding

Straight answers to the questions that usually come up during procurement: how do we get our data out, what happens if we cancel or downgrade, and what does uninstalling actually remove. Where something isn't automated or self-serve, it's stated as such rather than glossed over.

Getting your data out

Because a Swarmfile repository is a real mounted drive rather than a walled-off web UI, there's no special export step: copy files off the mount with Finder, Explorer, `cp`, `rsync` — whatever you'd normally use.

There's no throttle or block on bulk reads while your org is in good standing, and any method below works right up until the moment you cancel — which, as If you cancel or your payment fails explains, starts a countdown rather than leaving reads available forever. If you're leaving, export first. (There is also a WAN bandwidth throttle you can optionally schedule for office hours, but that's a setting you control for your own network's sake, not a vendor-imposed export limit.)

For a scripted or continuous bulk export, Branch Mirror (Pro/Enterprise) does exactly this: it continuously exports a repository branch into an S3-compatible bucket you control, as fully-hydrated real files at their real paths — not Swarmfile's internal format — a practical continuous export.

The honest caveats:

- It's configured per branch.
- It's asynchronous (changes land in your bucket within a few hours — about three on an active repository, about six while it is idle — not instantly).
- It tracks the branch continuously rather than capturing a point-in-time snapshot, so it complements a scheduled backup rather than replacing one.
- It's managed-tier only (end-to-end encrypted repositories are refused, since the hub can't decrypt them to hydrate readable files).
- It requires a Pro or Enterprise plan.

One thing worth knowing in your favor: an already-running mirror keeps syncing even through the grace period described below, when the mount and web app are otherwise locked out — set it up before you need it, not after. `swarmfile-migrate`, by contrast, only moves data *into* Swarmfile, not out. If you'd rather not stand up a mirror, "mount it and copy the files" is always available as the manual export mechanism — talk to us if you need help.

If you use Swarmfile as a **git-LFS server**, there's a third exit that's fully self-serve and needs no Swarmfile software at all:

- `git lfs fetch --all` pulls every LFS object into your local git-LFS store with stock git-LFS.
- `git lfs fsck` verifies each one.
- Repoint `.lfsconfig` at any other LFS server (GitHub, GitLab, an S3-backed server, self-hosted), then `git lfs push`.

Because your source lives in your own git host and only the binary objects are in Swarmfile, this gets those objects out with standard tooling and no vendor dependency — see Leaving Swarmfile in the git-LFS guide.

On each machine that pushed large objects, run `swarmfile-lfs uninstall` to remove the transfer-agent wiring from git config once you're done (committed `.lfsconfig` files and stored credentials are separate — the command tells you what it left alone).

If you cancel or your payment fails

Both paths lead to the same place: a **28-day grace period** during which the org's content is inaccessible, followed by permanent deletion. There is no indefinite post-cancellation read state — treat cancellation as a deletion deadline and get your data out first (see Getting your data out).

If you cancel deliberately (via the Stripe Customer Portal): the org leaves its paid subscription state (there's no free tier it falls back to — it doesn't become a Free-plan org), and for the first ~24 hours nothing else changes, a deliberate cushion so an accidental click can be undone by resubscribing. After that window the org is flagged as unsubscribed and the standard grace period begins, exactly as if a payment had failed:

- **A real 28-day grace period runs, tracked independently of Stripe's own retry schedule** — from the moment the cancellation is flagged, not shortened or extended by whatever dunning timeline Stripe itself runs.
- **Reads are blocked too, not just new uploads — everywhere, not only the mount.** The desktop mount, the web dashboard's own download/preview, and public share links you've handed out all stop serving that repository's content for the whole window, not only once the deadline nears. The dashboard and billing page themselves stay reachable so you can resubscribe. The one deliberate exception is Branch Mirror: an already-running mirror keeps exporting through the grace period, covered above.
- **We email the org owner as the window closes** (day 1, 7, 21, and the 27th day) so there's real warning before anything happens, not a surprise.
- **If the 28 days pass with nothing resolved, the org's data is permanently deleted — and so is the owner's account, if they don't belong to any other org.** This includes every repository, file, and version in the org, the underlying storage, and the org itself; it also includes the owner's Swarmfile login if that org was their only one (an owner who's a member of another org keeps their account either way). This is genuinely irreversible.

If a payment fails and is never resolved (a declined renewal, an expired card, and Stripe's own retries are exhausted): the same 28-day window starts from the moment the payment first failed, with the same blocked reads, the same reminder emails, and the same permanent deletion at the end.

Resubscribing at any point before the 28 days are up cancels the countdown entirely and restores access, with nothing lost. If a payment is failing and you want your data out, do it before the window closes — and if you want to leave without a deadline, either export first and then cancel, or ask us to delete the org outright (below) once your export is done.

If you downgrade over your new plan's limit

Downgrading (Pro to Starter, for instance) when your stored data exceeds the smaller plan's included allowance doesn't delete or lock anything either. Data above the smaller plan's allowance is billed as overage up to your spend cap, which defaults to the new plan's seat charge. Once overage reaches that cap, new uploads that would add storage pause until you raise the cap, add storage packs or free space; on a large excess that pause comes well before the 3x hard ceiling. Existing files stay fully readable throughout, and reads, deletes and renames keep working.

Deleting an organization entirely

If you want an org torn down on your own schedule — not because of a lapsed payment — the org owner can do it from the web dashboard: **Settings → Organization → Delete organization**. It asks you to retype the org slug, and to enter your password if your account has one; it must be done from an interactive sign-in — an API key or personal access token can't perform it.

It then cancels any active subscription, revokes the org's API keys, disconnects any connected GitHub App installation, and deletes the org, its repositories, its files, and its stored data. The purge writes durable audit rows that outlive it, so the org deletion remains auditable after the data is gone; if the credential revoke stalls, the purge reports `credential_revoke_pending` rather than success.

It is immediate and irreversible, so make sure your export is complete first. Admins and members won't see the control; it's owner-only. If you'd rather have us handle the deletion, contact us and we'll do it directly.

The automatic 28-day path described above is the one exception: that deletion is triggered by billing status, not a support request, for the reasons explained there.

Uninstalling the desktop client

Uninstalling a machine never touches your account or org data — that all lives on the hub, not the machine you're uninstalling from. What the uninstaller does *not* do, on any platform, is clear that machine's local cache (the block cache, upload queue, and local metadata database) — that's left in place, in case you're reinstalling rather than fully decommissioning the machine.

The one real risk: Swarmfile's writes are designed to return instantly and upload in the background — a save is only visible to your teammates once it's actually finished uploading to the hub, which for a large file can take a while after the save itself completes locally. If you uninstall (or, more precisely, wipe that machine's local Swarmfile cache) while an upload is still queued and hasn't finished confirming with the hub, that specific save is lost — it never makes it to the hub, and nobody else ever sees it.

Reinstalling on top of the same, un-wiped cache resumes the queue and finishes the upload normally; wiping the cache and starting fresh does not. In practice: before decommissioning a machine, check the Desktop App for pending uploads and let them finish first.

Self-hosted seed nodes as an exit ramp

A self-hosted seed node fetches and mirrors a repository's data through hardware you own — keeping a warm local cache on it — which sounds like a natural answer to "what if the hosted service goes away."

For the standard managed (encrypted) tier — the default, and what most repositories use — it's a more qualified answer than that framing suggests: what the seed node holds is ciphertext, and reading it depends on a decryption key fetched live from the hosted hub. There's no offline fallback for that key.

So a seed node is a genuine speed and reliability win for your office day to day — most reads served locally, less dependence on your internet connection — but it isn't an independently readable backup you could fall back to if the hosted service were ever permanently unreachable.

If true offline independence — not just a warm local cache, but a fully self-contained deployment with no dependency on our hosted infrastructure at all — is a hard requirement, that's the Enterprise self-hosted control plane: the same code running entirely on infrastructure you control, available on Enterprise (a fully air-gapped deployment is on the roadmap) — see Deployment Topologies and talk to us about what it would involve for your environment.

Where to go next

- Billing & Plans — how plan switching, seats, and overage actually work.
- Self-Hosted Seed Nodes — running your own warm cache tier.
- Security Architecture — the full key-custody and threat-model detail behind the encryption note above.

Branch Mirror

Branch mirror continuously exports one repository branch into an S3-compatible bucket you control — as **real files at their real paths**, not Swarmfile's internal content-addressed block format. It's a secondary, one-directional copy: our storage platform stays the primary, authoritative home for your data, and the mirror is a fully-hydrated, browsable duplicate that lives in your own bucket for backup, disaster-recovery, or downstream-pipeline use.

Available on **Pro and Enterprise** (the `s3_mirror` entitlement), configured per branch from the dashboard.

How it's different from the alternatives

- **Not `swarmfile-migrate`.** That tool is a one-shot import that moves data *into* Swarmfile (from a NAS, S3 bucket, or file list). A branch mirror runs the other direction and keeps running — a continuous export *out* to a bucket you own, not a single copy-in.
- **Not Dedicated Storage.** Dedicated Storage is about isolating your bytes *inside our own infrastructure* — a bucket that's physically yours but still one we create and operate. A branch mirror puts a readable copy in *your* infrastructure entirely, on a provider and account you control.
- **Not bring-your-own-storage.** Bring-your-own-storage makes your bucket the *primary* home where the live blocks actually live. A branch mirror is strictly a read-only export copy — the authoritative data still lives on our platform, and the mirror just tracks it.

The key thing a mirror gives you that the block store can't: the files land in your bucket at their natural paths (`repositoryroot/scenes/shot_010.exr`, not an opaque content hash), directly readable by anything that speaks S3 — no Swarmfile client, no decryption step, no reassembly.

Where to configure it

Open the repository in the dashboard and go to its **Mirror** tab. Configuration is **per branch** — each branch of a repository mirrors independently (or not at all), so you can export just `main` while leaving working branches unmirrored.

The tab is **owner/admin-gated** and requires a Pro or Enterprise plan. On a plan without the `s3_mirror` entitlement the tab explains the gate; an existing mirror keeps its last-synced state and can still be turned off, but its configuration can't be changed until the plan includes mirroring again.

Configuration fields

Field	Notes
Endpoint URL	Your provider's S3-compatible endpoint. Not needed for AWS S3 itself if you use the region-default endpoint; required for Wasabi, Backblaze B2, MinIO, and most others.
Region	The bucket's region.
Bucket name	The target bucket. It must already exist — Swarmfile writes into it, it doesn't create it.
Key prefix	Optional. A sub-path inside the bucket to confine the mirror to (e.g. `swarmfile/`). Leave blank to write at the bucket root.
Access key ID	The S3 access key ID for a credential with write access to the bucket.
Secret access key	Stored **encrypted at rest** (AES-256) and **never returned** by the dashboard. On an existing mirror, leave it blank to keep the stored credential — you only re-enter it to change it.

The credential is validated with a real round-trip against your bucket at save time, so a bad endpoint, region, bucket name, or key is caught immediately in the settings screen rather than silently on the first sync.

Behavior and limits

- **Full-fidelity tracking.** The mirror reflects the branch as it changes: new and modified files are re-exported, and **deletes, restores, and renames propagate** to the target bucket. What's in the mirror tracks what's on the branch.
- **Asynchronous, up to a few hours of lag.** Syncing is driven by a periodic maintenance scan — about every three hours while the repository is active, backing off to about six hours while it is idle. So a change can take a few hours to appear in your bucket (up to ~3 h on an active repository, ~6 h on an idle one). This is not a live, write-through copy.
- **Fire-and-forget.** There's no delivery guarantee, no built-in retry-until-success alerting, and no per-file receipt. A file that fails to sync on one pass is simply left unconfirmed and picked up again on the next scan's fresh diff — the sync self-heals over time rather than escalating a failure to you.
- **One oversized file skips; a bad credential fails the whole sync.** If a single file trips a size cap (or hits a transient error), that one file is left unsynced and retried next tick — the rest of the batch still goes through. A broken or rotated-away credential, by contrast, fails every file, so the whole sync fails until you fix the credential.
- **Status is visible in the tab.** The Mirror tab shows the current state — **Never synced**, **Syncing**, **Synced**, or **Sync failed** — along with the last sync time and, on failure, the last error.
- **Turning it off leaves the bucket alone.** Disabling a mirror stops future syncs but never deletes what's already been written — that data is yours in your bucket.

End-to-end encrypted repositories are not eligible

A mirror is **refused at configuration time for end-to-end (customer-managed) encrypted repositories**. The hub never holds an E2E repository's key, so it structurally cannot decrypt the blocks to hydrate readable files — there's nothing it could write to your bucket but ciphertext. Mirroring is a managed-tier feature; on an E2E repository the save is rejected with an explanation rather than producing an unusable export.

Data residency is your bucket's responsibility

Once a file is written to your bucket, where it physically lives is governed by *your* provider, account, and region — not by Swarmfile's data residency pinning. Swarmfile can't verify or enforce the residency of an arbitrary customer-supplied endpoint, so for an org with a **pinned jurisdiction** — or for any individual **repository pinned to its own region** — a branch mirror is refused outright rather than silently allowed to route data outside that jurisdiction. If you're using a mirror, choosing a bucket whose region matches your own residency commitments is on you.

Where to go next

- Bring-your-own-storage — making your bucket the *primary* home for live blocks (Enterprise).
- Dedicated Storage Isolation — physical isolation inside our own infrastructure.
- Data Portability & Offboarding — the fuller picture on getting your data out.

Filesystem Compatibility & Conformance

Swarmfile mounts as a real drive — `~/Swarmfile` on macOS and Linux, a drive letter on Windows — through FUSE/FUSE-T and WinFsp, not through a synced local folder. The exact location is per-machine and can be configured (the engine's `mount_point`; Windows picks its drive letter at runtime), so read yours from the Desktop App's Status pill rather than hard-coding a path. That distinction matters for exactly the kind of file your native apps care about: does a rename actually rename, does a lock actually stop a second writer, does opening a 200 GB file touch the whole thing or just the bytes you read.

This page states plainly what works today, per platform, and what's a known gap.

What's verified

Most rows below are exercised by automated write-semantics tests that drive the *mounted filesystem* the way an application does — not the internal storage code directly — on a live mount per platform. A few rows (adapter capabilities and platform-specific behavior) are covered by targeted tests instead; the Notes column says when.

Capability	macOS	Linux	Windows	Notes
Create / read / write	Verified	Verified	Verified	
Delete (file & directory)	Verified	Verified	Verified	On Windows, verified via Win32 `RemoveDirectory`, `cmd rmdir /s` and .NET's recursive `Directory.Delete`, including on empty folders. PowerShell's `Remove-Item -Recurse` fails through the mount (known limitation — use one of the above instead)
Rename — same directory	Verified	Verified	Verified	
Rename — cross-directory (move)	Asserted	Asserted	Verified	Entry identity, history, comments, and locks move with the file — this is not a copy-then-delete. Windows verified on a live mount; macOS/Linux covered by the conformance suite
Exclusive create (`O_EXCL` / Win32 `CREATE_NEW`)	Verified	Verified	Verified	A create that races an existing name is refused (`EEXIST` / `STATUS_OBJECT_NAME_COLLISION`), enforced at our layer — so it holds even offline, not just when the OS routes the open
Overwrite / truncate an existing file	Verified	Verified	Verified	Content-checksummed, not just size-checked
Directory listing at scale	Verified	Verified	Verified	Wildcard/exact-name filters are verified on Windows only — POSIX shells and apps glob client-side over a plain readdir, so there's no equivalent operation to assert there
Case handling	Case-preserving	Case-preserving	Case-insensitive	Matches each platform's native convention. Names also resolve across Unicode forms on every platform: `café.txt` created as NFC opens when asked for the NFD spelling, an ASCII-case-only difference resolves, and the Turkish dotted/dotless I family (`İ` / `ı` vs `i`) is treated as one name — the store keeps the spelling you created with. Verified live on Windows in both request directions and unit-tested cross-platform; the WinFsp boundary reports the requested spelling when the stored one differs by more than case, which is what lets the open succeed
Free-space reporting (`statfs`)	Verified	Verified	Verified	
`fsync` durability	Verified	Verified	Verified	A write is not reported durable until it actually is; on Windows this is the `FlushFileBuffers` path
Extended attributes (get/set/list/remove)	Local-only, via macOS's `._` files (see below)	Implemented	Not implemented	Windows EA specifically — see below. Not the same thing as alternate data streams, the row below. On Linux the adapter implements all four operations; nothing on a Linux mount creates them the way macOS does, so end-to-end use isn't exercised
Alternate data streams — create/read/write/enumerate/delete	N/A	N/A	Verified, incl. sync between two clients	Windows-only concept (`file.ext:streamname`). A stream also survives a copy through the mount rather than being silently dropped, the way most non-native sync tools drop it. The `Zone.Identifier` stream (Windows' Mark of the Web) stays on the machine that wrote it and never syncs — see Sync Exclusions. Every other stream's content travels the same hub upload/commit pipeline as file content. Verified on real Windows mounts: these operations, and sync between two clients through the hub. A stream written or changed on one client reads back identical on the other, a stream deletion reaches the other client, and a `Zone.Identifier` never leaves the client that wrote it
Alternate data streams — rename	N/A	N/A	Not supported (see below)	
Symlinks — create	Verified	Verified	Not supported (see below)	
Symlinks — read / follow / list / delete	Verified	Verified	Read, follow & list	On Windows, `readlink` (reparse-point resolution), following a link to its target, and listing it as a reparse point are all verified; delete-through-the-mount isn't verified on Windows
Executable bit (`chmod +x` / `-x`)	Verified	Verified	Preserved (no executable bit)	Any `x` bit sets the repository's per-file executable flag; the drive reports `0755` / `0644`. A create with mode `0755` (including `git checkout` of a `100755` file) comes out executable. Windows has no mode bit: edits and safe-saves there keep the flag, and `swarmfile chmod` sets it from any platform
Byte-range locking (API)	Verified	Verified	Verified	Hub-enforced acquire/release/overlap-detection, driven over the engine control socket — the same primitive a native worksharing plugin (e.g. for Revit) would call. Release is always an explicit unlock call over that socket; on Windows in particular, a lock is not auto-released when the OS closes the file handle, so a caller must unlock explicitly. Revit/SolidWorks worksharing itself needs no plugin on Windows — Swarmfile honors the native OS file lock those apps already take and turns it into a hub-enforced claim across machines; on macOS/Linux only cooperative locks apply. A deployment that needs an open refused whenever the hub can't be reached can enable that (see Environment Variables). This byte-range API is the lower-level primitive for finer, element-level integrations beyond that. See Swarmfile vs. LucidLink for how the underlying lock guarantee compares
Folder/file ACL enforcement	Verified	Verified	Verified, incl. Explorer-visible permissions	Enforced hub-side for every client; only Windows additionally enforces and presents ACLs at the mount itself (Explorer-visible DACL). On macOS and Linux, Finder and `ls` show the mode bits from `getattr`, not the repository's ACLs

Declared gaps

Each of these is a documented limitation:

- **Hard links.** Not supported on any platform. Our metadata model is one entry = one path = one content id; a hard link needs a content-identity indirection the schema doesn't have.
- **Symlink creation on Windows.** Windows mounts read and follow symlinks correctly but refuse to create one. Creating a Windows symlink as an unprivileged process requires Developer Mode or a specific privilege — support would work for some users and silently fail for others on the same drive, so we refuse

consistently instead.

- **Renaming an alternate data stream itself.** Renaming the base file that carries a stream works normally, and the stream travels with it. Renaming just the stream — `file.txt:secret` to `file.txt:hidden` — is out of scope and refused outright, not silently mishandled: we won't rename the base file when you only asked to rename one of its streams, and we won't leave the stream qualifier in a request that no path would ever resolve to.
- **Extended attributes on Windows.** Not implemented — a deliberate scope decision, separate from alternate data streams (see the verified-capabilities table above, which streams *are* supported).
- **A save the hub refuses reaches the app as `EIO`.** When a save is refused (a file held elsewhere, a plan or policy refusal, a root-create restriction, quarantine), the application only sees a generic I/O error; the reason is written under **Why did my save fail?** in the tray and printed by `swarmfile sync stuck`. The errno is deliberately generic, so read the reason there rather than from the app's error dialog.
- **`fallocate` / `copy_file_range` / `lseek(SEEK_HOLE)`.** Not implemented on the POSIX adapters.
- **`fcntl()` byte-range advisory locks are kernel-local only.** They don't coordinate across machines — that's a different mechanism from the hub-enforced byte-range locking API above. An app that relies on plain POSIX advisory locks for multi-writer coordination (rather than calling into the lock API) gets locking that looks like it works but only protects against other processes on the same machine.
- **Windows: a memory-mapped view can outlive the handle for reads, not for writes.** Reading through a memory-mapped view works, including when an application closes the file handle before reading the map — `git` does exactly that with `.git/config`, which is why `git init` / `commit` / `status` work on the mount. Writing through a mapping is only served while a file handle is open: an application that maps a file **read-write**, closes the handle, and then writes through the view can have those write-backs fail. Keep the handle open until the mapping is unmapped, or write through the handle. Read-only mappings (the common case, git included) are unaffected.
- **A file isn't readable the instant it's written — for about a second, on Windows.** The engine itself publishes the new content *synchronously* as the flush completes (the local metadata row, size included, is updated before close returns), so a script that opens the file through a fresh handle gets the new bytes on macOS and Linux. Windows can still report the old attributes (0 bytes for a new file, the previous size for an overwrite) for up to its **1-second Cache Manager revalidation timeout**, because the kernel caches file info between revalidations. It matters for any workflow that writes a file and immediately reads it back — a save-then-verify step, a render job that re-opens its own output, or an app that reloads the file right after saving. A person clicking around is usually slower than the one-second window, but not always: reopening a just-saved file within that second can still see the old size. If you need certainty, wait on the engine rather than the clock: `swarmfile uploads --wait` blocks until the upload queue is empty, and `swarmfile status` splits the rest — `pending uploads` (blocks still going up), `commits pending` (bytes up, commit not landed; a plan-held commit shows under `quota-held commits`). (A plain `sync` forces pending writes to flush locally but does not wait for the hub upload and commit, so use the counters, not `sync`, when that distinction matters.)
- **`access()` / `faccessat()` aren't enforced locally.** Permission is decided hub-side by the ACL system, not by local file mode bits, so a pre-flight `access()` check reports success even for a write the hub will later refuse.
- **A read can WAIT, for a file that is still uploading.** A file that has never finished uploading can be opened — it reports the size it is going to be, and reads work. A read into a range that has not arrived yet waits for it, bounded at 15 s by default, and returns a *retryable* error — `EAGAIN` on macOS and Linux, `STATUS_IO_TIMEOUT` on Windows — rather than an I/O error if the wait times out, because an application that sees an I/O error part way through a file usually decides the file is damaged and discards your document. A stall is the correct answer for a player; a corrupt-file error is not. It only ever applies to a file with no committed version yet, so nothing you are already reading changes behavior.
- **macOS: `ditto` works.** Plain `cp` / `cp -R`, `rsync`, Finder, and the installers and build scripts that reach for `ditto` all work.
- **macOS extended attributes are local-only.** The drive is mounted without NFS named attributes, so macOS keeps Finder tags, comments, and the like the way it does on any such volume: in an AppleDouble `._name` file beside the original. Swarmfile keeps those `._` files, and `.DS_Store`, on the machine that wrote them and **never syncs** them (see Sync Exclusions) — `cp -p`, `ditto`, and `xattr` work with them on that machine, but a teammate (or that machine after a fresh install) won't see them. Windows alternate data streams are the exception and do travel (except the `Zone.Identifier` Mark of the Web, which stays local); POSIX xattrs crossing machines would be a hub-side format addition, not a mount fix.

How we verify this

Each platform has its own conformance suite driving the live mount, and a shared manifest keeps the three platforms' coverage aligned — a capability missing on one platform is caught rather than left for you to find. Known gaps are listed above.

How this compares to typical cloud storage clients

Most general-purpose cloud storage clients — the kind built primarily for syncing documents and photos — are sync engines wearing a filesystem's clothing: a background process reconciling a local folder against the cloud, not a filesystem driver with defined semantics for the operations above. That's a fine model for a folder of PDFs. It tends to fall over for exactly the workloads Swarmfile targets: whole-file download before an app can open anything, no cross-process byte-range locking, and "two people edited it" resolved by keeping both files rather than by a lock.

This isn't a hypothetical concern — it shows up in vendor documentation for the professional tools this product is built for.

Autodesk's own support article on using cloud-synced services with Revit files states that file-based worksharing isn't supported on common cloud-sync storage, and names corruption of the central model and lost work as the consequence.

Esri's knowledge base similarly documents common problems running ArcGIS Pro against cloud storage services.

If you're evaluating any filesystem — including this one — for this kind of workload, the questions worth asking are the ones this page answers: is locking enforced per byte range or just per whole file, is it enforced on every platform your team actually uses, and is any of it independently verified rather than asserted.

Supported Platforms & System Requirements

Swarmfile ships desktop installers for macOS, Windows, and Linux, plus command-line tools with each. This page lists what runs where, what each installer includes, and the limits worth knowing before you plan a deployment.

At a glance

Platform	Architectures	Installer	Minimum OS
macOS	Universal (Apple Silicon + Intel)	`Swarmfile.pkg` (full install), `Swarmfile.dmg` (app only — no service or PATH setup)	macOS 11.0
Windows	x64, ARM64	`Swarmfile-Setup.exe` / `Swarmfile-x64-Setup.exe` (bundled), `Swarmfile.msi` / `Swarmfile-x64.msi` (bare, for IT)	Windows 10 or Windows 11
Linux	x86_64 only	`swarmfile-amd64.deb`	Ubuntu 22.04+ or an equivalent current Debian-based distro

Download them from swarmfile.com/downloads, or ask for the stable `releases.swarmfile.com/latest/...` URLs for a scripted install.

macOS

- **One universal build** for Apple Silicon and Intel; **macOS 11.0 or later**, enforced by the installer.
- Use the `.pkg`: it installs the Desktop App, the engine as a LaunchAgent, the macOS mount helper, and symlinks for every CLI tool into `/usr/local/bin`.
- The `.dmg` installs the Desktop App, whose bundle carries the engine, the runner, and the CLI binaries — but it doesn't set up the engine's LaunchAgent service, the mount helper, or the `/usr/local/bin` symlinks, so the engine isn't running as a service and the CLI isn't on `PATH`. On first launch the tray opens **Diagnostics** with the Reinstall action so one click puts them in place.
- Mounting is **kext-free**: Swarmfile bundles FUSE-T, so there is no kernel extension, no Recovery Mode step, and no system-extension approval on Apple Silicon.
- **macOS 15+** additionally requires **Local Network** permission for LAN discovery; the app asks for it, and headless engines running outside the app bundle need to inherit it (or run as a LaunchDaemon). See Network Requirements.
- Installers are signed with a Developer ID and notarized/stapled.

Windows

- **x64 and ARM64** builds; Windows 10 or Windows 11. Both install **per-machine** into `C:\Program Files\Swarmfile` and add that folder to the machine `PATH`, so the CLI is available to every account.
- **Use the bundled `Setup.exe` for people**: it chains WinFsp, the Visual C++ Redistributable, and the WebView2 runtime, skipping whatever is already present. Windows Server/VDI images usually need those three staged by your own tooling first.
- **Use the bare `.msi` for scripted rollouts** (`msiexec /i Swarmfile-x64.msi /qn /norestart`): it installs only the app and expects those prerequisites to be present. It checks for WinFsp and the VC++ Redistributable during install and refuses with a message naming whichever is missing, rather than failing later with a side-by-side error; WebView2 isn't part of that gate, so stage it too. Repair and upgrade are never blocked by these checks. See Deploying to Your Team.
- One Windows-specific behavior worth knowing: **symlink creation is refused on the mount.** Windows needs Developer Mode (or a specific privilege) for an unprivileged process to create symlinks, so Swarmfile refuses consistently instead of failing for some users; reading, following, and listing symlinks all work.
- The `Setup.exe` and `.msi` installers, the desktop `-bin.zip` / `-app.zip` payloads (their `.exe` / `.dll` contents, not the archive itself), and the standalone `swarmfile-doctor` download are Authenticode-signed in the desktop release pipeline. The Windows tool `.zip` downloads are signed by a separate signing job that is fail-closed as well: if it fails, the tool release is blocked and no unsigned archive is published. The desktop release likewise refuses publication instead of falling back. Windows still shows the SmartScreen prompt on first run until the signing identity builds reputation — see Getting Started.
- The bundled **WinFsp is pinned to a tested 2.2 Beta build** (the exact pin is carried in the Windows installer's arch map); the pin moves to 2.2 stable when it ships.

Linux

- **x86_64 only** for the Desktop App from swarmfile.com. (The self-hosted edition also ships an arm64 `.deb` in its bundle.) The `.deb` requires `libwebkit2gtk-4.1-0`, `libgtk-3-0`, and `libfuse2`; install it with `apt` and the dependencies resolve normally.
- The downloaded `.deb` is **checksum-only by default**: verify it against the published `.sha256` / `SHA256SUMS`. When a release also publishes the GPG-signed apt repository, add it as an apt source and install or upgrade through it so apt verifies the package against the repository key; the repository ships its public key and fingerprint.
- It installs **per-user systemd units** (`swarmfile-engine`, `swarmfile-tray`) enabled globally, so new logins start them with no extra action; a user already logged in needs to log out and back in (or start the units) for them to take effect. The Desktop App unit is conditioned on a graphical session, so it does nothing on a headless box.
- The `.deb` **does not touch your firewall**; on a host with a default-deny inbound policy, allow UDP from the local subnet — see Linux firewalls.
- **`.rpm` isn't published.** If you run RHEL/Fedora/OpenSUSE, talk to us about a supported `.rpm` path.

Hardware and disk

There is no formal RAM or CPU minimum: the engine is a native service, and the mount is a filesystem driver, both modest by desktop standards. The numbers that matter are disk-related:

- **Local cache: 10 GiB by default** (`cache_max_bytes`). Files you touch are cached locally and evicted as the cache fills; it is not a copy of the repository. Adjust it in the Desktop App's settings or `swarmfile cache set <GiB>`.

- **The mount reports at least 1 GiB of free space** to applications even when the cache is full but evictable, so a "disk full" error from your app reflects local disk pressure, not the repository's size.
- **A seed/NAS node** is different: it fetches whole repositories (mirroring them to cloud storage) and keeps them in a warm cache with its own budget. Size `SWARMFILE_CACHE_MAX_BYTES` for the working set you want resident — the sample seed config suggests a **100 GiB cache as a starting point** and raising it to 1 TiB or more on a capable NAS; an evicted block re-fetches on the next read.
- **Repository metadata**, not file bytes, has a per-repository limit of roughly **ten million files and versions combined** — very hard to reach unless you keep every version of a very large tree forever. See Repository storage limit.

File and object size limits

Content is chunked (FastCDC, 256 KiB–4 MiB chunks) and manifests are segmented once they grow past ~40,000 chunks, so a **file on the mounted drive has no practical size cap** — multi-terabyte files work, and the manifest structure extends rather than truncating. Two specific ceilings are worth planning around:

- **git-LFS objects: 256 GiB maximum.** A single LFS object beyond that is refused. On an unencrypted repository, stock `git lfs push` works for objects up to about 5 GiB and larger pushes need the transfer agent; on a managed repository the transfer agent is required for every upload (stock direct uploads are refused) — see git-LFS.
- **Hub-mediated downloads** assemble up to 900 manifest segments per read; segmented manifests keep large files readable. A very large *native* `git-lfs` download from the hosted hub may first be paced while the hub prepares it (the client's retry setting carries it through); the self-hosted community server assembles it directly instead. The mount reads it either way.

Platform capability differences

The filesystem semantics are the same everywhere for ordinary work (create, read, write, rename, delete, locking). A short list differs by platform:

Capability	macOS	Linux	Windows
Symlink creation	Yes	Yes	Refused (reading/listing works)
Extended attributes	Kept locally, not synced (`cp -p` / `ditto` work)	Adapter implemented (unit-tested; local-only)	N/A — use streams (streams do sync)
Alternate data streams	N/A	N/A	Yes (rename of a stream itself is refused)
Filtered/wildcard directory listing	App globs client-side	App globs client-side	Native filters verified
ACLs at the mount	Hub-enforced; `ls` shows mode bits	Hub-enforced; `ls` shows mode bits	Hub-enforced **and** projected as Explorer DACLs
`ditto`	Works	N/A	N/A
PowerShell `Remove-Item -Recurse` on a directory	N/A	N/A	Not supported — the cmdlet fails through the mount; use `cmd rmdir /s` , .NET's `Directory.Delete` , or Explorer

The full per-operation table, including the declared gaps (hard links, `fcntl` advisory locks, sparse files), is in Filesystem Compatibility & Conformance.

Where to go next

- Release Channels & Updates — how installs update and reinstall.
- Deploying to Your Team — silent install and MDM/GPO rollout.
- Network Requirements — the firewall allowlist before rollout.

Release Channels & Updates

Swarmfile installs update themselves, but never silently and never without you saying so. (The self-hosted edition is the exception: it never updates itself; you upgrade by installing the next bundle.) This page covers where installers come from, how updates are delivered and verified, and how to reinstall a machine cleanly.

Where installs come from

What	Host
Installers	`releases.swarmfile.com`
Update manifest	`updates.swarmfile.com/stable/latest.json`
Dashboard / web app	`swarmfile.com`
Hub / identity	`hub.swarmfile.com` , `id.swarmfile.com`
git-LFS agent installer	`get.swarmfile.com/lfs`

A machine's hosts are baked into its install, not compiled into the binary: packaging writes the hub, identity, site, and update endpoints into the bundled `config.json` , so an install keeps the endpoints it was packaged with. See Engine Config File.

Installers are available two ways under `releases.swarmfile.com` :

- **Stable aliases** — `releases.swarmfile.com/latest/Swarmfile.pkg` , `.../latest/Swarmfile.dmg` , `.../latest/swarmfile-amd64.deb` , `.../latest/Swarmfile-Setup.exe` (Windows ARM64), `.../latest/Swarmfile-x64-Setup.exe` , and the bare `.msi` files. These are what the download page's links resolve to (the page links through the hub, which redirects here).
- **Versioned paths** — `releases.swarmfile.com/v<version>/Swarmfile-<version>.pkg` and so on, so a scripted install can pin an exact build whose SHA-256 stays valid. Each artifact ships with a `.sha256` file and there is a combined `SHA256SUMS` per release. Homebrew, winget, and Chocolatey packages point at the same signed artifacts.

Use the **stable aliases** for people and the **versioned paths** for automated mass installs.

How updates work

Updates live in the Desktop App, not the engine:

- It checks once at startup and then **every 45 minutes** while running. Repeated failures back off, up to about every six hours.
- A found update **only** raises a notification: a pill in the header and a card in **Diagnostics**. Swarmfile **never installs an update on its own** — **Install now** is the only path, and that is deliberate: an unattended restart while a save or render is running is worse than being a few days behind.
- If an update sits uninstalled for **a week**, the Desktop App sends **one** desktop notification naming the version — still informational, still install-on-your-command. (A restart re-raises the pill, so the nudge is for the tray that stays running for weeks.) You can also press the manual **Check for updates** action in Diagnostics at any time.

What happens when you press **Install now** differs by platform:

- **macOS and Windows** — the app downloads the signed package and installs it, then restarts itself (and its engine). On Windows an unelevated app starts the **Swarmfile Updater** helper (an on-demand, never-auto-triggering elevated helper) which verifies the package signature and applies it.
- **Linux** — the app cannot self-install (it runs as your user, and package installation needs root). It shows a notification pointing at the `.deb` for you to install; there is no silent privilege escalation.

There is no beta, nightly, or insider channel. Every release publishes to the single `stable` channel.

Update integrity

Updates are verified before they are applied:

- The update manifest's artifacts are signed with a repository **Ed25519 (minisign)** key, and each release channel is checked only with the key configured for it; staging uses the production key until a separate staging key is configured. Each signature also names the exact file it covers, so a genuine signature from another release does not verify. The public keys are compiled into the app, the Windows updater helper and `swarmfile-doctor` , so a compromised update server cannot hand you a package that verifies — the endpoint can be moved, the trusted key cannot.
- The **app itself** refuses an update whose signature does not verify, and refuses a manifest that is not strictly newer than the installed version.
- **macOS** installers are Developer ID–signed and notarized/stapled. **Windows** installers (`Setup.exe` and the bare `.msi`), the desktop `-bin.zip` / `-app.zip` payloads (the `.exe` / `.dll` files inside them, not the archive itself), and the standalone `swarmfile-doctor` download are Authenticode-signed in the desktop release pipeline, which refuses publication without the macOS and Windows signing evidence. The standalone Windows tool archives are signed by a separate signing job that is fail-closed too: if it fails, the tool release is blocked and no unsigned archive is published. The desktop release likewise refuses publication without its zip-signing and installer-signing markers. **Linux** `.deb` s are checksum-only by default: verify them against the published `.sha256` / `SHA256SUMS` and rely on the updater's own signature when updating. When a release also publishes the GPG-signed apt repository, installing or upgrading through that repository makes apt verify the package against the repository key. On Windows, the SmartScreen prompt is expected on first run until the signing identity builds reputation — see Getting Started.
- Installing an **older** version over a newer one is not supported: the Windows MSI refuses downgrades, and the updater refuses non-newer manifests. To move a machine backward, remove the app and install the exact older package from its versioned path.

Version numbering

Versions are `0.<minor>.<patch>`; the current line is `0.3.x`, and the newest published version is whatever the stable release manifest reports (`updates.swarmfile.com/stable/latest.json`). The version is embedded in the installer name and in every versioned URL, and release publishing is monotonic — an older version can never overwrite the current `latest` manifest.

The engine, CLI tools, and Desktop App in one install share the same version; if they ever drift — an update that leaves a background service pointing at an older engine binary, say — the app raises a version-mismatch banner and offers the right remedy: restart or install for a plain mismatch, and repair/reinstall when the service itself points at the old binary, where a restart would just re-run it.

Reinstalling or repairing

When an install is unhealthy (a broken mount helper, a missing CLI tool, a half-applied update), reinstall rather than removing things by hand:

- **Desktop App: Diagnostics → Reinstall...** (Windows, macOS and Linux). On Windows it downloads the `.msi` the update channel publishes, checks its release signature before running it, and reinstalls the installed version in place (every file, registry entry and shortcut is rewritten), the same way `swarmfile-doctor --repair` does; a newer version goes through the updater helper first. On macOS and Linux it downloads the full installer (the `.pkg` or the `.deb`) and opens it in the system installer for you to finish. Linux software centers usually treat a `.deb` of the installed version as already installed and offer no reinstall; to rewrite the same version, run `sudo swarmfile-doctor --repair`.
- **Command line, headless, or scripted:** `swarmfile-doctor --repair` (add `--yes` to skip the confirmation prompt; a script with no terminal needs it). It reads the update channel from the installed desktop app's own configuration (only Swarmfile's own update channel for that install), downloads the **full installer** for the platform (the `.pkg`, the `.msi`, or the amd64 `.deb`), verifies it before running it (the release signature published in the update channel for the `.msi` and `.deb`; the published SHA-256 plus Swarmfile's own Developer ID Installer signature and Gatekeeper for the `.pkg`), and runs it (`installer`, `msiexec`, or `dpkg -i`). It needs the same privileges a normal install does (`sudo` on macOS and Linux, administrator approval on Windows); it refuses when the installed app's configuration isn't root-owned, so on an app copied from the `.dmg` (owned by whoever copied it) download and run the `.pkg` from Downloads instead. On a machine without the desktop app it refuses. See `swarmfile-doctor --repair`.

On Windows, an update also refuses to run when the install directory is writable by non-administrators — the elevated helper stages and swaps files there, so a location any user can write is not a safe update target. If you hit that, reinstall to a protected location such as `Program Files`.

A repair **keeps your local state**: cached content, the queued-upload database, config, and sign-in token all live outside the application bundle (see Uninstall & Clean Reinstall for the exact paths). It is the supported way to move a `.dmg`-only macOS install onto the `.pkg` layout that installs the engine service and puts the CLI on `PATH`.

`--repair` installs the version the update channel currently publishes. If that is older than the installed version it refuses (`--allow-downgrade` overrides this on macOS and Linux; Windows never downgrades). If the channel can't be read, or the download doesn't verify, it stops without installing anything.

Where to go next

- Supported Platforms & System Requirements — what each installer contains.
- Network Requirements — the hosts above in firewall-allowlist form.
- Deploying to Your Team — mass deployment, silent install, and repair at scale.

Diagnostics & Error Reference

When something goes wrong, Swarmfile names the problem in one of a few places. This page maps the surfaces — doctor checks, status counters, health endpoints, and error codes — to plain meanings and next steps.

Start here

Surface	What it is	How to reach it
Desktop App → Settings → Diagnostics	Live checks, report file, **Repair drive**, **Reinstall...**, update check	The tray's settings
Status pill (header)	Mount location, sync state, the hub's reason for any pause	Click it
`swarmfile status`	The same snapshot as JSON or prose: counters, peers, bandwidth	CLI
`swarmfile-doctor`	Standalone probe suite that runs with no engine and no sign-in; engine-only checks are reported as a pass carrying `skipped (requires the engine)` when no engine is running	CLI
`swarmfile doctor`	Runs the same probes through a live engine	CLI
Hub error codes	Machine-readable reasons on API refusals (bring-your-own scripts)	JSON bodies

`swarmfile-doctor` returns a verdict, and writes its JSON report to a file when you pass `--report <path>`:

```
swarmfile-doctor --json --report doctor-report.json   # full detail
sudo swarmfile-doctor --repair --yes                   # reinstall the desktop app (keeps your queue)
```

Exit codes: `0` everything passed, `1` at least one failure, `2` warnings only. No report file is written without `--report`.

There is one `swarmfile-doctor`. It ships on `PATH` with every desktop install and, for machines without the desktop app (a hub or a self-hosted community server checked from a server or a build agent), as a standalone download (`curl -fsSL https://get.swarmfile.com/doctor | sh`); its source is open, in the `swarmfile-doctor` public repository (built on `swarmfile-lib`).

On its own it runs the engine-free checks; when a `swarmfile-engine` is running it finds it through `control.json` and asks it over the local control socket for the engine/OS-only checks above (real, against the engine's own hub, cloud or community). With no engine running those appear as `pass` + "skipped (requires the engine)".

It takes the headless inputs `--token` / `--org-id` / `--repository-id` / `--timeout`. `--repair` reinstalls the desktop app from the update channel it was installed from, after verifying the download, and refuses on a machine without the desktop app; see `swarmfile-doctor --repair` and Diagnosing Connectivity.

Engine logs live in the cache directory (`~/.cache/swarmfile` on macOS and Linux, `%LOCALAPPDATA%\swarmfile` on Windows) as `engine.log`; the Desktop App writes `tray.log` beside it, and macOS mount-helper output lands in `~/Library/Logs/fuse-t/fuse-t.log`. Diagnostics has an **Open log directory** action. Increase detail with `RUST_LOG=swarmfile_engine=debug`.

Status counters

`swarmfile status` (and the Desktop App's panels) report these; the names are the JSON fields from `swarmfile status --json`:

Counter	Meaning	What to do
`pendingUploads`	Blocks queued on this machine awaiting cloud upload, plus closed files whose save is still running	Wait; check for connection trouble
`peerPendingUploads`	In-flight uploads reported by other machines on this branch	Usually informational
`pendingCreates`	Files/folders created locally and queued for the hub (an object with the count, stuck rows at six or more failed attempts, and the oldest age)	They land automatically; see Coding Agents
`createsReconciled`	Creates whose response was lost and whose registry row the engine then adopted by re-listing — the "dialog said the create failed but the repository exists" case, recovered	Informational; non-zero is for support (`swarmfile status --json`)
`commitsPending`	Bytes uploaded but the commit hasn't landed (retrying, rate-limited, quota-held, or locked)	Check `quotaHeldCommits` and connectivity
`quotaHeldCommits`	Commits refused on billing grounds, retried on a 15-minute cadence; `spendCapHeldCommits` is the subset held by the org's spend cap, which the tray names ("Waiting on spend cap")	Raise the limit or free space; retries resume
`commitsRefused`	Commits the hub keeps refusing for other reasons (a subset of `commitsPending`)	Investigate the refusal; usually policy or conflict
`deferredOps`	Deferred rename/delete queue: ops applied locally and still owed to the hub (`queued`), how many are `held` (parked on a lock, a closed branch scope, or a child move still settling), the oldest op's age, and lifetime `confirmed` / `rolledBack` counts	Informational; `held` is the one to check — it makes no progress until the holder changes. See Working with Files
`abandonedGroups`	Upload groups that exhausted retries or hit a plan cap	`swarmfile sync stuck` lists them; re-save after fixing the cause
`failedWrites`	Guest-session writes permanently given up (guest replay only)	Re-open the guest session and retry
`conflicts`	Files blocked by an unresolved conflict. A blocked write reaches an application as a plain I/O error, so `status` counts it here instead of staying silent; on a repository (or drive) in **hard conflict mode** the losing save fails immediately instead of blocking the file	Run `swarmfile conflicts` to list the paths and resolve
`activeLocks`	Write locks this engine currently holds	Informational
`checkoutLocks`	Long-lived per-file checkout locks (explicit, git-LFS) you hold	Release with `unlock` / `offline return` when done
`scopeReservations`	Offline scope reservations this engine holds — one row can cover a whole subtree; Pack & Go uses these rather than per-file locks	`swarmfile offline status`
`checkoutRenewFailures`	Renewal passes that re-issued nothing — reservations at risk of lapsing	Reconnect and re-prepare before the lease ends
`checkoutExpiringSoon`	Reservations expiring within 24 hours	Extend or return; see Working Offline
`peers`	LAN/office/seed peers currently reachable, with RTT	More peers = faster reads for shared files
`bandwidth.bytesIn` / `bandwidth.bytesOut`	Transfer volume for this session	Useful when a link is saturated
`blockStore`	Pinned blocks and local cache usage	`swarmfile cache status` / `hydrate status`
`auth`	Session state ("signed out" when there's no usable session)	Sign in again
`recentErrors`	A ring of the most recent engine errors	Often the fastest pointer to the root cause
`lastSequenceId`	Latest committed sequence known here	Diagnostic for history lag
`mounted` / `mountState` / `mountError`	Whether the mount is live and, if not, why (a stable code plus a user-facing sentence)	See the mount errors below

Health endpoints

There is **no default health port** — the server starts only when `SWARMFILE_HEALTH_PORT` is set (typical choices: `8080` in container deployments, `9100` for a seed node). It binds `0.0.0.0:<port>` and serves exactly:

- `GET /livez` — a cheap `{"status":"ok"}` for liveness probes (Kubernetes and friends). Allowed from anywhere.
- `GET /healthz` — the detailed snapshot (version, uptime, peer counts and RTT, pin/cache stats, encryption state). **Loopback-only**: a non-loopback caller gets `403 {"error":"loopback only"}`, because the detail exposes operational internals.
- Anything else — `404` .

Doctor checks

`swarmfile-doctor` reports one row per check. The whole report is capped at 10 seconds; if the cap fires, a synthetic `report` check fails, and a probe that couldn't complete counts as a failure rather than a silent pass. Each of the three network checks — `hub https /health` , `oidc token` , `cloud path` — retries one fast transient failure (a connection error or a 5xx) before it reports Fail, so a brief hub blip doesn't read as an outage; a definitive refusal like an expired token's `401` is reported on the first reply. The ones that name a common failure:

Check	Failing usually means
`engine-running`	No engine for this user; sign in / start the app, or expect the standalone checks below
`install-layout`	The engine isn't where the installer put it (moved out of the app bundle, or an older macOS install whose LaunchAgent still runs a stale copy); reinstall or use **Repair**. On macOS it also warns when `/usr/local/bin` is missing the CLI-tool links (a `.dmg`-only install the tray couldn't repair), since `swarmfile` is then not on `PATH`.
`updater-helper-task`	The elevated Windows updater helper is missing or mis-registered; reinstall (Windows)
`dns <host>`	DNS resolution for the hub/identity/storage host failed
`hub https /health`	The hub host is unreachable or blocked; proxy/firewall issue
`hub deep health`	The hub is up but one of its dependencies (its database, storage, the identity provider, or its own clock) is degraded
`oidc token`	The sign-in session expired, was revoked, or the account isn't a member of the configured org; sign in again
`cloud path`	Cloud storage credentials, presigning, or the block path failed for this org
`iroh quic dial` / `mdns listen`	Peer-to-peer connection or local-network peer discovery blocked (firewall, VLAN, VPN); falls back to the hub
`host firewall` / `macos local network`	OS firewall or macOS Local Network consent blocks LAN discovery
`uploads in flight` / `in-flight streaming`	Uploads or reads waiting on in-flight bytes exist — expected during activity, informative otherwise
`fuse-t mount helper` / `mount`	The macOS mount helper died, the mount point is wedged or absent, or a killed session left a stuck mount attempt behind. For an **attached** mount, try **Repair drive** first; a stuck attempt that survives the repair needs a restart of the computer. For a mount that **never attached** (e.g. all helper slots held), **Repair drive** cannot cycle it — free a slot and use the boot gate's **Try again** / `swarmfile mounts retry`; a failed runtime `mounts open` must be re-opened.
`duplicate mount stack`	Two engines are competing for the same mount point
`exclusive writes`	Never fails — informational: names the drives in the opt-in exclusive-write mode
`history retention`	Never fails — informational: the active repository's retention mode (every saved version kept and cold-archived, or named commits only). "Not checked" without a repository scope
`upload quota`	Blocks are parked because the org is at a billing limit — see the count/bytes/age in the row
`plan`	Subscription inactive, a feature isn't in the plan (e.g. seed mode), or a metered dimension is ≥80%
`pack and go policy`	A `SWARMFILE_PACK_AND_GO_POLICY` value that isn't `allowed` / `disabled` — failing closed
`previous run`	The last engine run ended abnormally (panic or unexpected exit) — the row points at `engine.log`
`url scheme`	`swarmfile://` links (Open in app, sign-in, Explorer's Swarmfile menu) won't reach this install: no app is registered for the scheme, another app or install owns it, or (macOS) several copies of Swarmfile compete for it and a link may start an old one. **Fix link handling** re-registers this install; on Windows, a registration pointing at another install needs **Repair** from Settings › Apps

The standalone `swarmfile-doctor` asks a running engine for engine-only checks. With no engine it reports those as a **pass carrying** `skipped (requires the engine)` — treat that as not checked, not as a successful mount/portal probe.

Engine and mount errors (what the app shows)

Code / message	Meaning	Fix
`not_signed_in`	The engine has no usable session	Sign in from the app
`not_a_member`	The session doesn't include the configured org/repository	Switch organization or ask for an invite
`repository_deleted`	The mounted repository was permanently deleted	Mount another repository; contact support about recovery within retention
`config_parse_error`	A `config.json` couldn't be read, so the engine came up without a repository	The message names the file and the parse position — fix it and restart; see Engine Config File
`mount_failed` / `no_free_drive_letter`	Windows could not attach the mount	Free a drive letter or choose one in the app; `swarmfile mounts open auto`
`fuse_t_helper_ports_exhausted`	All 18 fixed FUSE-T helper ports are held — by this engine's own drives, a leaked helper, another Swarmfile engine, another app, or a non-FUSE-T process	Close a drive or quit the app holding one, then use the boot gate's **Try again** / `swarmfile mounts retry`; restart Swarmfile only if that route is unavailable. For a non-helper process, quit that process or restart the computer. `swarmfile doctor` names the holders it can identify; **Repair drive** cannot cycle a mount that never attached
`fuse_t_helper_port_probe_failed`	The engine could not probe FUSE-T's helper ports (resource pressure or a socket error), so the range may be free	Free resources, then use **Try again** / `swarmfile mounts retry` for the boot mount (or re-open a failed runtime mount); restart Swarmfile if it persists. This is not a "close another drive" problem
`drive_letter_in_use`	The chosen letter (or Auto's candidates) are all taken	The message names what holds it; pick another
`mount_point_not_empty`	Windows refuses to mount over a non-empty folder	Choose an empty folder
`pack_and_go_disabled`	An administrator disabled bulk offline reservation	Use a per-file lock or ask your admin
`access_lease_expired`	The organization's offline access window elapsed — this computer hasn't reached Swarmfile within it, so local files are paused	Reconnect to the network; the drive resumes on its own (a restart won't help)
`confirmation_required`	A destructive CLI command was run without `--yes` in a non-interactive context	Re-run with `-y` after checking what it will delete
`usage_error`	A CLI invocation didn't parse (unknown command, flag, or argument); emitted as JSON when `--json` is present, with clap's `kind`	Fix the invocation — see Scripting against the CLI
`changelist_open`	A branch/repository switch was refused because staged work is open	Submit, cancel, or `--park` first
`cross_repository_mount_open`	An org switch was refused because an open drive points at a different repository — that drive can't follow the engine to another org	Close that drive (the Desktop App names it in the switch confirm), then switch
`changelists_open`	`swarmfile mounts close` was refused because the mount has open changelists	Submit or cancel them, or pass `--force`
`is_default_mount`	You tried to close the boot mount on its own	Close another mount, or quit the engine instead
`shared_branch_context`	The mount shares the engine's branch context	Re-open it with `mounts open --branch <name>` to make it independently switchable
`already_running` / `repository_switch_running`	Another switch or long job of that kind is in flight	Wait for it to finish, then retry
`pending_create`	A lock/comment/share on a file that is still being created	Retry after it syncs (`swarmfile status` → `pendingCreates`)
`account_switch_requires_restart`	A runtime sign-in for a **different account** was refused	Restart the app (the tray offers the restart); the requested sign-in is kept and the restart completes the switch — the local cache identity is reconciled at boot, and the previous account's unsent work is quarantined, not lost
`account_switch_park_failed`	The requested account switch couldn't be saved for the restart (e.g. the disk is full)	Your current session is unaffected — free up disk space and try signing in again

Hub error codes (for scripts and integrations)

API refusals carry a machine-readable `code`. The ones most worth handling, in the order you'll meet them:

Code	Meaning	Typical handling
`bad_request`, `invalid_request`, `invalid_state`, `invalid_name`, `bad_cid`, `invalid_cid`, `bad_hash`	Malformed or out-of-order request	Fix the request; don't retry blindly
`credential_invalid`, `email_verification_required`, `email_not_verified`, `session_expired`, `session_revoked`, `unauthenticated`, `terms_acceptance_required`	Auth/session problem	Sign in again; verify email for share actions; accept updated terms
`api_key_*` (`api_key_grant`, `api_key_org_mismatch`, `api_key_repository_mismatch`, `api_key_scope_forbidden`)	API-key scope doesn't cover this operation	Use a key scoped to the repository/org, or mint a new one
`forbidden`, `not_member`, `admin_role_required`, `idp_binding_rejected`	Not permitted for this account/role	Escalate to an admin
`root_create_restricted`	Creating or moving an entry directly at the root of a protected repository, by a caller who isn't the repository creator and holds no repository-wide `write` grant (a folder-scoped grant, even `admin`, doesn't qualify)	Create inside an existing folder, or ask an owner or the repository creator; a held create-queue save resumes once access changes, and a `git push` that would add a root name is refused the same way — its error names the remedy (move the files inside a folder, or gain repository-wide write)
`org_mfa_required`, `org_email_verification_required`	The org's authentication policy refuses this credential	Enroll an authenticator app / verify your email under Account settings, then sign in again — the refusal body's `enrollUrl` points there. Don't retry unchanged; a personal access token must be re-minted from a compliant session
`owner_mfa_required`, `mfa_status_unavailable`	Turning on "require MFA" needs the acting owner's own confirmed factor (or the factor-status probe was unreachable)	Enroll an authenticator app first, then retry the policy change
`identity_conflict`	An external-IdP sign-in conflicts with an existing Swarmfile account: its subject names a first-party user, or a principal provisioned by a different issuer	Contact the organization's administrator; sign in through the org's own SSO rather than a built-in Swarmfile account
`entry_not_found`, `entry_deleted`, `repository_not_found`, `repository_deleted`, `path_not_found`, `not_a_file`, `source_gone`, `target_gone`	The target is gone	Re-list; treat as terminal
`exists`, `name_conflict`, `id_conflict`, `name_taken`	Collision or illegal name	Rename / resolve the conflict
`entry_uploading`	The entry has no committed version yet	Wait for upload, then retry
`unscoped_repository_route`	The client called an org URL that needs `/repositories/:id` — an out-of-date desktop app does this	Update Swarmfile; the desktop app holds the affected save and reports it once instead of retrying
`stream_retired`	The notification/activity stream isn't supported; clients poll instead	Poll `GET /orgs/:orgId/notifications` (and `/activity`); don't reconnect the stream
`stage_discarded`	A `POST /jobs/:id/retry` was refused because the job's staged operation was already discarded	Start the operation again — there is nothing left to retry
`not_cancellable`	`POST /jobs/:id/cancel` (and `swarmfile op cancel`) refused: this operation can't stop part-way — a commit, merge or delete completes or fails on its own	Let it finish; follow it with `op wait`
`too_late`	`POST /jobs/:id/cancel` refused: the staged revert/checkout has already begun applying (including the moment after it published), so there is no unpublished stage to discard	Nothing to stop — the commit lands; the operation ends with its real outcome (`op status` / `op wait`), not `canceled`
`chunk_missing`, `cid_mismatch`, `checksum_mismatch`, `size_mismatch`, `manifest_unreadable`, `manifest_too_large`, `tree_object_corrupt`, `tree_object_missing`, `commit_object_corrupt`, `commit_object_missing`	Content integrity/structure problem	Re-upload the file; report if persistent
`conflict_detected`, `conflict_unresolved`, `merge_conflict`, `stale_merge`	Concurrent edits, or the target moved since the merge/conflict was prepared. With conflict mode **hard**, this surfaces at the saving call itself as a stale-version error (`ESTALE`; on Windows, "the volume for a file has been externally altered")	Re-read the file and retry; resolve the conflict (`swarmfile conflicts resolve`) when one was recorded
(no `code`; the envelope's `error` text says `target_changed` / `merge_candidate_changed`)	The file changed again after the conflict was recorded, or the automatic merge result changed after you previewed it	Re-read the file and resolve against the newer version; run the merge attempt again before accepting. Match the text, not a `code` — the hub sends these two without one

Code	Meaning	Typical handling
`locked`	Someone else holds the file, or a machine's Pack & Go scope reservation covers it	Wait, or request an unlock (`swarmfile unlock-request`); a scope reservation is released by its owner with `swarmfile offline return`
`commit_mode_locked` , `policy_enforced`	An admin lock requires a different commit mode, or refuses the setting	Check the enforced mode with `swarmfile config get-commit-mode` , then use it
`commit_race` , `head_changed` , `merge_in_progress` , `fork_in_progress`	History moved under you, or a merge/fork is running	Re-read, wait, then retry; when a repository job holds the same operation, the refusal names that job — retry or cancel it first
`branch_not_found` , `branch_mismatch` , `branch_locked` , `branch_has_open_mr` , `branch_has_active_children` , `merge_main`	Branch rules prevented the action	Use the allowed path (merge the MR, delete children first, never merge into main directly)
`not_approved` , `self_review` , `is_draft` , `not_draft` , `protected_branch`	Merge-request gates	Get approvals, un-draft, don't self-review
`source_changed`	The merge request's source branch changed after you opened it, so your review would cover changes you haven't seen	Reload the merge request and review it again
`repository_archived` , `storage_exceeded` , `quota_exceeded` , `spend_cap_reached` , `plan_restricted` , `keys_cap` , `subscription_inactive` , `too_many_entries` , `too_many_descendants` , `registry_too_large` , `batch_too_large` , `body_too_large` , `merge_too_large` , `too_many_cids`	Plan or size limit	Free space, upgrade, or split the request/repository; `merge_too_large` means the merge exceeded the single-repository merge capacity — land the branch in smaller pieces — otherwise retry after the allowance changes. `spend_cap_reached` is the dollar ceiling: the envelope carries `capCents` / `usedCents` / `remainingCents` (and `keyLimitCents` / `keyId` when a per-key budget was the binding wall) — raise the cap or free space and held writes resume on their own
`repository_storage_full`	This **repository** hit its metadata ceiling (files and versions combined)	Delete or purge files, delete stale branches, shorten history retention, or switch the repository to **Named commits only** (the same guidance the owner notification carries) — see Repository storage limit
`checkout_required`	The org hasn't completed subscription checkout	Finish checkout in Billing before creating repositories
`rate_limited`	Request budget exceeded	Back off and retry; see Rate limits
`grace_period_inaccessible`	The org is in the post-unpaid 28-day window	Resubscribe — see Data Portability
`quarantined`	Ransomware detection quarantined the account; writes stop until an owner releases it (on Windows the drive stays mounted meanwhile, with saves held locally)	Review and release from the quarantine panel — see Ransomware quarantine
`repository_key_unavailable` , `repository_key_rotation_unsupported`	Encryption key unavailable, or the repository can't rotate (plaintext storage, or it has reached the maximum key generations)	Check the repository's encryption tier; contact support if the cap is reached
`stale_generation` , `incomplete_recipients` , `unexpected_recipients` , `key_rotation_cap_reached`	E2E key rotation conflicts: another device rotated first; the caller's wrap set was missing (or named non-) current recipients; or the 64-generation cap was reached	Re-run the rotation (the engine re-fetches the recipient set and retries once automatically); an owner should confirm the device roster before rotating
`e2e_upload_unsupported` , `e2e_unsupported` , `e2e_not_supported` , `encryption_tier_not_public`	The operation isn't allowed for this tier	Use the supported tier (e.g. LFS is rejected on E2E)
`jurisdiction_incompatible`	The destination isn't in the pinned jurisdiction — the org's, or a repository's own	Use storage/a destination in the pinned region
`residency_unavailable` , `residency_storage_unavailable` , `jurisdiction_unverifiable` , `residency_unverifiable`	A per-repository residency pin was refused: this deployment can't route one, the region's storage isn't ready yet, or the check couldn't be completed just now	Retry shortly (a region still provisioning becomes ready on its own); if it persists, contact support
`share_too_large` , `block_not_in_share` , `block_session_expired` , `raw_byte_budget_exceeded`	Share-link limits or a public-streaming budget	Reduce scope, create a new share, or wait out the budget
`too_many_webhooks` , `blocked_target` , `invalid_url`	Webhook limits or an unsafe target URL	Delete unused webhooks; use a public HTTPS endpoint
`preview_budget_exhausted` , `preview_too_large` , `preview_type_unsupported`	Preview generation budget, size, or type limit	Wait for the daily reset; serve the file directly or download it
`too_many_active_jobs` , `too_many_concurrent_reads`	Temporary concurrency limit	Back off and retry
`do_write_budget_exceeded`	The repository's writes are paused for the rest of the UTC day after it wrote several times its daily budget; starting a background job, a merge or a `git push` is refused (saves, reads and deletes still work)	Retry after the body's `retryAtMs` or the `Retry-After` header; an org owner or admin can release the pause once a day. See Write pause

Code	Meaning	Typical handling
`write_budget_release_used`, `write_budget_not_tripped`	A write-pause release was refused: today's release for this repository is already used, or nothing is paused	After the daily release, only Swarmfile support can release the repository; otherwise nothing to do
`fork_index_building`	The hub is still building its index of the files this branch inherited	Temporary: retry after `retryAfterSeconds` / `Retry-After`; the Desktop App retries a commit on its own
`fork_too_large`	The branch inherited more files than the hub can index, so commits and pushes on it can't land	Final for that branch: work on the default branch or ask the repository owner. See Branches & Merging
`ci_not_available`, `ci_disabled`, `ci_branch_unprotected`, `ci_e2e_unsupported`	Hosted CI is refused for this org, repository or branch: a repository owner hasn't enabled it, the branch isn't protected and unprotected runs aren't opted in, a hosted job targets an end-to-end encrypted repository (a hosted runner holds no E2E key), or a self-hosted E2E run has no plaintext sidecar or no live granted runner	Enable it as an owner (`swarmfile ci enable`); protect the branch, or opt in with `--allow-unprotected`; for E2E use a self-hosted runner
`self_hosted_runners_not_enabled`, `self_hosted_runners_unavailable`	Self-hosted CI is disabled for this organization, or the organization's policy could not be read	An owner can enable or disable it under **Settings → Hosted CI → Policy**; a policy-read failure is transient, so retry the run — see Self-Hosted CI Runners
`ci_e2e_no_workflow`, `ci_e2e_hosted_unsupported`, `ci_e2e_tree_file_unsupported`	Self-hosted E2E CI is refused: the repository has no plaintext workflow sidecar, a hosted-size job is declared in an E2E workflow, or the sidecar uses a local action/reusable workflow (their sources are encrypted tree files)	Save a self-contained workflow in the CI tab; run E2E jobs on a `runs-on: [self-hosted, ...]` job; keep the sidecar self-contained — see Self-Hosted CI Runners
`ci_workflow_invalid`, `ci_yaml_invalid`, `ci_expression_invalid`, `ci_jobs_too_many`, `ci_secret_undeclared`	`.swarmfile/ci.yml` didn't compile (the error carries the line), the expansion passed 256 jobs, or a job referenced a secret it didn't declare	Fix the workflow; declare every secret a job reads in its `secrets:` list — see Hosted CI
`invite_unavailable`, `org_not_empty`, `installation_exists`, `already_provisioning`, `storage_provisioning`	Admin lifecycle state	Complete or wait for the pending operation
`announce_unverified`	A seed node's announce failed node authentication	Re-enroll the node; see Self-Hosted Seed Nodes
`org_migration_frozen`, `repository_unreachable`, `async_unavailable`	A migration/maintenance state	Retry after it completes
`not_supported_on_this_server`, `server_profile_too_old`	Returned by the desktop app's engine (and so the CLI) against a self-hosted community server: the server doesn't offer this feature, or it is older than this app needs. The CLI adds a `hint:` line (a `hint` field under `--json`) for the first	Use the hosted service for that feature; for a new repository, ask the server's administrator to create it (`swarmfile-server admin create-repository`); upgrade the server for `server_profile_too_old`
`not_found`, `forbidden`	Generic — the meaning depends on the operation	Use your caller's contextual fallback; don't guess

The full set is generated from the hub's own routes and shipped in the hub's published API schema and OpenAPI contract; an unrecognized code is preserved verbatim rather than dropped, so a script can always log it.

Exit codes

Binary	Codes
`swarmfile`	`0` success; `1` failure (unreachable engine, transport error, no product equivalent); `2` actionable refusal **or a usage error** — under `--json` the two are distinguishable by `code` (`usage_error` vs the refusal's code); `3` a long operation is still running (returned by `--no-wait`, or after Ctrl-C detaches the command) — follow it with `swarmfile op status` / `op wait`; see Scripting against the CLI
`swarmfile-doctor`	`0` all passed; `1` ≥1 failure; `2` warnings only. With `--repair`: `0` reinstalled, `1` a step failed, `3` refused or canceled with nothing changed. A usage error under `--json` prints the shared `{"code":"usage_error",…}` body, as every standalone tool does
`swarmfile-runner`	The engine's code (the alias re-execs it); `127` if the sibling engine is missing
`swarmfile-migrate`, `swarmfile-search`, `swarmfile-seed`, `swarmfile-mirror`	`0` success; `1` failure (a failed verification or upload, an unreachable hub, a refused destination collision); `2` a malformed invocation — the shared `{"code":"usage_error",…}` body under `--json`
`swarmfile-verify-history`	`0` the loaded history verified; `1` verification found tampering or a gap; `2` the check couldn't run (bad arguments, unreachable hub, auth error) — see the CLI reference
`swarmfile-engine lfs-transfer`	Not a scripted CLI: git-LFS spawns it for one operation and reads its protocol replies. It exits `1` only when it can't start
`git-remote-swarmfile`	Not a scripted CLI: git spawns it for `git clone` / `git push` over a `swarmfile://` remote. A failure prints its reason (and writes `git-remote-last-error.log`) and exits non-zero — see the CLI reference

Where to go next

- Troubleshooting — symptom-first workflows.
- Network Requirements — the hosts and ports the checks probe.
- Environment Variable Index — `SWARMFILE_HEALTH_PORT`, `RUST_LOG`, and the diagnostic switches.

Environment Variable Index

Almost every engine setting has an environment-variable form, and some settings exist **only** as environment variables. This page is the index: what each supported `SWARMFILE_*` variable does, its default, and whether the same setting can be put in `config.json` instead.

Use environment variables when a launcher can set them (systemd, Docker, a scheduled task, a CI job). Use `config.json` when it can't — notably macOS launchd agents and Windows services, which don't inherit a shell environment. See Engine Config File for the file's precedence rules and the list of keys a packaged install manages itself.

To see what a running install actually resolved — secrets redacted — run `swarmfile-engine --print-env-vars`.

Conventions

- Booleans accept `true` / `1` to enable; anything else (including an empty string) is false. A few "disable with" flags are the reverse: only `false` / `0` disables them.
- Empty strings are treated as unset for credentials, ids, paths, and tool commands.
- Values set here beat the `config.json` file; both beat built-in defaults.
- Variables marked **advanced** are safe but rarely needed — reach for them while diagnosing, not as standard configuration.
- Variables marked **diagnostic only** are for support sessions and are noisy.
- Variables marked **security-relevant** weaken a protection when set; read the row before using one.

Credentials, identity, and routing

Variable	Default	Purpose	File key
`SWARMFILE_SERVER_PUBLIC_URL`	*(unset)*	**Community server only.** The public origin (`scheme://host[:port]` , no path) the server publishes in `/.well-known/swarmfile.json` . The identity document signs this configured value, never the request `Host` ; unset means the route answers 503 and `--check-host-identity` fails closed. Set it in production (it is not secret).	—
`SWARMFILE_HUB_URL`	`https://hub.swarmfile.com` on a packaged install	Hub API base URL.	`hub_url`
`SWARMFILE_SITE_URL`	derived from the hub host (`hub.X` → `X`) for engine-printed links; the compiled build value for the tray	Web-app base URL for links: the tray's in-app and dashboard links, and the links the engine's git-push helper prints. Wins over the installer's `site_url` . With neither, the git-push helper derives it from the hub host (and omits the link when even that can't be derived); the tray uses the value compiled into the build.	`site_url`
`SWARMFILE_ORG_ID`	*(unset)*	Org whose hub routes are called.	`org_id`
`SWARMFILE_REPOSITORY_ID`	*(unset)*	Repository every metadata operation is scoped to (required for headless nodes).	`repository_id`
`SWARMFILE_BRANCH`	the repository's default branch (each repository names it at creation — commonly `main`)	Branch this mount starts on.	`branch`
`SWARMFILE_OFFICE_ID`	`default`	Peer-grouping label for same-office LAN behavior.	`office_id`
`SWARMFILE_OIDC_ISSUER`	`https://id.swarmfile.com` on a packaged install	Identity provider the engine signs in against.	`oidc_issuer`
`SWARMFILE_OIDC_CLIENT_ID`	`swarmfile-desktop` on a packaged install	OIDC client id.	`oidc_client_id`
`SWARMFILE_OIDC_REFRESH_TOKEN`	*(unset)*	Non-interactive sign-in for headless engines; the rotated token is persisted under the cache dir.	—
`SWARMFILE_API_KEY`	*(unset)*	Repository-scoped `sf_key_…` credential; wins over a refresh token if both are set.	—
`SWARMFILE_NO_KEYCHAIN`	*(unset)*	Skip the OS keystore (Keychain, DPAPI, Secret Service): the sign-in refresh token and your end-to-end encryption private key are written as **plain files** in the cache dir, protected only by owner-only file permissions (`0600` on macOS and Linux; the user-profile ACL on Windows). Only for a headless machine with no usable keystore; protect that directory accordingly. **Any value enables it**, including `0` , `false` or an empty string. **Security-relevant**	—
`SWARMFILE_SWARM_KEY`	*(unset)*	Extra deployment-wide entropy for the private peer-to-peer swarm.	—
`SWARMFILE_USER_ID`	`$USER` / `$USERNAME` , else `anonymous`	Display label recorded for locks and presence (not an authenticated identity; your sign-in decides access).	—
`SWARMFILE_MACHINE_ID`	OS hostname (tray: hash of hostname + data dir)	Machine identity for locks and presence.	—
`SWARMFILE_NODE_AUTH`	`false`	For orgs that enforce P2P node authentication: refuse all peers until the hub confirms the approved roster.	`node_auth`
`SWARMFILE_LAN_FROM_OFFICE`	`false`	Treat same-office peers discovered through the hub as LAN peers when mDNS is blocked.	`lan_from_office`

Paths and local data

Variable	Default	Purpose	File key
`SWARMFILE_CONFIG`	*(unset)*	Explicit single `config.json` path; bypasses the install + per-user merge.	—
`SWARMFILE_DATA_NAMESPACE`	*(empty)*	Suffixes the data dir, mount, socket, and lock so a second install can coexist.	`data_namespace`
`SWARMFILE_CACHE_DIR`	`~/.cache/swarmfile` (macOS/Linux), `%LOCALAPPDATA%\swarmfile` (Windows)	Block/metadata cache, auth token, control socket, logs.	—
`SWARMFILE_CACHE_MAX_BYTES`	`10737418240` (10 GiB)	Local block-cache cap.	`cache_max_bytes`
`SWARMFILE_MOUNT_POINT`	`~/Swarmfile` (Unix), `S:` (Windows)	Where the boot mount attaches.	`mount_point`
`SWARMFILE_IROH_DIR`	`<cache_dir>/iroh`	Peer-to-peer node identity and blob store.	—
`SWARMFILE_READ_ONLY`	`false`	Force a kernel read-only mount (guest sessions).	`read_only`
`SWARMFILE_SESSION_KIND`	`owner`	Session role (`owner` / `member` / `guest`).	`session_kind`
`SWARMFILE_HUB_ONLY`	`false`	Cloud-only mode: no QUIC, no mDNS, no peer traffic. An org policy can also force it on.	`hub_only`

Networking and peer discovery

Variable	Default	Purpose	File key
`SWARMFILE_IROH_BIND_ADDR`	OS-assigned ephemeral port	Fixed QUIC bind address, for a firewall that needs a predictable port.	`iroh_bind_addr`
`SWARMFILE_IROH_PUBLIC_ADDR`	*(unset)*	Public address advertised to the hub when behind NAT.	—
`SWARMFILE_IROH_PEER_BUDGET_SECS`	`30`	Aggregate time budget for peer fetches on interactive paths. **Advanced**	—
`SWARMFILE_IROH_PEER_BUDGET_BACKGROUND_SECS`	`10`	Same budget for background prefetch. **Advanced**	—
`SWARMFILE_HEALTH_PORT`	*(unset — health server off)*	Port for the `/healthz` and `/livez` endpoints.	—
`SWARMFILE_MDNS_INTERFACES`	auto-selected LAN interfaces	Comma-separated allowlist mDNS must use exclusively.	—
`SWARMFILE_MDNS_EXCLUDE_INTERFACES`	*(unset)*	Comma-separated interfaces mDNS must never use.	—
`SWARMFILE_MDNS_SERVICE_TYPE`	`_swarmfile._udp.local.`	Override the mDNS service type, to keep separate groups of machines on one LAN from discovering each other. Must be a valid `_<name>._udp.local.` type; an invalid value is logged and disables mDNS (the engine continues with hub-only peer discovery) rather than silently using the default type. **Advanced**	—
`SWARMFILE_WORKSHARING_FAIL_CLOSED`	`false`	Refuse an exclusive-open check when the hub is unreachable, instead of allowing the open. Windows only.	—

Encryption and erasure coding

Variable	Default	Purpose	File key
`SWARMFILE_ENCRYPTION_ENABLED`	`true`	Master switch for client-side block encryption. **Leave it unset.** Any value other than `true` / `1` turns encryption off, and the engine then uploads new blocks **unencrypted into any repository, including managed and end-to-end encrypted ones**, breaking that repository's at-rest guarantee. The only legitimate use is a seed node mirroring blocks that are already ciphertext. **Security-relevant**	—
`SWARMFILE_ENCRYPTION_KEY`	*(unset)*	Hex 32-byte static master key (no hub key fetch). It bypasses the hub-managed key, so hub previews and other members can't read blocks sealed with it, and losing the key loses that data. **Security-relevant**	—
`SWARMFILE_ENCRYPTION_REPOSITORY_ID`	falls back to `SWARMFILE_REPOSITORY_ID`	Repository whose hub-managed key is fetched.	`encryption_repository_id`
`SWARMFILE_EC_ENABLED`	`auto`	Erasure coding: `on` / `true` , `off` / `false` , or `auto` (adapt per save).	—
`SWARMFILE_EC_LAN_PLACEMENT`	`false`	Deliberately spread erasure shards across LAN peers.	`ec_lan_placement`
`SWARMFILE_EC_LAN_PLACEMENT_REPLICATION`	`3`	Replication factor K per shard. **Advanced**	—
`SWARMFILE_EC_LAN_PLACEMENT_POLL_INTERVAL`	`300` (seconds)	Placement reconcile interval. **Advanced**	—

Seed mode and CI runner

Variable	Default	Purpose	File key
`SWARMFILE_SEED_MODE`	`false`	Run as a NAS/seed node: fetch every block into a warm cache, mirror to cloud storage, no mount. A headless seed also needs `SWARMFILE_ORG_ID` / `_REPOSITORY_ID` and a credential; the dashboard's **Settings → Office caches** wizard generates the full file.	`seed_mode`
`SWARMFILE_SEED_POLL_INTERVAL`	`60` (seconds)	How often a seed polls the hub for metadata changes.	—
`SWARMFILE_SEED_UPLOAD_CONCURRENCY`	`8`	Concurrent cloud uploads from a seed.	—
`SWARMFILE_RUNNER_MODE`	set by `swarmfile-runner`	Headless CI runner; watches `SWARMFILE_BRANCH` for commits/tags matching `.swarmfile/runner.yml`. A runner also needs a **repository API key** (`sf_key_…`), which resolves its org and repository automatically (or set `SWARMFILE_ORG_ID` + `SWARMFILE_REPOSITORY_ID`, or an org-scoped hub URL). The hosted hub refuses to start with a PAT unless `SWARMFILE_RUNNER_ALLOW_PAT=1` (jobs run, reports are refused). Refuses to execute the rule file on a branch without an effective branch protection unless `SWARMFILE_RUNNER_ALLOW_UNPROTECTED=1` is set (trusted/dev runners only).	—
`SWARMFILE_RUNNER_ALLOW_UNPROTECTED`	*(unset — protected branches only)*	Let a headless runner execute its rule file on a branch with no effective branch protection. Trusted/dev runners only; leave unset in production.	—

Create queue, streaming, and mount I/O

Variable	Default	Purpose	File key
`SWARMFILE_PENDING_CREATE_WAIT_SECS`	`8` (0–120; `0` refuses immediately)	How long an operation on a still-syncing file waits before answering `409 pending_create`.	—
`SWARMFILE_CREATE_DISPATCH_CONCURRENCY`	`8` (1–64)	Creates sent to the hub at once. **Advanced**	—
`SWARMFILE_CREATE_COALESCE`	on (`0` / `false` / `off` disables)	Attach the first save to a create instead of sending the create empty. **Advanced feature switch**	—
`SWARMFILE_CREATE_HOLD_SECS`	`2` (0–10; `0` disables holding)	Max wait for the first save of an open-but-unsaved file. **Advanced**	—
`SWARMFILE_CREATE_HOLD_UPLOADING_SECS`	`60` (0–600)	Same, when that first save is already uploading. **Advanced**	—
`SWARMFILE_CREATE_BATCH`	on (`0` / `false` / `off` disables)	Batch creates through the hub's batch route. **Advanced feature switch**	—
`SWARMFILE_ARRIVING_REPORTS`	on (`0` / `false` / `off` disables)	Publish the "N files arriving" presence hint while creates are in flight. **Advanced feature switch**	—
`SWARMFILE_STREAM_BUFFER_SECS`	`30` (seconds)	Read-ahead a `fetch --follow` job keeps buffered. **Advanced**	—
`SWARMFILE_STREAM_READ_WAIT_SECS`	`15` (seconds)	How long one mount read waits for in-flight bytes before a retryable error. **Advanced**	—
`SWARMFILE_STREAM_WAIT_CONCURRENCY`	`4` (1–64)	How many reads may be waiting at once. **Advanced**	—
`SWARMFILE_READAHEAD_CONCURRENCY`	`12` (1–64)	Concurrent mount read-ahead fetches for an established sequential run; a run's first, speculative batch stays at 4. **Advanced**	—
`SWARMFILE_READAHEAD_MAX_MB`	*(per file type)*	Mount read-ahead window ceiling, in MiB; `0` disables read-ahead entirely. **Advanced**	—
`SWARMFILE_READAHEAD_BUFFER_SECS`	*(per file type; 15 s for video containers)*	Seconds of playback the rate-adaptive read-ahead ceiling keeps ahead; `0` = the flat profile ceiling. Overrides the file value. **Advanced**	`readahead_buffer_secs`
`SWARMFILE_MOUNT_ATTEMPT_TIMEOUT_SECS`	`120` (`0` disables the bound)	Wall-clock bound on a mount attempt so a wedged driver can't hang the app. **Advanced**	—
`SWARMFILE_FS_CALLBACK_RESCUE_SECS`	*(unset — off)*	Windows only. When a drive operation has been stuck this many seconds (30–3600), reset that drive the way **Repair drive** does, so the app waiting on it is released; at most once per drive every 10 minutes. Unsaved changes in the app that hung may be lost. Off by default: a stuck operation is only logged. **Advanced**	—
`SWARMFILE_FS_CALLBACK_STUCK_SECS`	`30` (5–3600)	Windows only. A callback running longer than this is logged as stuck (the WARN the rescue knob reacts to). **Advanced**	—
`SWARMFILE_FS_CALLBACK_WATCHDOG_TICK_SECS`	`10` (5–3600)	Windows only. How often the callback watchdog scans for stuck operations. **Advanced**	—
`SWARMFILE_MOUNT_OPEN_WAIT_SECS`	`20` (`0` = don't wait)	How long `mounts open` waits for the platform mount to settle. **Advanced**	—
`SWARMFILE_PACK_AND_GO_POLICY`	`allowed`; any unrecognized value disables Pack & Go (and is logged at startup, plus a `swarmfile-doctor` warning)	Operator gate for bulk offline reservation — see Working Offline.	—
`SWARMFILE_GIT_ACCEPT_PROVISIONAL`	`false`	The `swarmfile://` remote helper accepts **provisional** (derived-but-unconfirmed) commit ids when resolving a shallow clone's boundary — needed for `--depth` on a repository whose commits only one person derives. Unset, a boundary id must be confirmed. It trusts commit ids no one has confirmed. **Security-relevant**	—
`SWARMFILE_GIT_TIMING`	*(unset)*	The `swarmfile://` helper prints per-phase timings to stderr (`live`, `prepare`, `preflight`, `upload`, `apply`, `derive`, `claim`, `verify`, `after-apply`, `chain`) for diagnosing a slow push. Output only; no behavior change.	—
`SWARMFILE_GIT_INSECURE_HOSTS`	*(unset)*	Comma-separated `host[:port]` authorities an explicitly host-addressed `swarmfile://<host>/…` URL may reach over plaintext HTTP. The match is the exact authority (a trailing `/` is ignored); loopback (`localhost`, `127.0.0.1`, `[::1]`) never needs an entry. Without an entry the helper uses HTTPS. **Security-relevant**	—
`SWARMFILE_ALLOW_CROSS_HUB_KEY`	*(unset)*	`1` / `true` lets an explicit `SWARMFILE_API_KEY` be sent to a `swarmfile://` URL that names a different hub than the configured one (for the standalone helper, than `SWARMFILE_HUB_URL` when it is set). Unset, the environment key stays with the configured hub, and the stored Desktop session and the key saved for a self-hosted server are never offered to another authority (a git credential stored for that host still applies). **Security-relevant**	—

Variable	Default	Purpose	File key
`SWARMFILE_CONFLICT_MODE`	*(unset — soft)*	How this engine reports a save that lost a concurrent-edit race: `soft` parks it and surfaces the conflict later (the default); `hard` makes the saving `fsync` (Linux also the close-time flush; Windows `FlushFileBuffers`; a macOS `close()` cannot carry the error) wait for the server and fail with a stale-version error — for automated writers. Unrecognized values fall back to soft (unpinned). A repository's `conflict-mode` setting applies unless this is set; per-mount overrides go through `mounts open --conflict-mode`. **Advanced**	—
`SWARMFILE_DEFERRED_RENAME` (alias `SWARMFILE_DEFERRED_NAMESPACE`)	*(unset — on)*	Apply non-displacing/displacing renames and file/empty-directory deletes of confirmed entries locally and queue them, while hub confirmation, retraction and refresh run in the background. Set to `0` / `false` / `no` / `off` to opt out; any other value is ignored (and logged) so the config file/default applies. Also settable as `deferred_namespace` in `config.json` (environment beats file; if both spellings are set, `SWARMFILE_DEFERRED_RENAME` wins). **Advanced**	`deferred_namespace`

Bandwidth, QoS, and sync behavior

Variable	Default	Purpose	File key
`SWARMFILE_THROTTLE_ENABLED`	`false`	Enable bandwidth throttling; set a site bandwidth first.	`throttle_enabled`
`SWARMFILE_SITE_DOWNLOAD_BANDWIDTH_MBPS`	*(unset)*	Site download capacity.	`site_download_bandwidth_mbps`
`SWARMFILE_SITE_UPLOAD_BANDWIDTH_MBPS`	*(unset)*	Site upload capacity.	`site_upload_bandwidth_mbps`
`SWARMFILE_OFFICE_HOURS_START`	`08:00`	Start of the office-hours window (local `HH:MM`).	`office_hours_start`
`SWARMFILE_OFFICE_HOURS_END`	`18:00`	End of the office-hours window.	`office_hours_end`
`SWARMFILE_OFFICE_HOURS_WAN_PCT`	`20`	WAN percentage allowed during office hours.	`office_hours_wan_pct`
`SWARMFILE_OFF_HOURS_WAN_PCT`	`80`	WAN percentage allowed outside them.	`off_hours_wan_pct`
`SWARMFILE_DSCP_INTERACTIVE`	`34`	DSCP marking for interactive traffic. **Advanced**	—
`SWARMFILE_DSCP_BACKGROUND`	`0`	DSCP marking for bulk traffic. **Advanced**	—
`SWARMFILE_XREF_PREFETCH`	`true`	Prefetch `.dwg` / `.max` external references on open.	—
`SWARMFILE_SYNC_IGNORE_ENABLED`	`true`	Honor `.gitignore` / `.swarmfileignore` exclusion.	`sync_ignore_enabled`
`SWARMFILE_CHANGESET_IDLE_SECS`	`300` (`0` disables session minting)	Idle timeout before a staged changeset commits.	`changeset_idle_secs`
`SWARMFILE_AGENT_CHECKPOINT_IDLE_SECS`	`30` (`0` uses the normal idle window)	Quiet period before an `agent/<label>-<6 hex>` workspace closes its changeset as a checkpoint.	`agent_checkpoint_idle_secs`
`SWARMFILE_COMMIT_LINGER_MS`	`0` (off; clamped to 2000)	Milliseconds to gather a burst of ready commits into one batch before committing. **Advanced**	—
`SWARMFILE_RWS_TTL_DAYS`	`7`	How long a read or written entry stays in the recent-working-set list that offline writes are admitted against. **Advanced**	—
`SWARMFILE_AUTO_OPEN_CHANGELIST_FOR_PREFERENCE`	`false`	Open a changelist on boot when the user's commit-mode preference asks for it.	`auto_open_changelist_for_preference`
`SWARMFILE_ATTR_CACHE_SECS`	`2` (`0` keeps FUSE-T's own default)	macOS/Linux attribute-cache ceiling.	`attr_cache_secs`
`SWARMFILE_MERGETOOL_CMD`	*(unset — git config fallback)*	External merge-tool command template.	`mergetool_cmd`
`SWARMFILE_MERGETOOL_NAME`	*(unset)*	Display name for that tool.	`mergetool_name`

Diagnostics and telemetry

Variable	Default	Purpose	File key
`SWARMFILE_FUNNEL_BEACON`	on (`0` / `false` / `no` / `off` disables)	The anonymous install / clone-completion beacon the engine and the `swarmfile://` helper send to the hosted hubs. Hosted service only — a community or self-hosted hub never receives it. See Telemetry & Data Collection.	—
`SWARMFILE_DOCTOR_PERIOD_SECS`	`300` (`0` disables)	Period of the engine's background connectivity self-check.	—
`SWARMFILE_MEMBERSHIP_WATCHDOG_SECS`	`5`	How often the engine's membership-enforcement loop runs locally (flag checks, purge/close retries). A definitive refusal from any hub call sets the revocation immediately, so this interval affects how long an unobserved loss can linger, not detection itself; for an unmounted repository, detection waits for the next hub contact. `0` /unset falls back to `5` .	—
`SWARMFILE_UPLOAD_TRACE`	off (`1` / `true` on)	Promote per-attempt upload logging to `warn` , including the hub's refusal body. **Diagnostic only**	—
`SWARMFILE_FUSE_DEBUG`	off	libfuse protocol tracing (macOS). **Diagnostic only**	—
`SWARMFILE_WINFSP_DEBUG`	off	WinFsp protocol tracing (Windows). **Diagnostic only**	—

Installers and the git-LFS agent

These are consumer/operator knobs rather than engine settings.

Variable	Default	Purpose
`SWARMFILE_FW_PROFILES`	`3` (Domain + Private; bitmask: 1 Domain, 2 Private, 4 Public)	Windows MSI property controlling which firewall profiles get the LAN rules, e.g. `msiexec /i Swarmfile-x64.msi SWARMFILE_FW_PROFILES=7 /qn` .
`SWARMFILE_LFS_VERSION`	latest release	Pin the git-LFS agent version installed by the agent installer.
`SWARMFILE_LFS_BINDIR`	`~/.local/bin` (Unix), `%LOCALAPPDATA%\Swarmfile\bin` (Windows)	Where the LFS agent installer puts the binary.
`SWARMFILE_LFS_REPO`	`swarmfile/swarmfile-lfs`	Release repo the installer pulls from.
`SWARMFILE_LFS_DOWNLOAD_BASE`	provider default	Mirror/CDN base for agent downloads.
`SWARMFILE_LFS_API_URL`	provider default	Release-listing API override for a mirror.
`SWARMFILE_LFS_ALLOW_UNVERIFIED`	unset (`1` = allow)	Proceed when a mirror strips `.sha256` checksums. **Security-relevant** — only for a mirror you trust.

Standard environment variables the product honors

Variable	Effect
`RUST_LOG`	Log filter for the engine, CLI, and Desktop App backend (for example `RUST_LOG=swarmfile_engine=debug`).
`HOME` / `XDG_CONFIG_HOME`	Base for the cache dir and the per-user `config.json` on macOS/Linux.
`LOCALAPPDATA` / `USERPROFILE`	Base for the data dir and config discovery on Windows.
`USER` / `USERNAME`	Fallback identity when `SWARMFILE_USER_ID` is unset.
`NO_COLOR`	Disable ANSI color in `swarmfile-doctor` output.

Where to go next

- Engine Config File — the file form of these settings, precedence, and live vs restart-required keys.
- Network Requirements — what the networking variables above connect to.
- Diagnostics & Error Reference — the doctor checks that read these values.

Engine Config File (`config.json`)

The Swarmfile engine reads its startup configuration from these sources, in this order of precedence:

1. **Environment variables** (`SWARMFILE_*`) — always win, with one exception: `data_namespace` on a packaged install, where the bundled install config wins even over the env var (see "A note on environment-selection keys" below).
2. **A stored OIDC session** *(applies to `oidc_issuer` / `oidc_client_id` only)* — after an interactive sign-in, the engine remembers the issuer and client id it signed in against; those beat a hand-edit of `oidc_issuer` / `oidc_client_id` in the file. Editing either key on a signed-in install can appear to do nothing until you sign in again.
3. **The `config.json` file** — the subset of settings documented below.
4. **Built-in defaults** — a released build ships pointing at Swarmfile's hosted service, so an unconfigured engine connects to production rather than nothing.

Most people never touch this file: the Desktop App writes the keys it manages for you, and a normal sign-in fills in the rest. You edit `config.json` directly for **headless and unattended setups** — a NAS/seed node, a render-farm worker, a VM, or any machine that boots the engine from autostart rather than a signed-in Desktop App session — where there's no UI to click and, on macOS/Windows, no shell environment for the engine to inherit exported variables from.

Where the file lives

Platform	Default path
macOS / Linux	`~/.config/swarmfile/config.json` (respects `$XDG_CONFIG_HOME`)
Windows	`%LOCALAPPDATA%\swarmfile\config.json`

That per-user path is only half the picture. A packaged install also ships its own **install config** — read-only, written by the installer, and living next to the binary (on macOS, inside the bundle's `Contents/Resources/config.json`; on Linux's `.deb`, `/etc/swarmfile/config.json`). At startup the engine merges the two, generally with the per-user file winning per key — except for a small, fixed set of keys a packaged install manages itself, where the install config wins instead. See "A note on environment-selection keys" below for exactly which keys, and why, before hand-editing any of them.

To point the engine at a **specific file** instead of this two-file merge — e.g. one config per repository on a multi-tenant node — set `SWARMFILE_CONFIG=/path/to/config.json`. That bypasses the merge entirely and reads only that one file.

Editing safely

A few things are worth knowing before you hand-edit this file:

- **Restart the engine after editing.** The file is read once at startup.

  A handful of values can also be applied to a running engine through the control socket with no restart: `cache_max_bytes`, `readahead_buffer_secs`, `throttle_enabled` and the four office-hours keys, `office_id`, `lan_from_office`, and `branch`. Everything else is read only at startup (including `seed_mode`, `hub_only`, `ec_lan_placement`, `changeset_idle_secs`, `attr_cache_secs`, `mount_point`, `read_only`, and `session_kind`). `sync_ignore_enabled` updates live and then drains and restarts the engine. Either way, a change to the **file itself** needs a restart to be read.

- **A single syntax error discards the *whole* file — and says so.** If the JSON doesn't parse, the engine logs a warning, ignores that file, and keeps going with whatever the other sources provide.

  On a packaged install that means the bundled install config and environment still apply — but a broken per-user file still drops your `org_id` and `repository_id`, so the drive has nothing behind it. Instead of a generic "no repository" prompt, the Desktop App's status then names the file and the parser's position (for example `config.json: expected value at line 1 column 3`), because a repository picker can't help when the org id lived in the file that failed to parse. (With `SWARMFILE_CONFIG`, there is no other file, so a bad file leaves the engine on built-in defaults.)

  A JSON syntax error — a trailing comma, say — silently drops those along with everything else in the file. Validate your JSON before restarting.

- **Unknown keys are named, not ignored in silence.** A valid file with a misspelled key (say `repository_idd`) loads, but the engine warns — and when there is no repository to mount, the Desktop App's status names the unrecognized key(s), because a typo like that is otherwise invisible: the setting simply never applies. The key list is derived from the engine's own parser, so the key table below stays current with it.

- **A UTF-8 byte-order mark (BOM) is tolerated.** Windows Notepad and PowerShell 5.1's `-Encoding UTF8` prepend one; the engine strips it. Still, prefer a plain UTF-8 editor.

- **Environment variables override the file.** If a launcher (systemd unit, Docker `-e`, a scheduled task) sets `SWARMFILE_HUB_URL`, editing `hub_url` in the file will appear to do nothing. Check what actually took effect with `--print-env-vars` (below).

- **Don't fight the Desktop App.** For keys the Desktop App or `swarmfile` CLI manage (mount point, seed mode, hub-only, cache size, read-ahead, throttle, LAN placement, branch), prefer their controls — the Desktop App may rewrite the file and overwrite a manual edit. Hand-editing is for machines with no Desktop App.

Connecting to the hosted service

A released build ships pointing at Swarmfile's production service, so you do **not** need to configure the hub or identity provider by hand. With nothing set, the engine defaults to:

Setting	Default
`hub_url`	`https://hub.swarmfile.com`
`oidc_issuer`	`https://id.swarmfile.com`
`oidc_client_id`	`swarmfile-desktop` on a packaged install; `swarmfile-engine` is the bare binary's fallback when no bundled config supplies one

A headless machine on the hosted service therefore only needs to say *which* org and repository it serves and supply a credential — `org_id`, `repository_id`, and one of `SWARMFILE_API_KEY` or `SWARMFILE_OIDC_REFRESH_TOKEN` (see below). Set `hub_url`, `oidc_issuer`, or `oidc_client_id` only when you run a **self-hosted or Enterprise hub** on your own domain.

Keys you can set in the file

Every key is optional. Booleans are JSON booleans (`true` / `false`) in the file — the `true` / `1` string form is only for the environment-variable equivalents.

Key	Type	Default	Env override	What it doe
`hub_url`	string	`https://hub.swarmfile.com`	`SWARMFILE_HUB_URL`	Hub API bas
`org_id`	string	*(none)*	`SWARMFILE_ORG_ID`	Multi-tenant
`repository_id`	string	*(none)*	`SWARMFILE_REPOSITORY_ID`	Scope every headless/se
`branch`	string	the repository's default branch (each repository names it at creation — commonly `main`)	`SWARMFILE_BRANCH`	Branch this
`office_id`	string	`default`	`SWARMFILE_OFFICE_ID`	Peer-discov `set <offic` discovery in requires a re
`oidc_issuer`	string	`https://id.swarmfile.com`	`SWARMFILE_OIDC_ISSUER`	OIDC issuer
`oidc_client_id`	string	`swarmfile-desktop` on a packaged install	`SWARMFILE_OIDC_CLIENT_ID`	OIDC client to `swarmfi` appear to d
`site_url`	string	*(install config)*	`SWARMFILE_SITE_URL`	Public site b share URLs) the installer note below)
`updater_endpoint`	string	*(install config)*	—	Update-mar installer's re
`encryption_repository_id`	string	falls back to `repository_id` when encryption is enabled and no `SWARMFILE_ENCRYPTION_KEY` is supplied	`SWARMFILE_ENCRYPTION_REPOSITORY_ID`	Repository
`read_only`	bool	`false`	`SWARMFILE_READ_ONLY`	Force a read
`session_kind`	`owner` | `member` | `guest`	`owner`	`SWARMFILE_SESSION_KIND`	Active sessi badge).
`hub_only`	bool	`false`	`SWARMFILE_HUB_ONLY`	Disable all P from the hu
`seed_mode`	bool	`false`	`SWARMFILE_SEED_MODE`	Run as a NA to cloud sto
`ec_lan_placement`	bool	`false`	`SWARMFILE_EC_LAN_PLACEMENT`	Deliberately
`lan_from_office`	bool	`false`	`SWARMFILE_LAN_FROM_OFFICE`	Treat same- unavailable. block mDNS nothing, so to spread ac members sh via the same above.
`node_auth`	bool	`false`	`SWARMFILE_NODE_AUTH`	For orgs tha the hub con authoritative to use until enforces no through this
`sync_ignore_enabled`	bool	`true`	`SWARMFILE_SYNC_IGNORE_ENABLED`	Whether `.g` sync (the gi **rules"** esca listings, exc that wants e
`iroh_bind_addr`	string	*(OS-assigned ephemeral)*	`SWARMFILE_IROH_BIND_ADDR`	Fixed UDP b Defaults to firewall requ settable, inc
`throttle_enabled`	bool	`false`	`SWARMFILE_THROTTLE_ENABLED`	Enable banc explicit site so set `site_down` to the real li
`site_download_bandwidth_mbps`	number	*(unset)*	`SWARMFILE_SITE_DOWNLOAD_BANDWIDTH_MBPS`	Download b
`site_upload_bandwidth_mbps`	number	*(unset)*	`SWARMFILE_SITE_UPLOAD_BANDWIDTH_MBPS`	Upload banc
`office_hours_start`	string	`08:00`	`SWARMFILE_OFFICE_HOURS_START`	Office-hour to `office_`

Key	Type	Default	Env override	What it doe
				settable per `hours-star`
`office_hours_end`	string	`18:00`	`SWARMFILE_OFFICE_HOURS_END`	Office-hour `office_ho` hours windo
`office_hours_wan_pct`	number	`20`	`SWARMFILE_OFFICE_HOURS_WAN_PCT`	WAN bandw
`off_hours_wan_pct`	number	`80`	`SWARMFILE_OFF_HOURS_WAN_PCT`	WAN bandw
`cache_max_bytes`	number	`10737418240` (10 GiB)	`SWARMFILE_CACHE_MAX_BYTES`	Local block-
`readahead_buffer_secs`	number | null	*(unset — each file type's own default; 15 s for video containers)*	`SWARMFILE_READAHEAD_BUFFER_SECS`	Read-ahead playback. A file type's d containers, **Settings →**
`changeset_idle_secs`	number	`300`	`SWARMFILE_CHANGESET_IDLE_SECS`	Idle seconds disables gro
`agent_checkpoint_idle_secs`	number	`30`	`SWARMFILE_AGENT_CHECKPOINT_IDLE_SECS`	Quiet secon checkpoints
`attr_cache_secs`	number	`2`	`SWARMFILE_ATTR_CACHE_SECS`	macOS and Windows/W 5–60s defa
`mergetool_cmd`	string	*(unset)*	`SWARMFILE_MERGETOOL_CMD`	Command t `tool` runs, the same `$` convention swarmfile us swarmfile fa `tool` — any nothing here
`mergetool_name`	string	*(unset)*	`SWARMFILE_MERGETOOL_NAME`	A cosmetic in messages also set; it c git config.
`update_policy`	string	*(unset)*	*(none)*	`"pinned"` for or install `repair` ref user value c
`data_namespace`	string	*(unset)*	`SWARMFILE_DATA_NAMESPACE`	Per-install d production) something y selection ke
`auto_open_changelist_for_preference`	bool	`false`	`SWARMFILE_AUTO_OPEN_CHANGELIST_FOR_PREFERENCE`	Commit-mo changelist," have to rem overrides ar switch. Off l open chang
`deferred_namespace`	bool	`true`	`SWARMFILE_DEFERRED_RENAME` (alias `SWARMFILE_DEFERRED_NAMESPACE`)	Apply non-c deletes of c confirmatio `false` to o env value is env spelling
`mount_point`	string	`~/Swarmfile` (Unix), `S:` (Windows)	`SWARMFILE_MOUNT_POINT`	Where to m on Windows configure th opening add
`mount_point_auto`	*(managed)*	—	—	**Do not set.** selected for a letter the

> **Additional mounts aren't configured here.** One engine process can hold several full mount points open at once against the same repository/org — see `swarmfile mounts` — but that's an entirely runtime, CLI/Desktop-App-driven operation with no config-file or env-var equivalent: there's no `mount_point_2` key or similar, and no way to declare a second mount up front for the engine to open at boot.

> `mount_point` / `mount_point_auto` above describe the boot mount specifically, whether or not anything else is opened later in the session. You don't need to edit this file to get a free drive letter for an additional mount: `swarmfile mounts open auto` (or the Desktop App's Open mount form, which defaults to it on Windows) picks one.

A note on environment-selection keys

`hub_url`, `oidc_issuer`, `oidc_client_id`, `site_url`, `updater_endpoint`, `data_namespace` and `update_policy` are handled differently from every other key in the table above.

A packaged install (the macOS `.app`, the Windows installer, the Linux `.deb`) ships its own read-only install config that sets these — that's how an install stays on the environment it was packaged for, and why one signed binary set can serve different environments without drifting. When the install config sets one of these seven keys, **it wins over whatever you put in your per-user `config.json`** — a hand-edit of `hub_url` or `oidc_issuer` at the per-user path documented above will silently appear to do nothing on a normal packaged install.

Your per-user value is only used as a fallback for a key the install config left unset — which is the normal case for a from-source/dev build. Two keys never take a per-user value at all: `data_namespace` and `update_policy` come only from the install config. Every other key in this file follows the ordinary env-var-then-file precedence at the top of this page; only these seven are install-config-first.

Settings that are environment-variable-only

Many settings can be tuned via `SWARMFILE_*` env vars but are **not** readable from `config.json` — the file supports only the keys in the table above. Environment Variable Index is the full list; the notable ones include:

- **Credentials:** `SWARMFILE_OIDC_REFRESH_TOKEN` (non-interactive OIDC), `SWARMFILE_API_KEY` (repository-scoped headless key — takes precedence over a refresh token), `SWARMFILE_ENCRYPTION_KEY`, `SWARMFILE_SWARM_KEY`. Credentials are deliberately kept out of the file.
- **Paths & identity:** `SWARMFILE_CACHE_DIR`, `SWARMFILE_IROH_DIR`, `SWARMFILE_USER_ID`, `SWARMFILE_MACHINE_ID`.
- **Encryption toggle/mode:** `SWARMFILE_ENCRYPTION_ENABLED`, `SWARMFILE_EC_ENABLED` (`on` / `off` / `auto`). Leave `SWARMFILE_ENCRYPTION_ENABLED` unset: any value other than `true` / `1` makes the engine upload new blocks unencrypted, even into a managed or end-to-end encrypted repository.
- **Networking:** `SWARMFILE_IROH_PUBLIC_ADDR`, `SWARMFILE_HEALTH_PORT`. `SWARMFILE_MDNS_INTERFACES` (comma-separated interface names) restricts LAN discovery to exactly those interfaces and skips the built-in VPN/virtual-interface filter. `SWARMFILE_MDNS_EXCLUDE_INTERFACES` removes more interfaces from it. See Network Requirements. (`SWARMFILE_IROH_BIND_ADDR` / `iroh_bind_addr` **is** file-settable — see the table above.)
- **QoS:** `SWARMFILE_DSCP_INTERACTIVE`, `SWARMFILE_DSCP_BACKGROUND`, `SWARMFILE_SITE_DOWNLOAD_BANDWIDTH_MBPS` / `SWARMFILE_SITE_UPLOAD_BANDWIDTH_MBPS`. The office-hours schedule keys (`SWARMFILE_OFFICE_HOURS_START` / `_END`, `SWARMFILE_OFFICE_HOURS_WAN_PCT`, `SWARMFILE_OFF_HOURS_WAN_PCT`) **are** file-settable — see `office_hours_*` / `off_hours_wan_pct` in the table above.
- **LAN shard placement tuning:** `SWARMFILE_EC_LAN_PLACEMENT_REPLICATION` (default `3`) is the replication factor K — each erasure shard is deliberately assigned to this many LAN peers. `SWARMFILE_EC_LAN_PLACEMENT_POLL_INTERVAL` (default `300`) is the reconciler's safety-net pass interval in seconds, run alongside the reactive pass that fires on a peer-roster change. Both only matter when `ec_lan_placement` is on.
- **Seed tuning:** `SWARMFILE_SEED_POLL_INTERVAL` (default `60`, seconds), `SWARMFILE_SEED_UPLOAD_CONCURRENCY` (default `8`, concurrent cloud uploads).
- **Runner (headless CI):** `SWARMFILE_RUNNER_MODE` — watches `SWARMFILE_BRANCH` for commits/tags matching a rule in `.swarmfile/runner.yml` and runs it. Env-only by design, no `config.json` key or Desktop App toggle — see Runner (Headless CI). Mutually exclusive with `seed_mode`.
- **Streaming an upload in progress:** `SWARMFILE_STREAM_BUFFER_SECS`, `SWARMFILE_STREAM_READ_WAIT_SECS`, `SWARMFILE_STREAM_WAIT_CONCURRENCY` — env-only, no `config.json` key.

  `STREAM_BUFFER_SECS` (default `30`) is how many **seconds** of playback a `fetch --follow` job keeps buffered ahead of the reader; it is seconds rather than bytes because 30 s of a proxy and 30 s of a full-resolution master are wildly different byte counts and one figure would be wrong for both.

  `STREAM_READ_WAIT_SECS` (default `15`) is how long a read of a not-yet-uploaded range waits before giving up — see below.

  `STREAM_WAIT_CONCURRENCY` (default `4`, clamped 1–64) caps how many reads may be *waiting* at once, so a burst returns the retryable "try again" promptly instead of parking every operation thread; it is sized when the mount starts, so a change needs a remount.
- **Relay policy:** `SWARMFILE_ALLOW_RELAY_FETCH` (default `false`; enable on both serving and fetching engines to let a relay-only peer carry block bytes; those bytes are throttled) and `SWARMFILE_RELAY_PATH_GRACE_MS` (default `800`; how long a connected peer gets to upgrade to a direct path before it is classified as relay).
- **Peer-to-peer budgets:** `SWARMFILE_IROH_PEER_BUDGET_SECS` (default `30`) and `SWARMFILE_IROH_PEER_BUDGET_BACKGROUND_SECS` (default `10`) cap how long a peer connection attempt may take for interactive and background work respectively.
- **Durable create queue tuning:** `SWARMFILE_CREATE_DISPATCH_CONCURRENCY` (default `8`, 1–64 — creates sent to the hub at once), `SWARMFILE_PENDING_CREATE_WAIT_SECS` (default `8`, 0–120 — how long a lock/ACL/share/comment on a not-yet-confirmed file waits before answering `409 pending_create`), and the coalescing/batching knobs (`SWARMFILE_CREATE_COALESCE`, `_HOLD_SECS`, `_HOLD_UPLOADING_SECS`, `_BATCH`). See Coding Agents.
- **Worksharing refusal mode:** `SWARMFILE_WORKSHARING_FAIL_CLOSED` (Windows only) makes a native worksharing open refuse when the hub can't be reached.
- **mDNS service type:** `SWARMFILE_MDNS_SERVICE_TYPE` overrides the `_swarmfile._udp.local.` service type, to keep separate groups of machines on one LAN from discovering each other.
- **Upload diagnostics:** `SWARMFILE_UPLOAD_TRACE=1` promotes per-attempt upload logging to `warn`, including the hub's status and response body when a block upload or commit is refused — the first thing to set when a save isn't landing and you need to know why. Off by default; the lines are noisy for ordinary work.
- **Misc:** `SWARMFILE_XREF_PREFETCH`, `SWARMFILE_DOCTOR_PERIOD_SECS`.

How streaming an in-progress upload behaves

Always on: a file that has never finished uploading can be opened through the mount — it appears at its full eventual size and reads work, waiting briefly where the bytes have not landed.

Worth knowing how it behaves, because a read into a range that has not been uploaded yet has to wait, and this runs on a filesystem operation thread. An operation that waits too long does not stall one read — it takes the whole drive with it.

The wait is therefore strictly bounded (`SWARMFILE_STREAM_READ_WAIT_SECS` , default 15 s, and `0` does not mean "forever" — it falls back to the default), and a read that outruns the upload returns a retryable "not yet" rather than an I/O error, because an application that sees an I/O error part way through a file will usually conclude the file is damaged and discard your document.

Different NLEs and DCC tools react differently to a delayed read; the bound is tuned to release the drive before an editing application gives up on it. Two consequences worth knowing: only so many reads may be *waiting* at once (a burst beyond that gets the "try again" immediately rather than parking every operation thread), and only a file with **no** committed version streams — a re-save of an existing file is readable once it commits, never mid-save.

Scope worth knowing:

- It applies **only to files with no committed version yet** — a new file landing for the first time. A file you are already reading never changes size or content because somebody started a new save.
- A file that *is* streaming reports the size it is going to be, not `0` . Without that, nothing could read it: no application asks for bytes past the end of an empty file.
- If the uploading machine is on the same **LAN**, it serves those blocks directly from its own disk, before they reach the cloud. Over a WAN it does not — that would carry every block twice on the same contended uplink.

And the limit that no setting changes: this decides **which** bytes arrive first, not how fast the link is. A 4 TB file on a 100 Mbit/s uplink delivers about 12 MB/s however well it is ordered.

Repository-local files (`.swarmfile/`)

Some configuration lives **inside a repository** — committed like any other file, so it versions with the branch and every contributor gets the same behavior — instead of in the engine's `config.json` :

File	Keys	What it does
`.swarmfile/runner.yml`	rule list	Headless CI rules — see Runner (Headless CI).
`.swarmfile/merge.yml`	`diff3: true | false`	Repository default for `swarmfile merge --diff3` and the tray's "Auto-resolve conflicts": on a merge conflict, try a three-way text merge of each conflicting file before giving up. `true` turns it on by default; `false` turns it off and the CLI/tray name this file as the reason. Absent means off (auto-resolve stays opt-in). An explicit `--diff3` / `--no-diff3` always wins over the file.
`.swarmfile/cache.yml`	`caches:` list of `name` , `env` , `subdir`	Tool-cache redirects for `swarmfile run` : each entry sets the `env` variable a tool reads for its cache location (e.g. `PNPM_HOME` , `CARGO_HOME`) to this repository's shared machine-local scratch directory plus `subdir` . Every mount of the repository on this machine shares those directories, so a second mount does not re-download what the first already fetched. A bad entry is dropped with a warning, never the whole file. See `scratch` .
`.swarmfile/publicignore.yml`	`version` , `paths` , `derived.ci_logs`	Path globs (gitignore-style, with `!` re-includes) kept out of a release's public view; `derived.ci_logs` sets CI-log exposure (`member-only` by default). Read from the tree of the tag being published and enforced as the release is built. See Publishing Releases.

The runner, merge and cache files are read from the branch the operation runs against, so they can differ per branch; `.swarmfile/publicignore.yml` is read from the tree of the tag being published. A `.swarmfile/merge.yml` that is missing or unparseable is treated exactly like `diff3: false` — never as "turn extra automatic behavior on".

For example, to make text-conflict auto-resolve the default for a repository:

```
# .swarmfile/merge.yml
diff3: true
```

Engine command-line flags

Beyond reading `config.json` , the `swarmfile-engine` binary takes a handful of one-shot flags — useful on a headless node where there's no Desktop App to click.

Answered immediately — about this binary, or a running engine:

Flag	What it does
`--status`	Engine, hub, peers, and pending work at a glance. Queries a running engine; falls back to a one-shot read of local state if none is running.
`--print-env-vars`	The resolved `SWARMFILE_*` values this build consumed, secrets redacted, with a footer naming the install-config and per-user config files that fed the merge (or the `SWARMFILE_CONFIG` path).
`--capabilities`	The compiled feature set of *this* binary — which optional features it was built with.
`--build-info`	The same compiled features plus the target architecture, in a one-line machine-readable form, then exit.
`--version` , `-V`	Print the version and exit.
`--help` , `-h`	Print the flag list and exit.

Lifecycle — against a running engine:

Flag	What it does
`--pause`	Pause sync without unmounting the drive.
`--resume`	Resume after `--pause` .
`--quit`	Ask the running engine to drain in-flight work and stop.

`--help` and `--version` answer *before* the engine reads any config, so they work on a broken or unconfigured install. For day-to-day version-control and mount operations, use the `swarmfile` CLI — it drives the same engine over its control socket — rather than these flags.

The engine's own `--help` also lists offline/pinning flags (`--pin`, `--prepare-offline`, and related). These run only while the engine is **stopped**, so they can't be driven against a live mount and aren't part of the normal headless workflow. To force a scope resident for offline or render-farm use against a *running* engine, use `swarmfile hydrate` instead.

Verifying what actually took effect

Because several sources feed the final config, print the **resolved** values the engine will use — with secrets redacted, and a footer naming the install-config and per-user files that fed the merge (or the `SWARMFILE_CONFIG` path):

```
swarmfile-engine --print-env-vars
```

`swarmfile doctor` (or the standalone `swarmfile-doctor`) then confirms the resolved hub URL, org, and credentials actually reach the hub. See CLI: swarmfile-doctor.

Example: a headless seed node

On the hosted service the hub and IdP are already the defaults, so a seed node only names its org and repository:

```
{
  "org_id": "org_abc123",
  "repository_id": "proj_xyz789",
  "seed_mode": true,
  "cache_max_bytes": 107374182400
}
```

(On a **self-hosted or Enterprise** hub, add `hub_url`, `oidc_issuer`, and `oidc_client_id` for your own domain.)

Pair this with a credential in the environment — `SWARMFILE_API_KEY=sf_key_…` for a repository-scoped key, or `SWARMFILE_OIDC_REFRESH_TOKEN=…` — since credentials aren't stored in the file. See Deployment Topologies and Self-Hosted Seed Nodes for the full headless-node walkthrough, Network Requirements for exactly what a machine using this file needs to reach, and Deploying to Your Team for pre-staging this file across a fleet.

Network Requirements

What Swarmfile actually talks to on the network, so you can configure a firewall allowlist before rollout instead of discovering it by trial and error. Everything below is what the desktop client (engine + Desktop App) initiates outbound — nothing here requires an inbound port opened on your network firewall for ordinary use. Host firewalls on each machine are a separate matter: LAN peer discovery needs inbound UDP from the local subnet, which the Windows installer allows for you (see LAN peer discovery).

Summary

Purpose	Protocol	Port	Host(s)	Required?
Control plane — metadata, auth, commits, presigned URLs, live sync	HTTPS	443	`hub.swarmfile.com`, or your own hub if self-hosted	Always
Identity — sign-in, token refresh	HTTPS	443	`id.swarmfile.com`, or your own IdP if you bring your own SSO	Always
File content — block upload/download	HTTPS	443	the org's storage origin: `*.r2.cloudflarestorage.com` by default, or the org's dedicated/bucket endpoint (a customer S3-compatible endpoint under BYOS)	Always — this is the actual file data; the hub only mints short-lived presigned URLs
Installer downloads	HTTPS	443	`releases.swarmfile.com`	Install and repair only
git-LFS agent installer	HTTPS	443	`get.swarmfile.com`	Only while installing/updating the git-LFS agent
Auto-update check	HTTPS	443	`updates.swarmfile.com`	Whenever the Desktop App is running
Peer-to-peer file transfer	QUIC (UDP)	OS-assigned by default	direct to other peers' IPs (relay-only peers are skipped by default)	Only when P2P is enabled (the default; off under cloud-only mode)
P2P relay and peer discovery (NAT traversal)	HTTPS/443 + UDP/7842	third-party hosts — see below	Only when P2P is enabled; the relay is used for registration/address lookup, not block data by default	
LAN peer discovery	mDNS, UDP 5353 multicast	local subnet	Only when P2P is enabled	
Dashboard / web app	HTTPS	443	`swarmfile.com`	User-initiated browsing only, opened in your default browser — not the engine or Desktop App process

Which exact hub/identity/releases/updates hostnames a given install uses is a per-install `config.json` setting, not something baked into the binary at compile time — a fresh install on the standard hosted plan defaults to the prod hosts above; check that file on a specific machine if you need to confirm. See Engine Config File.

The self-hosted edition of the Desktop App contacts none of the swarmfile.com hosts above: it talks to your own server, to your other machines, and to the peer-to-peer relay and discovery hosts below unless your server turns peer-to-peer off. Windows Setup may also download the WebView2 runtime from Microsoft.

Control plane and identity

The engine talks to one hub host for control-plane work: browsing and editing metadata, commits, minting the short-lived presigned URLs for block transfer, and a live WebSocket (`wss://`, same host) that pushes changes from other users in real time.

File content does not flow through the hub. For a read the hub answers with a redirect to a presigned URL and the client fetches the block bytes directly from the org's storage origin — `*.r2.cloudflarestorage.com` on the hosted default, a jurisdiction-pinned endpoint, or a customer-supplied S3-compatible endpoint under BYOS; uploads are presigned the same way.

Sign-in and token refresh go to a separate identity host — the built-in IdP by default, or your own OIDC provider if you've configured bring-your-own SSO (see Identity).

Both are plain HTTPS on port 443. No other port is used for control-plane or identity traffic.

git-LFS

On a repository used as a git-LFS server, `git lfs` talks to the hub host for the Batch API, locks, and the transfer endpoint. On a plaintext repository the stock `basic` transfer (and a push ≥ 64 MiB through the desktop agent) uploads to the storage origin directly via a presigned URL, same as block traffic; on a managed repository only the transfer agent's chunked block uploads are accepted (the server mints presigned block URLs, or the agent falls back to the hub). Downloads stream through the hub. The git-LFS agent installer is fetched once from `get.swarmfile.com` (see the summary table). Everything stays on 443.

Peer-to-peer file transfer

By default, machines in the same office exchange file data directly over QUIC (UDP) rather than always routing through the cloud — this is what makes LAN-first fast. The engine binds to an OS-assigned ephemeral UDP port for this; it does not listen on a fixed, predictable port unless you explicitly set one (`"iroh_bind_addr"` in `config.json`, or `SWARMFILE_IROH_BIND_ADDR` in the environment — an operator choice for a specific deployment, not a default).

Peer-to-peer relay and discovery

Two pieces of this are third-party infrastructure Swarmfile doesn't operate itself, run by Number Zero (the maintainers of the P2P transport Swarmfile builds on):

- **Relay and address discovery**, used when two peers can't establish a direct connection (e.g. both behind restrictive NATs): a handful of regional relay hosts (`*.relay.n0.iroh.link`), reached over HTTPS/443 with QUIC address-discovery on UDP/7842. The relay is a **transport** fallback: by default the engine does not move block bytes over it. A peer reachable only through the relay is skipped, and the read uses the hub (`SWARMFILE_ALLOW_RELAY_FETCH=1` opts in on both serving and fetching engines; those bytes are throttled).
- **Peer discovery**, used to look up how to reach another peer: `dns.iroh.link`, over HTTPS and plain DNS.

If your firewall policy needs to name every host P2P might touch, these are worth listing alongside your own hub domain — though in practice, if UDP/QUIC is blocked entirely (common on locked-down corporate networks), the simpler answer is usually cloud-only mode rather than trying to allowlist P2P traffic through it.

LAN peer discovery

Finding peers on the same local network uses standard mDNS (multicast DNS, the same mechanism as Bonjour/AirPlay discovery) — no custom protocol, no non-standard port.

If your network already blocks mDNS multicast (some corporate/VLAN setups, and Docker bridges, do this by default), Swarmfile still works — it just can't discover LAN peers automatically, and every read falls back to the cloud; a relay-only peer is skipped by default (see Peer-to-peer file transfer). There's a separate `lan_from_office` setting for networks that block multicast but where you still want deliberate LAN peer placement; see Deployment Topologies.

mDNS runs only on physical LAN interfaces (Ethernet, Wi-Fi). The engine skips VPN and tunnel adapters (macOS `utun*`, WireGuard, OpenVPN, Tailscale, ZeroTier), Hyper-V/WSL, VMware, VirtualBox, Parallels and Docker virtual networks, and Apple's AirDrop links (`awdl0`, `llw0`), because none of them reach a colleague's machine on the office LAN. It advertises only its LAN addresses, never a VPN address.

It checks interfaces again every 30 seconds, so connecting a VPN later doesn't move discovery onto it. To override the choice, set `SWARMFILE_MDNS_INTERFACES` (use only these interfaces, comma-separated) or `SWARMFILE_MDNS_EXCLUDE_INTERFACES` (never use these). `swarmfile doctor`'s `mdns listen` check lists the interfaces in use and those it skipped. It warns when no LAN interface is available.

Windows Firewall. Windows filters *inbound* traffic per program. If `swarmfile-engine.exe` has no inbound rule, other machines' mDNS queries and announcements never reach it, but its own outgoing packets still leave. The result is one-sided: other machines may briefly see this one, and this one never finds anyone. Windows only offers to create a rule when someone runs the engine interactively, and then only for the Public profile, so silent/GPO installs and standard users would otherwise get nothing.

The installer (both the bundled `Setup.exe` and the bare `.msi`) therefore adds one inbound rule, and removes it on uninstall:

Setting	Value
Display name	`Swarmfile - local network file sharing`
Group	`Swarmfile`
Program	`C:\Program Files\Swarmfile\swarmfile-engine.exe`
Direction / action	Inbound, Allow
Protocol / ports	UDP, any local port (mDNS on 5353, plus the QUIC port, which is ephemeral unless you pin it with `iroh_bind_addr`)
Remote addresses	Local subnet only
Profiles	Domain, Private (not Public)

A second rule in the same group, `Swarmfile - local network check (doctor)`, does the same for `swarmfile-doctor.exe`, so its one-second `mdns listen` check can hear replies when no engine is running. To include the Public profile as well, install with `msiexec /i Swarmfile-x64.msi SWARMFILE_FW_PROFILES=7 /qn` (the value is a bitmask: 1 = Domain, 2 = Private, 4 = Public). Check the result with:

```
Get-NetFirewallRule -Group Swarmfile | Format-Table DisplayName, Enabled, Profile, Direction, Action
Get-NetFirewallRule -Group Swarmfile | Get-NetFirewallAddressFilter   # RemoteAddress: LocalSubnet
```

If you deploy from the `.zip`, run the engine from a different path, or your GPO sets **Apply local firewall rules: No** (which ignores rules an installer adds), create the equivalent rule yourself, through GPO/Intune or with:

```
New-NetFirewallRule -DisplayName "Swarmfile - local network file sharing" -Group "Swarmfile" `
  -Direction Inbound -Action Allow -Protocol UDP `
  -Program "C:\Program Files\Swarmfile\swarmfile-engine.exe" `
  -RemoteAddress LocalSubnet -Profile Domain,Private
```

To avoid a program-scoped rule, pin the QUIC port (`"iroh_bind_addr": "0.0.0.0:4433"` in `config.json`) and open UDP 5353 and 4433 from `LocalSubnet` with `-LocalPort 5353,4433` instead of `-Program`. A **Block** rule for `swarmfile-engine.exe` (created when someone clicked Cancel on a firewall prompt) overrides any allow rule, so delete it if one exists. `swarmfile doctor`'s `mdns listen` check warns when the hub lists peers in your office or on your subnet but mDNS has found none of them. That usually means a firewall or a network that filters multicast.

Linux firewalls

The `.deb` doesn't change your firewall. On a host running ufw or firewalld with a default-deny inbound policy, allow UDP 5353 (mDNS) and the engine's QUIC traffic from the local subnet. QUIC uses a random UDP port unless you pin it with `iroh_bind_addr`, so you can either allow all UDP from the subnet or pin the port and open only that one. The examples use `192.168.1.0/24`. Use your own subnet, which `ip -4 addr` shows (an address of `192.168.1.23/24` is on `192.168.1.0/24`).

The tidy option is to pin the port. Add `"iroh_bind_addr": "0.0.0.0:4433"` to `config.json`, restart the engine (`systemctl --user restart swarmfile-engine`), and then:

```
# ufw
sudo ufw allow proto udp from 192.168.1.0/24 to any port 5353 comment 'Swarmfile mDNS'
sudo ufw allow proto udp from 192.168.1.0/24 to any port 4433 comment 'Swarmfile QUIC'

# firewalld
sudo firewall-cmd --permanent --add-rich-rule='rule family="ipv4" source address="192.168.1.0/24" service name="mdns" accept'
sudo firewall-cmd --permanent --add-rich-rule='rule family="ipv4" source address="192.168.1.0/24" port port="4433" protocol="udp" accept'
sudo firewall-cmd --reload
```

If you leave the port unpinned, allow all UDP from the subnet instead:

```
# ufw
sudo ufw allow proto udp from 192.168.1.0/24 to any comment 'Swarmfile LAN'

# firewalld
sudo firewall-cmd --permanent --add-rich-rule='rule family="ipv4" source address="192.168.1.0/24" protocol value="udp" accept'
sudo firewall-cmd --reload
```

Some stock rule sets already allow part of this. Ubuntu's default ufw rules accept multicast mDNS, and Fedora Workstation's default firewalld zone allows mDNS and UDP ports above 1024. The explicit rules do no harm there. They also cover the unicast replies that some mDNS responders send. If your LAN uses IPv6 only, add the same rules for your IPv6 prefix.

`swarmfile doctor`'s `host firewall` check reads these firewalls without root and warns when one is active with no rule that allows UDP 5353 and the QUIC port. ufw only shows its rules to root, so as a normal user the check can't confirm them. In that case it warns only when `mdns listen` also reports missing LAN peers.

macOS

The macOS Application Firewall allows the signed engine by default. Two settings stop LAN peers from reaching it: **Block all incoming connections** (System Settings → Network → Firewall → Options…), and a *Block incoming connections* entry for `swarmfile-engine`, which appears when someone clicks Deny on the firewall prompt. The `host firewall` check reports both and gives the command that reverses each one.

On macOS 15 and later, **Local Network privacy** also gates LAN access. The Desktop App asks for it, and the engine installed inside the app is covered by that permission (System Settings → Privacy & Security → Local Network).

An engine binary running outside the app bundle, such as a render-farm or CI node, has no such permission. Unless it runs as root (a LaunchDaemon) or in the foreground of a Terminal or SSH session that stays open, macOS can deny it LAN access without telling anyone. Connections to LAN peers or a LAN hub then fail with "No route to host", and mDNS finds nobody, while internet traffic keeps working. The `macos local network` check warns when the engine runs this way. Apple's TN3179: Understanding local network privacy lists these rules.

Cloud-only mode (hub-only)

For a network that blocks UDP/QUIC outright, or a security policy that wants zero peer-to-peer connections and zero peer IP exposure, **cloud-only mode** turns P2P off entirely — not just "prefers not to use it." (The engine and config file call it `hub_only`; the Desktop App and CLI label the same switch *Cloud-only mode*.)

With it enabled, the engine never opens a QUIC endpoint, never starts mDNS, and never dials another peer.

The entire network surface collapses to the HTTPS/443 rows in the summary table above — metadata to the hub, file bytes to the org's storage origin (control plane, identity, storage, and — while the Desktop App is active — installer/update checks), so a firewall must allow both, not just the hub.

The engine creates no UDP listener, mDNS probe, or peer traffic in this mode; DNS resolution may still use UDP 53.

This is set per-machine (`"hub_only": true` in `config.json`, or `SWARMFILE_HUB_ONLY=1` in the environment), or centrally for the whole org by an owner or admin (Settings → Network). The two combine with OR, not override: an org-enforced policy can only ever turn cloud-only **on** for a machine, never force it back **off** — a machine an operator has locally pinned to cloud-only stays cloud-only regardless of what the org policy says. See Security for how this fits the broader threat model, and Operations for where the setting shows up in your audit log.

A self-hosted server operator can also narrow or disable the swarm for the whole team, server-side: `SWARMFILE_SERVER_P2P=sealed` (no cleartext over peers) or `off` (no peer-to-peer at all). Engines follow that posture on their periodic check — see Self-hosting → Peer-to-peer.

Verifying what a specific machine actually needs

`swarmfile doctor` runs a battery of connectivity checks — DNS resolution, HTTPS reachability to the hub, an authenticated round-trip, a QUIC dial test (skipped cleanly when cloud-only mode is on), and an mDNS listen test (same) — and reports pass/warn/fail for each, with a `--json` mode for pasting into a firewall-exception ticket. Run it from a machine on the network you're configuring rather than guessing; see swarmfile-doctor.

What's not covered here

Self-hosted seed and NAS nodes participate in the same P2P swarm as any other machine — no additional inbound ports are required for them either. A seed node run as a headless cloud service (rather than on-prem hardware) can optionally expose a health-check HTTP endpoint for your own orchestration (Kubernetes, Fly.io, etc.) — that's an explicit opt-in for that specific deployment shape, not something a normal desktop or on-prem seed install does, and it's the one case where the engine listens on all interfaces rather than loopback-only.

Even then, only `/livez` (a bare `{"status":"ok"}`) answers a non-loopback caller — `/healthz`'s detailed body, which exposes sensitive operational detail, 403s for anything but loopback regardless of which interface the port is bound to, so a container orchestrator's liveness probe works but a remote caller can't use the same port to enumerate your fleet.

Talk to us if you're deploying that way and need the specifics.

Storage Format

This page specifies how Swarmfile stores a file's bytes in object storage: how a file is split into blocks, how blocks are named, how the list of blocks (the *manifest*) is encoded, how blocks are encrypted, and where they land in a bucket. It is written so that someone holding the stored objects (and, for an encrypted repository, the repository key) can reconstruct a file with no Swarmfile software involved. It describes the format the current code writes and reads. Where something isn't pinned down, this page says so rather than guessing.

It covers **file content only**. File names, folder structure, versions, branches, commits, ACLs, and comments live in the repository's metadata on the hub, not inside blocks, and their formats aren't specified here. In practice you need that metadata to know which manifest belongs to which path: each file entry records its manifest's identifier as `rootCid`.

If you only want your files back, you don't need this page: copy them off the mount, or use Branch Mirror to export real files to your own bucket. See Data Portability & Offboarding.

Overview

- A file is split into **chunks**, usually with content-defined chunking (FastCDC) averaging 1 MiB.
- Each chunk is stored as one **block**, named by its **CID**: the BLAKE3 hash of the chunk's *plaintext* bytes.
- A **manifest**, a small JSON document listing the chunks in file order, is itself stored as a block with its own CID. That CID is the file entry's `rootCid`.
- A very large file's manifest is split into **segment** blocks under a small **segmented root** (manifest format v2).
- On encrypted repositories, every block, manifests included, is stored as AES-256-GCM ciphertext under a key derived from the repository key and the block's CID. The CID is still the plaintext hash.
- Some chunks also get **Reed-Solomon 10+4 erasure-coded shards**, stored as extra blocks *in addition to* the chunk itself.

```
entry.rootCid ──► manifest block (flat v1)  ──► chunk blocks (in file order)
             └──► segmented root (v2) ──► segment blocks ──► chunk blocks
```

Chunking

Default parameters

Constant	Value
FastCDC minimum chunk	262,144 bytes (256 KiB)
FastCDC average (target) chunk	1,048,576 bytes (1 MiB)
FastCDC maximum chunk	4,194,304 bytes (4 MiB)
Fixed-size chunk (when fixed chunking is used)	1,048,576 bytes (1 MiB)

The CDC implementation is FastCDC "v2020" from the Rust `fastcdc` crate (3.2.x), with its default normalization (level 1) and gear table. A compatible implementation in the hub uses the same parameters, so CIDs dedupe across both.

You never need to re-run the chunker to read a file. The manifest records every chunk's offset and size, so a reader just follows the manifest. The chunking parameters only matter if you want to reproduce Swarmfile's CIDs for new data.

File-type profiles

On the mount's write path, the chunking strategy depends on the file extension (case-insensitive):

Extensions	Strategy
everything not listed below (incl. `.rvt`, `.dwg`, `.max`, `.ifc`, `.drp`, `.settings`, `.prproj`)	CDC 256 KiB / 1 MiB / 4 MiB
`.pdf`, `.dwf`	Fixed 1 MiB
`.slog`, `.rws`, `.dat`, `.xml`, `.avb`, `.avp`, `.msn`, `.pmsm`, `.nksm`	Fixed 1 MiB, plus the small-file fast path

Small-file fast path: a file with one of the fast-path extensions that is smaller than 50 KiB (51,200 bytes) is stored as a **single chunk**, with the strategy recorded as `fixed` and `chunk_size` equal to the file's size (minimum 1).

The profiles also set read-ahead behavior. That doesn't affect the stored format.

Other writers can pick other strategies. For example, `swarmfile-seed` uses fixed-size chunks unless you pass `--cdc`, and small browser uploads are written as a single chunk. Whichever strategy was used is recorded in the manifest's `chunking` field. The per-extension table above may change. It is not part of the stable format.

Block identifiers (CIDs)

A CID is the **BLAKE3-256 hash** of the block's **plaintext** bytes, written as **64 lowercase hex characters**. There is no multihash or multibase prefix, no version byte, and no domain separation.

Block kind	CID is BLAKE3 of…
Chunk	the chunk's plaintext bytes
Flat manifest	the manifest's exact serialized JSON bytes
Manifest segment / segmented root	that block's exact serialized JSON bytes
Erasure shard	the shard's bytes as stored (see Erasure coding)

Because a CID hashes plaintext, encrypting a block never changes its CID, and identical content within a repository always has one CID. Encryption uses a fresh random nonce every time, so the same plaintext encrypted twice gives different stored bytes. Dedup happens at the CID level, not the ciphertext level.

Verification rule: after you fetch a block (and decrypt it, if it's encrypted), `blake3(plaintext)` must equal the CID you asked for. The only exception is an erasure shard, which is verified against its stored bytes.

Manifests

Manifests are JSON as written by Rust's `serde_json::to_vec`: compact, with no whitespace and fields in the order shown below. Because a manifest's CID is the hash of its exact bytes, **re-serializing a manifest with different formatting or key order gives a different CID**. To *read* a manifest, any JSON parser works. To *reproduce* a CID, you have to match the serialization byte for byte.

Flat manifest (format v1)

Used whenever the whole manifest fits in one block, which covers almost every file. It has no version field. Version 1 is implicit.

```
{
  "total_size": 3355443,
  "chunk_size": 1048576,
  "chunks": [
    {"cid": "9f1c…e2a0", "offset": 0,       "size": 1210334},
    {"cid": "4b7d…0c19", "offset": 1210334, "size": 862117},
    {"cid": "e03a…77f4", "offset": 2072451, "size": 1282992}
  ],
  "chunking": {"type": "cdc", "min": 262144, "avg": 1048576, "max": 4194304}
}
```

(Formatted and with CIDs shortened for readability. Real CIDs are 64 hex characters, and stored bytes have no whitespace.)

Field	Type	Meaning
`total_size`	u64	File length in bytes. Equals the sum of every chunk's `size`.
`chunk_size`	u32	For `fixed`: the chunk size. For `cdc`: the *target average*, not a real chunk size. Kept for backward compatibility. Use `chunking` and each chunk's `size` instead.
`chunks`	array of ChunkRef	Chunks in file order. An empty file has `[]`.
`chunking`	object	How the file was chunked (see below). If absent, readers treat it as `{"type":"fixed","chunk_size":1048576}`.

`chunking` is an internally tagged object:

- `{"type": "fixed", "chunk_size": N}`
- `{"type": "cdc", "min": N, "avg": N, "max": N}`

ChunkRef:

Field	Type	Meaning
`cid`	string	CID of the chunk's plaintext (64 lowercase hex).
`offset`	u64	Byte offset of this chunk in the file.
`size`	u32	Plaintext length of this chunk.
`erasure`	object, optional	Erasure-coding group for this chunk (see Erasure coding). Left out when there are no shards.

Chunks are contiguous: each chunk's `offset` equals the previous chunk's `offset + size`. The same CID can appear more than once when a file repeats content. Its block is stored once and referenced from each position.

Browser-upload shape (version 1)

Some small files uploaded through the web app may have a manifest in an older envelope, which readers still accept:

```
{"version": 1, "type": "file", "size": 35, "blocks": [{"offset": 0, "size": 35, "cid": "…"}]}
```

`blocks` maps one-to-one onto `chunks`, and `size` onto `total_size` (if `size` is missing, total the block sizes). Treat it as fixed chunking. New writes never produce this shape.

Segmented manifest (format v2)

When a flat manifest would be too big for one block (see Size thresholds), the chunk list is split into **segment** blocks, and the entry's `rootCid` points at a small **segmented root** instead. A flat manifest's bytes never change when this happens. A file is only segmented if it can't be stored flat.

Segmented root:

```
{
  "manifest_version": 2,
  "total_size": 5497558138880,
  "chunking": {"type": "cdc", "min": 262144, "avg": 1048576, "max": 4194304},
  "segments": [
    {"cid": "a41e…9d03", "first_offset": 0,          "chunk_count": 31207},
    {"cid": "07bc…41fe", "first_offset": 32730906112, "chunk_count": 31188}
  ]
}
```

Field	Type	Meaning
`manifest_version`	u32	Always `2`.
`total_size`	u64	File length in bytes.
`chunking`	object	Same as in the flat manifest, but always present here.
`segments`	array of SegmentRef	Segments in file order.

SegmentRef: `cid` (the segment block's CID), `first_offset` (file offset of the segment's first chunk, so a range read can binary-search to the segment it needs), and `chunk_count` (how many ChunkRefs the segment holds).

Segment block:

```
{"chunks": [ {"cid": "…", "offset": 0, "size": 1048576}, … ]}
```

A segment holds one contiguous slice of the ChunkRef list, in the same shape as the flat manifest's `chunks`, including any `erasure` objects. To rebuild the full chunk list, concatenate the segments' `chunks` in `segments` order. If a segment's length doesn't match its `chunk_count`, treat that as corruption.

There is one level of segmentation. The root is never nested. A file whose segmented root would itself exceed the block limit is refused at write time.

Telling the shapes apart

A root block is parsed like this, in order:

1. Parse as a **flat manifest**, which needs `chunks` and also accepts a `blocks` array. If that works, it's flat.
2. Otherwise parse as a **segmented root**, which needs `manifest_version`, `total_size`, `chunking`, and `segments`. If that works *and* `manifest_version == 2`, it's segmented.
3. Otherwise the block is an error. That includes a segmented root with any `manifest_version` other than `2`. Readers fail loudly and never guess.

The two canonical shapes are mutually exclusive: a flat manifest never has `segments` or `manifest_version`, and a segmented root never has `chunks`.

Size thresholds

Limit	Value
Maximum stored size of any manifest, segment, or root block	4,194,368 bytes (4 MiB + 64)
Target *plaintext* size per segment	¾ of that limit, 3,145,776 bytes

The flat-or-segmented decision is made on the **stored** size: after encryption on an encrypted repository, where encryption adds 29 bytes per block. Segments are packed greedily in file order. Each ChunkRef costs its compact-JSON length plus 1 byte, a new segment starts when the next ref would push the open one past the target, and a ref is never split. Readers don't need any of this. It only matters if you want to reproduce the exact segment and root CIDs for a file.

To give a sense of scale: a bare ChunkRef is about 100 bytes, so a flat manifest can hold roughly 40,000 chunks (about 40 GB at the 1 MiB CDC average), and when the list outgrows one block it splits into segments packed to the ¾ target — about 31,000 refs each. A ChunkRef carrying erasure metadata is about 10× larger, so erasure-coded files segment much sooner.

Encryption

Which repositories are encrypted

Each repository has a fixed **encryption tier**, chosen when it's created:

Tier	Blocks at rest	Who holds the repository key
`managed` (default on paid plans)	AES-256-GCM ciphertext	The hub, which stores the key wrapped by a hub-held key-encryption key (KEK) and hands it over TLS to authenticated clients that are allowed to have it
`e2e` (opt-in, Pro and above)	AES-256-GCM ciphertext	Only enrolled user devices and, if configured, the org's recovery key. The hub stores only wrapped copies it can't open.
`plaintext`	**Unencrypted**	Nobody. There's no key.

The `plaintext` tier is the only tier available on the Free (no-card) plan, and **every public repository must use it**. Every Free-plan repository (public or private) and every public-repository block is stored as plain bytes; a private Free repository's blocks are never peer-served on the hosted service. A paid-plan org that asks for `plaintext` on a private repository gets `managed` instead.

What is *not* encrypted, even on the managed tier

- **git-LFS objects on an unencrypted repository.** On the `plaintext` tier — every Free-plan repository and every public repository — LFS is plaintext by definition, whether it arrived through the transfer agent or the stock direct path.
  - On a managed repository, new uploads must use the transfer agent and become ordinary encrypted blocks with a normal manifest.
  - The old direct paths (stock `git-lfs push`, direct object PUT, presigned multipart) are refused with the typed `lfs_agent_required` error.
  - A managed repository that took direct uploads before that rule may still hold the old **one plaintext object** shape under `lfs/` until its operator-run migration reports clean; the migration writes and verifies encrypted native content before deleting the plaintext object.
  - LFS is refused outright on `e2e` repositories. See Git-LFS and Security Architecture.
- **Some hub-generated derivatives.** The hub renders previews of managed content in memory. Thumbnails, video poster frames and point-cloud preview images are then stored **encrypted** (see Hub-generated preview images). These stay unencrypted:
  - scrubbable **video proxies** (a 720p H.264 transcode) at `_videoproxy/{rootCid}`, kept up to 30 days;
  - **whole-file copies** the hub assembles to serve browser downloads and previews at `_materialized/{rootCid}`, kept up to 24 hours;
  - short-lived **render inputs and outputs** the preview containers read and write (`_videosrc/`, `_videoposter/`, `_pcpreview/`), deleted when that render finishes;
  - **bare-key preview images**: JPEGs at the bare key `{sha256}`. They aren't rewritten or deleted, and stop being used once the file's content changes.

  None of these exist for `e2e` repositories, because the hub can't read their content. On the `plaintext` tier, previews are plaintext like everything else.
- **Plaintext managed blocks.** Some managed-tier content was uploaded without envelope encryption; readers detect it by checking whether `blake3(stored bytes) == CID` before trying to decrypt (see below).

Block ciphertext layout

```
offset  length  field
0       1       version      0x02 (current), 0x01 (version 1, read-only), or 0x82–0xC0 (a rotated key generation)
1       12      nonce        random, fresh for every encryption
13      N+16    ciphertext   AES-256-GCM(plaintext), with the 16-byte GCM tag appended
```

Encryption adds exactly **29 bytes** to each block (1 + 12 + 16). There is no associated data (AAD).

The version byte also says **which generation of the repository key** sealed the block (see Key rotation): `0x01` and `0x02` mean generation 1, and `0x80 + g` means generation *g*, for *g* from 2 to 64 (bytes `0x82` to `0xC0`). Every other value is reserved, and readers reject it.

The **per-block key** is derived with HKDF-SHA256 from the 32-byte repository key of that generation:

Version byte	HKDF input key	HKDF salt	HKDF info	Output
`0x02`	generation-1 repository key (32 bytes)	the CID as its 64-character lowercase hex **ASCII string** (64 bytes, not the 32 raw hash bytes)	`swarmfile-block-encryption/v1` (ASCII)	32 bytes, the AES-256 key
`0x82` – `0xC0`	the repository key of generation `version − 0x80`	same as `0x02`	same as `0x02`	32 bytes
`0x01` (version 1)	generation-1 repository key	none (RFC 5869 default: 32 zero bytes)	the CID's hex ASCII string	32 bytes

The CID used in the derivation is the block's own CID: the chunk's CID for a chunk, and the manifest's, segment's, or root's CID for those blocks. Every block is sealed under a different key, and a block can't be decrypted under the wrong CID.

Compatible implementations exist in the engine, the hub, and the web app. The web app decrypts only E2E share content, with the single repository key carried in the share link; an E2E repository can be rotated, and a link minted before a rotation carries the pre-rotation key.

How the repository key is held (who can decrypt)

This section only explains who can decrypt. It's not needed to parse the format.

- **Managed:** the hub generates a random 32-byte repository key and stores it AES-256-GCM-wrapped under its own key-encryption key. An authorized, authenticated client receives the key over TLS; after a rotation it also receives every retained generation. The desktop engine may cache the key locally, protected by the operating system. With the keys and the stored blocks, you can decrypt the repository yourself.
- **E2E:** the client creating the repository generates the key and wraps it separately to each enrolled device's X25519 public key, and to the org's recovery public key if one is configured.

Each wrapped copy is an envelope: `[0x01][nonce:12][ephemeral X25519 public key:32][AES-256-GCM ciphertext + tag]`, where the AES key is `HKDF-SHA256(X25519(device private key, ephemeral public key), salt = nonce, info = "swarmfile-e2e-envelope/v1")`.

The org recovery private key is split into Shamir shares that are handed out when it's created and never stored by the hub. A quorum of shares recovers the repository key. See End-to-End Encryption.

Key rotation

A managed repository's key can be **rotated** by its owners or admins in **Repository settings → General**, or by the repository's creator through the API. Rotation creates a new key **generation** and makes it current. It never replaces or deletes an earlier generation:

- Every block records the generation that sealed it in its version byte. Everything written before a repository's first rotation is generation 1 (`0x01` or `0x02`), so a rotation doesn't change or re-encrypt any stored block.
- New blocks are sealed under the current generation (`0x80 + g`). Readers look up the generation the byte names and use that key. There's no trial decryption.
- Every retained generation is stored wrapped under the hub's KEK, like the generation-1 key. A KEK rotation re-wraps all of them.
- Retired generations are kept so sealed blocks stay readable; deleting one makes its blocks unreadable.

E2E repositories rotate the same way. The generation model and the version byte are identical (`0x80 + g`), and retained generations stay readable, so nothing here is managed-specific. The difference is *who holds the keys*: the hub never sees an E2E key, so a member's own device performs the rotation — it mints a new generation and re-wraps it to every remaining device and to the org recovery key — and the hub only stores the envelopes the device produced. A device that doesn't yet hold the new generation fetches it on its next check and keeps its older-generation key for existing content.

A reader that predates rotation rejects a `0x82`+ block as an unsupported version. It never decrypts one into wrong bytes. Such a reader still opens every generation-1 block, because the key response keeps generation 1 as the default key. For the same reason, an older engine keeps writing `0x02` blocks under generation 1, which every reader can open.

Why a marker in the version byte, and not trial decryption. Trying each generation's key until the GCM tag verifies would also have worked without a format change. But AES-GCM checks its tag only after processing the whole block, so every read of an older block would cost one full decryption per newer generation, and a missing key would look like a corrupt block. Putting the generation in the byte that already selects the key derivation keeps the layout and the 29-byte overhead unchanged, costs nothing on reads, and makes a missing key a specific, recoverable error. The trade-off is a limit of 64 generations per repository.

Hub-generated preview images

On a `managed` repository, the thumbnails, video poster frames and point-cloud preview images the hub generates are JPEGs stored with the block ciphertext layout and the repository key. Where a content block uses its CID, a preview uses the **SHA-256 of the JPEG bytes**, as 64 lowercase hex characters.

That id is both the HKDF salt and the object name, at `p/{repositoryId}/{sha256}` (behind the same org prefix or bucket as blocks). To read one, decrypt it like a content block (a `0x02` block, or a rotated generation's `0x82`+ block) with the SHA-256 hex standing in for the CID, then check that `sha256(plaintext)` equals that hex.

A preview written before this scheme is a plain JPEG at the bare key `{sha256}`. Readers try the scoped key first, then the bare key. On the `plaintext` tier, previews are always plain JPEGs at the bare key.

Erasure coding

Some chunks also get **Reed-Solomon** erasure coding over GF(2^8), with **10 data shards + 4 parity shards**. Any 10 of the 14 shards rebuild the chunk.

When it's applied. Erasure coding is *not* applied to every block. The engine decides for each save (`SWARMFILE_EC_ENABLED` = `on`, `off`, or `auto`, default `auto`).

In `auto` mode, shards are produced only for files of **at least 64 KiB**, and only when the machine has a peer-to-peer fabric with other peers, favoring WAN and seed peers. It's skipped when two or more LAN peers are present, unless deliberate LAN shard placement is turned on, and when WAN peers are measured as fast (p50 RTT ≤ 80 ms). Uploads with no P2P fabric (cloud-only mode, and bulk migration by default) don't produce shards.

These rules may change. A reader should just follow each ChunkRef's `erasure` field.

Shards are extra. The whole chunk block is always stored too. Shards add redundancy and are never a substitute for the chunk.

Encoding. The input is the chunk's **stored** bytes: the ciphertext envelope on an encrypted repository, or the plaintext otherwise.

1. `shard_size = ceil(len(stored) / 10)`.
2. Zero-pad `stored` to `10 × shard_size` and split it into data shards 0–9.
3. Compute parity shards 10–13 with Reed-Solomon (10, 4) over GF(2^8). The engine uses the Rust `reed-solomon-erasure` crate (`galois_8`). Its exact encoding matrix is defined by that crate and isn't restated here.
4. Each shard is stored as a block whose CID is `blake3(shard bytes)`. Shards are **not** encrypted again.

`erasure` object (on a ChunkRef):

```
{
  "original_cid": "9f1c…e2a0",
  "original_size": 1210363,
  "shard_size": 121037,
  "shards": [
    {"cid": "…", "shard_index": 0,  "shard_size": 121037},
    …
    {"cid": "…", "shard_index": 13, "shard_size": 121037}
  ],
  "encrypted": true
}
```

Field	Meaning
`original_cid`	The chunk's CID, the same as the ChunkRef's `cid`.
`original_size`	Length of the **stored** bytes that were encoded, so on an encrypted repository it includes the 29-byte envelope. It is not the plaintext size.
`shard_size`	Shard length, the same for all 14.
`shards`	Exactly 14 entries. `shard_index` 0–9 are data shards and 10–13 are parity.
`encrypted`	Whether the encoded bytes were ciphertext. If absent, it defaults to `false`.

Reconstructing from shards: collect at least 10 of the 14 shards, checking each against its CID. Run Reed-Solomon reconstruction, join data shards 0–9 in order, and truncate to `original_size`. The result is the chunk's stored bytes. Decrypt and verify them exactly as you would a directly fetched chunk block.

Where blocks are stored

Blocks sit in an S3-compatible bucket, and the object key is built from the CID. The layout depends on the org's storage tier:

Storage tier	Object key for a repository's block
Shared bucket (Free plan and default)	`{orgId}/p/{repositoryId}/{cid}`
Dedicated bucket (org-exclusive bucket)	`p/{repositoryId}/{cid}`
Bring your own storage	`{configured prefix}/p/{repositoryId}/{cid}`, or `p/{repositoryId}/{cid}` with no prefix configured

This applies the same way to chunks, manifests, segments, roots, and erasure shards.

Bare keys. Some blocks — written by clients that didn't name a repository — sit at the bare key `{cid}` (behind the same org prefix or bucket as above). Readers try the repository-scoped key first and fall back to the bare key. Take the **scoped key first**: on encrypted tiers, a bare object with the same CID can hold another repository's ciphertext, which this repository's key can't decrypt.

Other keyspaces, under the same org prefix or bucket:

- `lfs/{repositoryId}/{oid}`: the direct-upload git-LFS shape, addressed by git-lfs's `oid` (the SHA-256 of the file, 64 hex characters). Each is **one plaintext object** holding the whole file.

  It remains the normal shape on the `plaintext` tier; on a managed repository only objects written before the agent requirement use it, and the operator migration converts each to encrypted native blocks (the manifest and chunks under `p/{repositoryId}/…`, indexed by the oid-to-content mapping) before removing it.

  A compatible older layout may also exist: the object at that key has custom metadata `enc: managed-v1` and holds a small JSON `{"enc":"managed-v1","chunks":N,"size":S}`, with 4 MiB ciphertext pieces at `lfs/{repositoryId}/{oid}/c/{n}`. That layout is already encrypted; each piece uses the block ciphertext layout above, with the ASCII string `{oid}-{n}` (`n` counting from 0) taking the place of the CID in the key derivation. New agent uploads do not write the `lfs/` keyspace at all.
- `t/{orgId}/{repositoryId}/{cid}` and `c/{orgId}/{repositoryId}/{hash}`: tree and commit objects for version history. Their format isn't covered here.
- `p/{repositoryId}/{sha256}`: encrypted preview images for a managed repository. They share the repository-scoped namespace with blocks but are named by SHA-256, not BLAKE3.
- `{sha256}` (bare), `_videoproxy/`, `_materialized/`: unencrypted derived renderings (see above).

The physical bucket name, the account behind it, and anything else about the shared bucket's arrangement are operational details. They aren't specified here and may change.

Reconstructing a file by hand

You need:

- the file's `rootCid`, from the repository's metadata;
- read access to the stored objects (your dedicated or BYOS bucket, or blocks fetched through the hub's block API);
- for a `managed` or `e2e` repository, the 32-byte repository key, or every retained generation of it if the repository's key has been rotated.

Then, in this order:

1. **Define `fetch(cid)`**: read the object at the repository-scoped key, falling back to the bare key. Then:
   - no repository key (plaintext tier): the stored bytes are the plaintext;
   - with a **managed** key: if `blake3(stored) == cid`, the block is plaintext (a tolerance for managed repositories only — an E2E or static-key reader must treat a plaintext block as an error), so use it as-is. Otherwise its first byte must be `0x01`, `0x02`, or `0x82`–`0xC0`. Pick the key generation that byte names (generation 1 for `0x01`/`0x02`, `byte − 0x80` otherwise), derive the per-block key for that version, and AES-256-GCM-decrypt `stored[13..]` with nonce `stored[1..13]` (the tag is the last 16 bytes).
   - **Verify** that `blake3(plaintext) == cid`. Stop on any mismatch or GCM authentication failure.
2. **Fetch the root**: `root = JSON.parse(fetch(rootCid))`.
3. **Classify it** using the rules in Telling the shapes apart.
   - Flat: `chunks = root.chunks` (or a `root.blocks` array).
   - Segmented (`manifest_version == 2`): for each entry in `root.segments`, in order, run `seg = JSON.parse(fetch(entry.cid))`, check that `len(seg.chunks) == entry.chunk_count`, and append `seg.chunks`.
4. **Fetch each chunk**: for each ChunkRef in order, `data = fetch(ref.cid)` and check that `len(data) == ref.size`. If a chunk block is missing and the ref has an `erasure` group, rebuild the chunk's stored bytes from any 10 shards (see above), then decrypt and verify them as in step 1. A block 404 is not proof of loss: an

in-flight save's manifest can reference blocks that have not been uploaded yet, so re-fetch after the save completes before treating a block as permanently missing.

5. **Assemble**: write each chunk's bytes at `ref.offset`. Since chunks are contiguous and in order, you can simply concatenate them.
6. **Check the total**: the output length must equal `total_size`. The manifest has no separate whole-file hash. Integrity comes from the per-block CID checks above, and the manifest itself is covered by `rootCid`.

A 1-byte range read only needs the chunks that overlap it. With a segmented root, binary-search `first_offset` to find which segments to fetch.

Versioning & stability

Stable: a stored object is never rewritten to upgrade its format, so existing data stays readable without migration. Changing anything below would change CIDs for existing content, so the code treats each of these as a breaking change:

- BLAKE3-256 hex CIDs over plaintext;
- the flat manifest's field names and serialization;
- the segmented root and segment shapes, with `manifest_version: 2`;
- the `[version][nonce][ciphertext+tag]` envelope and the `0x02` key derivation;
- the key-generation version bytes `0x82` – `0xC0`, which use the `0x02` derivation under that generation's key;
- readers accepting the older manifest shape and `0x01` ciphertext.

How format versions are signalled:

What	Signal
Flat vs segmented manifest	Which fields are present, plus `manifest_version` (absent = v1, `2` = segmented). Readers reject unknown versions.
Ciphertext envelope	The first byte (`0x01` version 1, `0x02` current, `0x80 + g` for key generation $g \geq 2$). Readers reject unknown versions.
Chunking	The manifest's `chunking.type` (`fixed` / `cdc`).
E2E key envelope	Its first byte (`0x01`).

May change without notice (none of this affects reading data that's already stored):

- the default chunking parameters and the per-extension profiles;
- when erasure coding is applied;
- the block size limit and the ¾ segment target (which change which files get segmented and what their manifest CIDs are);
- bucket and prefix arrangement beyond the key shapes documented above;
- how git-LFS objects are stored.

Not specified here: file and folder metadata, tree and commit objects, and anything about Reed-Solomon beyond "10+4 over GF(2^8) via `reed-solomon-erasure`". If you depend on one of these, contact us.

Telemetry & Data Collection

What Swarmfile measures, what leaves your machines, and what it deliberately does not collect. This page is for security reviews and procurement questionnaires; the network-level view of every connection is in Network Requirements.

Short version

- **Client usage analytics are two first-party, hosted-only paths.** The anonymous beacon (below) and, on the Music page, pseudonymous music events keyed to the HMAC of the member and repository. The Music events record only a stage and typed outcome (session start/stop, Live detection, handoff, merge, validator result), never a path, file name, Set name, track title or note. They are on by default and can be turned off with **Share pseudonymous music usage** in the Music page; `SWARMFILE_FUNNEL_BEACON=0` covers only the anonymous beacon. No third-party analytics or crash-reporting SDKs ship in the desktop app or engine. A random per-install id (generated and stored locally; no machine id, account, org, repository, path or file name) is HMAC'd server-side for one **install** row, plus one **clone_completed** row after an anonymous public `swarmfile://` clone. It goes to the hosted service only — never to a community or self-hosted hub — and `SWARMFILE_FUNNEL_BEACON=0` disables it. No third-party analytics or crash-reporting SDKs ship in the desktop app or engine. The small operational reports the client sends during ordinary work (P2P bandwidth deltas, branch switches and metadata calls) are separate disclosures, listed below; they are counters, ids and the from/to branch names in a branch-switch report, not usage analytics.
- **No third-party analytics or crash-reporting SDKs ship in the desktop app or engine.** There is no telemetry sent to an analytics service, ad network, or crash collector from the client. The **website is separate**: Google Analytics is scoped to the public marketing/docs pages, loads **only after an explicit "Allow analytics" choice**, and is **not loaded before that choice or on public repository pages, Explore, the signed-in dashboard**, or any app route — so no repository slug, file path, or dashboard URL is reported to it. The choice is stored in the browser; declining (or making no choice) leaves analytics off. After consent it records simple interaction events such as download-button clicks (which file, which page — no account id). On the `/send` drop surface it records two non-identifying funnel events (`drop_created` , `drop_upload_completed`) carrying only a file count and folder/referral booleans — no file names, paths, sizes or account id; the `/d/:id` receipt page has no analytics. (Navigating from a marketing page into any of those becomes a full page load, whose document has no analytics at all.) The Google Fonts stylesheet, by contrast, loads site-wide (the dashboard uses the same type stack); its requests necessarily carry the page URL but nothing else.
- **OpenTelemetry only exists in Swarmfile-operated diagnostic builds, and only if you point it somewhere.** Shipped desktop installs don't emit traces. In a diagnostic build, an unset endpoint still defaults to a local OTLP collector at `http://localhost:4318` ; set it explicitly if you want no export attempts at all.
- **File contents never leave unencrypted** on a managed or end-to-end repository — the storage layer and peer transfers see ciphertext, with the documented exceptions on Security Architecture.
- **Usage metering counts bytes and seats, not contents.** Billing dimensions are computed from byte counters and org membership. The optional agent-spend attribution (below) adds token counts and model names, never prompt, tool or file content.

Client-side diagnostics (opt-in)

Mechanism	Default	What it sends	Where it goes
`SWARMFILE_OTEL_ENDPOINT`	`http://localhost:4318` in Swarmfile-operated diagnostic builds; unset on shipped installs, which don't emit traces	Traces/spans for engine operations: operation names, timings, counts, error kinds, and internal identifiers (entry ids, content hashes). No file bytes. **Diagnostic builds only**: shipped desktop installs don't emit traces.	The OTLP collector **you** configure
`SWARMFILE_OTEL_SERVICE_NAME`	`swarmfile-engine`	Service name on those spans	Same collector
`SWARMFILE_UPLOAD_TRACE`	Off	Promotes upload-attempt logging (including the hub's refusal body) to `warn` in the **local log**	Local log file only
`swarmfile-doctor` / `swarmfile doctor`	Runs when you ask	Probe results, hostnames, versions, and error text, written to a local JSON report	Stays local until you attach it to a support request
`RUST_LOG`	Off (info)	More detailed local logging	Local log file only

Normally the client also sends small **operational reports** to the hub — a P2P bandwidth delta roughly every 60 seconds, a branch-switch report when a machine changes branch, and the metadata calls that make up ordinary work. These carry counters, byte totals, ids, and (for a branch switch) the from/to branch names; never file content.

What the service collects

Usage metering (billing and limits)

The hub meters usage per org:

Dimension	Measured	Billed
Storage	Bytes stored per repository (from committed manifests and object sizes)	Overage, then hard ceiling
Egress	Bytes served, counted per direction (cloud egress and P2P in/out are tracked separately)	**Not billed** — unlimited on every plan; a separate platform-wide hourly ceiling can refuse further anonymous raw reads under extreme volume (it isn't plan- or org-specific and can't be raised). Anonymous release downloads additionally pass a one-time proof-of-work check and a per-client admission budget before bytes are served (counters, not content)
Collaborators	External collaborators on private repositories above the per-seat allowance (public-repository guests are free and uncounted)	Overage, then hard ceiling
Seats	Members counted per org and reconciled to the subscription	Per seat
Headless seats	Machine seats (repository-scoped keys) minted, by org and repository	Hard cap; additional seats raise it

Metering is counters and sizes — it never inspects file contents. Details, allowances, and ceilings are in Billing & Plans.

Agent spend attribution (opt-in)

An owner/admin enables it in **Settings → Usage → Agent spend**. A member's engine then posts, at most every five minutes, per-session token counts (input/output/cached), the model name and the session, workspace and repository identifiers, plus the landed / in-review / exploration / unknown split derived from the default-branch history and merged-MR chains.

It never sends prompt text, tool arguments or results, transcript lines, file names or file contents, and a member sees only their own attribution.

`swarmfile usage agents ––local` scans a transcript entirely on the machine and uploads nothing (it reads each line only for its id/model/timestamp/token counts and then discards it); the org ledger is the default. The only email is an optional weekly owner digest and an 80% daily-budget warning.

How the split works: Agent spend attribution.

Installer downloads and installs

- **Anonymous downloads.** Installer links on public pages (the download page, the welcome page, the get-started banner, and the Security page's doctor link) go through a hub redirect that records one event per fetch: which file, which page/button, and when. The event stores no token, org, user, or device identifier; the per-IP rate limiter keeps only a short-lived counter to bound abuse. Rows are deleted after 180 days.
- **Signed-in downloads.** A download from the dashboard's Setup tab goes through a per-seat token and records which org and seat fetched which installer and when (also deleted after 180 days), so a download can be tied to the seat that made it.
- **First-launch install report.** On first launch the Desktop App reports the machine id, OS, and app version, and refreshes a last-seen time on later launches, so the Setup tab can show which of your machines are connected. The server may link that report to a recent installer download from the same account; nothing is added to the installer itself.

These records never include file contents. Aggregate download counts are available to support for capacity planning.

Internal funnel instrumentation

The service records a small, first-party **pseudonymous** set of adoption events so the team can see where developers stop: account created, repository created, public clone session started, first clone read, agent workspace created, reflected LFS push, manual CI run, music session/handoff/merge/check events, and typed plan refusals such as the Free-org limit or a private repository on Free.

Every identity is HMAC-SHA-256'd under a server-side secret before storage; the table contains **no raw org or user id, email, IP address, path, file name, branch name or free-form detail**. There is no third-party analytics or collector in this path, and it is not read by any customer-facing page.

Rows are pruned after 400 days and the only reader is an operator-only report.

The client also sends two first-party beacons to the **hosted** service:

- One **install** event the first time a device boots (one row per install id, so later boots add nothing).
- One **clone_completed** event after an anonymous public clone through the `swarmfile://` helper (one row per install id per UTC day).

Both carry only the random install id the client generates and stores locally — never a machine id, account, org, repository, file name, path or branch name — and the server HMACs it before storage; the anonymous request also carries a stateless proof of work. A community or self-hosted hub never receives either beacon, and `SWARMFILE_FUNNEL_BEACON=0` turns them off.

The client counts derived from these rows are self-asserted and inflatable (fresh cache directories mint new ids; offline or opted-out clients are absent), so they are not an audited install count.

These rows are **global operational telemetry in the system database**, the same global database that holds the org registry and billing records.

They are not part of the per-org repository data-residency pin: an EU-pinned org's blocks and repository metadata stay in its region, while its pseudonymous funnel rows remain in the global system table. The controller holds the HMAC key, so the keys are pseudonymous rather than irreversible; they are personal data and an operator can purge them by the known org/user id (see the funnel instrumentation operator runbook). An org deletion does not automatically recompute those keys; use the operator purge as part of an erasure request.

Public-page referrers

Anonymous reads of a public repository page record where the reader came from — but only as an aggregate: one counter row per (repository, UTC day, referring host). The host is stored as a name (lowercased, `www.` dropped); an IP address is never stored as itself (it counts under `ip`), the hub's own pages count under

`internal`, a missing referrer under `direct`, and anything unrecognizable under `other`. No paths, no query strings, no full URLs, no cookies, and no visitor identifier of any kind exist in the data, so no single visit can be reconstructed. Rows are deleted after 90 days. Only a repository's owners/admins can see the aggregate.

Activity and audit events

The org **Activity** feed (and its CSV export) records user-visible metadata per event: `occurredAt` (the CSV column is `occurred_at_iso`), `kind`, actor (user id, display name, email), repository (id, name), entry (id, name), a summary, and a detail field. Examples are commits, merges, permission changes, quarantine decisions, and plan/policy changes. It does **not** record file contents, and it never records a file's bytes.

Retention applies to these rows — see Audit-event retention. The personal notification inbox and its email copies contain the same class of metadata (who, what, when), not content.

Notifications by email

Email delivery uses **Postmark** as the transport provider. The message contains the same event metadata as the in-app notification (repository and entry names, actor names, a link), never file contents. Providers and hosts are listed in Network Requirements.

Third parties that can see what

Provider	What it handles	What it can see
Cloudflare (Workers, R2, Pages)	Hub, identity, storage hosts	Encrypted blocks for encrypted private repositories; plaintext for Free-plan (public or private) and public repositories (by design); request metadata
Postmark	Notification and account email transport	Recipient address and the email's metadata content
Stripe	Billing	Org/account details and payment method, never file content
iroh/N0 relay network	NAT traversal for P2P when direct connections fail	Connection metadata (endpoints, timings); relayed traffic is encrypted end-to-end between peers
Google (swarmfile.com website)	Google Analytics on the marketing/docs pages after the visitor allows it; Google Fonts site-wide	Page views, the referring URL, and download-button click events (which file and which page); nothing from the app or your repositories

What is not collected

- No browsing history in the dashboard, no cursor tracking, no session recording.
- No content indexing of your files by the service beyond what a preview/search feature needs (managed tier), and none at all on end-to-end repositories.
- No telemetry from the Desktop App UI itself (the two client beacons above come from the engine/helper and carry only the random install id).
- No automatic upload of logs or doctor reports — support only gets what you attach.

Verifying this on a machine

1. Confirm no OTLP traffic leaves the host: shipped desktop installs don't emit traces, so ports 4317/4318 stay silent even if the endpoint variable is set.
2. Run `swarmfile-doctor --json --report /tmp/report.json` and inspect the file: it contains hostnames, versions, probe verdicts, and error strings.
3. Inspect the cache directory's logs for what `RUST_LOG` / `SWARMFILE_UPLOAD_TRACE` would add.
4. For an end-to-end repository, confirm the mount's blocks are ciphertext by reconstructing one as described in Storage Format.

Where to go next

- Security Architecture — encryption tiers and threat model.
- Operations — the audit log and its CSV export.
- Trust & Compliance — DPAs, subprocessors, and incident response.

Known Limitations

Swarmfile covers a wide surface — a mounted drive, version control, git, LFS, identity, billing — and some of it is deliberately not built, or works differently from the tool it replaces. This page collects the limits that most often decide a fit, stated plainly rather than left for you to discover during a rollout. Each entry links to the page that owns the detail.

If a hard limit here is a blocker for your team, tell us — it is how we decide what to build next. For the exhaustive per-subsystem gap list, see Filesystem Compatibility → Declared gaps.

What revoking access can and can't reach

Revoking access — a removed grant, a removed member, a revoked guest, or a revoked device key — is enforced by the hub on every request immediately, and affected machines are told to delete their cached copy of the revoked files (removed from the drive's listing, cached file data purged, and a purge receipt the organization can see under **Settings → Access revocations**). Four boundaries are worth knowing before you plan a rollout around it:

- **A machine that never reaches Swarmfile can't be told.** An offline computer deletes its cached copy the next time it reaches the hub. The organization bounds that delay with the offline access window, which is **on by default (7 days)**: a machine that hasn't reached the hub within the window pauses local access (even to kept-offline files) until it reconnects. An organization that turns the window off has no upper bound on how long a disconnected machine keeps serving what it already cached.
- **A repository a machine isn't mounting is cleaned up when it's next used.** A machine deletes a revoked repository's cached content when it next contacts that repository. A repository it isn't mounting (for example one the user switched away from) keeps its cached copy on disk until then. Opening that repository again means asking the hub, which refuses and starts the purge; a machine that can't reach the hub is held to the offline access window above.
- **Deletion is not cryptographic erasure on the managed tier.** Cached files on managed and plaintext repositories are deleted; on end-to-end encrypted repositories the key is dropped so what remains is unreadable. In neither tier can Swarmfile reach bytes an application already read (they live in that app's memory), the operating system's page cache, or copies made by a third-party previewer or backup tool.
- **A removed member's own unsynced saves are kept.** Work they saved that never uploaded stays on their computer, held rather than deleted, and uploads if access is restored (on encrypted tiers it is unreadable once the repository key is dropped). Removal revokes access; it does not destroy their unuploaded work.

The self-hosted server's swarm has no hosted trust layer

A self-hosted `swarmfile-server` runs the swarm's rendezvous, but not the hosted hub's trust machinery around it: it does not sign change attestations (LAN gossip between its engines is exchanged unattested), and it has no per-node approval list — the per-organization swarm secret, handed only to authenticated keys, is the membership boundary.

Its repositories are a single trust domain (`SWARMFILE_SERVER_P2P=sealed` keeps cleartext off the swarm entirely). The posture model and what else stays on the hosted service: Self-hosting → Peer-to-peer and What the community server leaves to the hosted service.

Federation is a public-plaintext, allowlisted read path

Federated reads let an allowlisted server proxy another's **public plaintext** repository data without copying it. They are not a private or E2E sharing mechanism: managed/E2E repositories, ACLs, member data and keys are never federated, and a federated clone is anonymous and read-only.

git-LFS downloads are carried; LFS uploads and locking are not. The mesh route (A→B→C) is experimental and off by default, and cross-hub managed copy/E2E key sharing are not built.

On the hosted service the peer setup is staff-managed, and a hosted origin also needs the repository owner's consent. Exporting makes the repository's public reads answerable to anyone with the URL unless every enabled export requires the peer's signed requests (then a direct public link stops working), and an origin whose served tree carries a hiding `.swarmfile/publicignore.yml` refuses the whole public/federated read path with the uniform 404 rather than filtering it. This is not ForgeFed/ActivityPub and not a mirror.

Detail: Self-hosting → Federating public repositories, git clone → Federated public reads, Hosted federation export.

Relay-only peers don't carry file data by default

If two machines can't establish a direct QUIC path and only the public relay is reachable, Swarmfile skips that peer for block data and reads from the hub's storage instead. The relay is a transport fallback for finding/meeting a peer, not a content path. `SWARMFILE_ALLOW_RELAY_FETCH=1` on **both** the serving and fetching engines opts into relayed block transfer; those bytes are throttled. This is deliberate: moving bulk file data through the public relay is slower and costs internet egress, and the hub is already the content fallback.

Anonymous drops are public links, not a private transfer

The `/send` drop surface creates a real repository with no account, but it is a public delivery receipt, not a secure transfer:

- **Anyone with the link can read it**, and the drop repository is **plaintext** (no encryption at rest), for up to 7 days (5 GiB / 200 files). Files are not scanned for malware.
- **Keeping a copy (claiming) and deleting work only in the browser that uploaded the drop.** The claim key lives in that browser's storage; clearing site data or switching devices loses the claim (the public link still works until it expires), and an uploader cannot keep a copy from a link opened elsewhere. A Free organization's claimed repository is public and plaintext; a paid organization's is managed and private.
- **The first-drop confirmation is a self-attestation** that you have the right to share the files; it is not an ownership check.
- **Email delivery is not tracked.** The page reports how many addresses the mail service *accepted*; a bounce or spam rejection does not appear there.
- **The caps are per network address.** Up to 5 drops per network address per 24 hours and 25 per 30 days (the address is a hashed client IP); a shared office, VPN or carrier NAT can reach them, and global capacity can refuse a drop. Signing in and uploading into a repository has no 7-day drop expiry.

Detail: Anonymous drops & claiming and Sharing and collaboration → Drops.

No Ableton Live plugin or ALS merging

Swarmfile does not install a VST/AU/Max device and never merges `.als` files. Live remains single-writer: one person holds the Set's session lease, and collaborators hand off stems through per-role folders, `stem/<song>/<role>` branches and merge requests. This is file-level parallel production, not real-time co-editing. Detail: Ableton Live Collaboration.

No Unreal Engine or Unity source-control plugins

There's no Unreal Editor or Unity source-control provider for Swarmfile today, so those editors' built-in check-out and submit buttons won't talk to it. Teams work on the mounted drive directly and use the Desktop App or the `swarmfile` CLI for locks and commits.

This is separate from Revit/SolidWorks-class worksharing, which needs no plugin on Windows: there, Swarmfile honors the native OS exclusive-open lock the application already takes and turns it into a hub-enforced claim across machines. Detail: Moving from Perforce → Not supported.

No Perforce history importer

There is no tool that reads a Perforce server's revision history and replays it as Swarmfile commits. A migration brings the **head revision** across; history in Swarmfile starts from there, and the old Perforce server stays around read-only for anything older. If you need the old revisions auditable, plan to export or retain them rather than decommissioning the server on cut-over. Detail: Moving from Perforce → Not supported, Migrating Existing Data.

No git rebase or history rewriting

Swarmfile history is an append-only, server-side DAG of whole-file commits. It isn't rewritten — you can't squash, rebase, or clean up a run of autosaves into one tidy commit.

What you can do instead: `swarmfile commit --amend -m "…"` changes the most recent commit's **message only** (contents and history are untouched), and `describe` or `tag` puts a name on any past commit, including a bare autosave. `restore` and `revert` move history *forward* as new commits rather than erasing it. Detail: Coming from Git → What genuinely has no equivalent, Git and Swarmfile → Compatibility at a glance.

Moving a folder up on a branch of a very large repository

When you move a folder closer to the repository root, Swarmfile first checks that every symlink inside it still points inside the repository from its new depth. On a branch, that check reads the files the branch inherited from the point it was forked. If that inherited set holds more than 100,000 files and folders, the check doesn't run and the move is refused with `destination_too_deep`. Moves to the same depth or deeper aren't affected, and neither is the default branch. To restructure a repository that size, make the move on the default branch.

Commits on a branch that inherited a very large tree

A branch inherits its files from the point it was created instead of copying them, and the hub indexes that inherited set to record commits and pushes on the branch. On a very large repository the index is built in steps, so the first commits on a new branch can wait briefly (`fork_index_building`; the Desktop App retries on its own).

A branch that inherited more than the hub can index can't record commits or accept pushes at all (`fork_too_large`); your saves on it are kept, but not grouped into commits. Work on the default branch for a repository that size. Detail: Branches & Merging.

Workspace dependency sharing has storage and platform limits

`swarmfile workspace warm` can keep the installed dependency tree, not just the downloads. `--share-deps` builds one inside the repository (`.swarmfile/deps/<OS-arch-Node-ABI>-<libc>-<lockhash>/tree`, staged as `tree/.packages.partial` and renamed to `tree/packages` with a completion marker) and links `node_modules` to it. An interrupted copy is rebuilt on the next run rather than linked partially. It is ordinary repository content: it syncs to the hub and counts against storage, `.npmrc` is excluded, and a fork of the warming branch inherits it without reinstalling. Measured once at 126 s for the tree used in the benchmark. Limits:

- **Windows on a WinFsp mount cannot use either form.** `--local` refuses before the install; `--share-deps` can install and copy, but fails at the link step because the host cannot create the link there. The NTFS copy/link path works; a full mounted-drive run does not.
- **`--local` is Unix-only and machine-local.** It imports the engine cache as `local_layer` entries with local blocks; it never syncs and has no hub row. A different machine cannot reuse it, and a second workspace on the same machine attaches it without an install (measured 14.6 s first / 12.6 s reuse). An imported layer's blocks are pinned against cache eviction; if they are gone (for example after a manual cache delete), re-running `--local` re-imports the tree instead of reporting a stale cache hit. It preserves executable bits, so `node_modules/.bin` tools run, and a strict E2E/static mount is refused because the layer's blocks are plaintext. A failed import removes its partial subtree, and the reuse check compares the imported row count with the source tree. The timings are macOS; the repository's Linux container lane exercises the runtime.
- **`--local` is install-only and root-only.** It must run from the mount root (a subdirectory is refused) and refuses workspace/monorepo installs. A cached hit does not run the command (the CLI says so; `--json` reports `commandRan: false`), and a lockfile change needs the old `node_modules` removed once.
- **pnpm workspaces** (packages with their own nested `node_modules`) and workspace manifests are unsupported for `--local` (no recursive install layout). Use a single root install.
- **Old layers are manual cleanup.** The repository's `swarmfile workspace deps gc` lists and removes them; it keeps the current link/key and refuses when it cannot identify it. A share-deps layer whose synced tree hides `node_modules` is blocked, not partially deleted.
- **Import cap:** a local layer is bounded at 20,000 entries and 512 MiB per file.
- **`.npmrc`:** the repository file is merged over the user's into a 0600 temp in the engine cache, and relative `cafile` / `certfile` / `keyfile` values are resolved against the file that declared them. The temp is removed after the install; one left by a killed process is swept after a day.

- **Windows `--share-deps`:** the link is created only when Node can resolve it (on a WinFsp mount the check fails and the link is removed); use a plain install there. The committed Windows lab evidence covers process/relay and the platform gate, not a mounted WinFsp drive.
- **Plain clones/CI:** `workspace deps materialize` copies a `--local` layer into an unmounted checkout; for another machine, `workspace deps export` the layer, copy the archive, `import` it there (same OS/arch/Node-ABI/libc prefix), then materialize. The archive is structurally validated (20k entries/512 MiB per file/2 GiB total), a healthy existing key is kept, a corrupt one replaced, and a live/in-progress one refused; an archive inside the source is refused. The `--local` layer itself remains Unix-only, as does workspace support; a **single-package** pnpm install is supported and its relative `.pnpm` links import.

Importing an existing repository converts or skips some git shapes

`swarmfile git import` brings a GitHub, GitLab or local repository into a repository, but the git view is not a full git server, so a few things are converted rather than carried byte-for-byte. Run `swarmfile git preflight` first (or `import --dry-run`): it fetches into a temporary mirror, creates nothing, and reports every branch's verdict, the tag policy, the blockers below and the largest historical blobs before you commit to the move.

- **Annotated tags** keep their exact tag object, message and id by default (`git push` and `git import` both preserve them). `--annotated-tags convert` imports them as lightweight tags, `skip` leaves them out, and `fail` stops before creating anything.
- **Submodules** push and clone as gitlinks (mode `160000`) with their exact target id; the mounted drive hides the submodule path, and `git import` still skips a branch containing one.
- **E2E git** clone/fetch/push works through the desktop engine helper and the standalone/API-key helper (the latter with `SWARMFILE_USER_KEY_PRIVATE_B64` / `SWARMFILE_USER_KEY_ID`); the view is all-inline and LFS stays refused.
- **A branch the importer currently skips** — one containing submodules (`.gitmodules`), git-LFS pointers, or an octopus merge — is skipped and listed by `git import` (git push itself supports submodules and octopus merges); if the problem is on the source's *default* branch, the import refuses before creating the repository.
- **Upstream rewriting and deletion stay out of sync.** Re-running the import adds new commits, branches and tags; a force-pushed (rewritten) upstream branch is refused rather than mirrored, and branches or tags deleted upstream are not deleted in the repository — remove those there with the normal branch and tag commands.
- **Git-LFS objects aren't fetched by the importer.** An LFS-using default branch is refused and an LFS-using side branch is skipped, rather than importing pointer files with no objects behind them.

What you can do instead: keep the source as history-of-record until the import is verified, and prefer `git push` / `git-LFS` as the ongoing bridge when the repository keeps moving. Detail: `swarmfile git import`, Git and Swarmfile → Compatibility at a glance.

git-LFS uploads on managed repositories need the transfer agent, and LFS is unavailable on E2E

A stock `git-lfs` client can't hold the per-user key, so git-LFS is refused on end-to-end encrypted repositories. On a managed-key repository, uploads must go through the Swarmfile transfer agent (`swarmfile-lfs install`; the Desktop App bundles it) so the object is stored as encrypted blocks — the stock direct paths are refused with the typed `lfs_agent_required` error. On an unencrypted repository, stock direct uploads are still allowed and are plaintext by definition.

Objects a managed repository accepted through the old direct paths before this rule are converted by an operator-run migration; until it reports clean for that repository (no remaining plaintext candidates), don't assume every older LFS object is encrypted. Detail: Git-LFS → Which repositories it works on, Security → Encryption.

Scoped keys are mint-time, and path-scoped keys are engine/API-only

A repository API key can be **branch-scoped** or **path-scoped**:

- **Branch-scoped** — one non-default branch; git, history and merge requests stay on that branch.
- **Path-scoped** — the same branch plus allow/deny globs; engine/API-only, with no git, history, commit walk, merge requests or LFS.

The scope is set at mint time: there is no scope update, so revoke and mint a new key to change it. Minting does not validate that the branch exists; the hub refuses a dead or default branch on first use.

Each allow/deny list holds at most 64 globs of at most 1,024 characters. A path-scoped key's batch provenance refuses a manifest with more than 8,192 refs outside the entry's previous root (`path_scope_provenance_too_large`); the cap is checked before the per-ref upload lookup, so refs the key uploaded still count. A flat manifest with enough chunks can hit the same cap, not only a segmented file, and a key **without a path scope** (branch-scoped) is not subject to that provenance check — a repository-wide key is not required. On an E2E repository a path-scoped key cannot introduce new content, and E2E git with a scoped key is not supported (the wrapped-key route is outside the scoped allowlist) — use an owner credential. Community servers reject scoped-key fields; scoped keys are hosted-only until P1.

Detail: Branch- and path-scoped keys.

Branch archive and delegated agent tokens

- The **branch archive** is one tar of the caller's readable files, emitting at most 5,000 files / 480 directories and 1 GiB per request on hosted (256 MiB on a self-hosted community server). The walk and the two route-level reads scan at most 20,000 rows (and 500 tree pages) — that scan bound is not an emitted-file count and applies to hosted and self-hosted alike. On the community server archive starts are limited to 30 per minute **per repository** (shared by all callers; a 429 carries `Retry-After`). A path-scoped key is refused; the folder form is member-token only. v2 adds repeatable `paths` prefixes (at most 64), a content-based `ETag` / `If-None-Match` 304, a JSON manifest for client-side deltas, and opt-in gzip (`format=tar.gz`). There is no server-stored delta, resume or checksum, and no zstd. The folder form supports the same v2 filters/format/ETag and a folder manifest. Empty directories, zero-byte files and safe relative symlinks are represented; a submodule/gitlink is an empty directory placeholder, not hydrated. An LFS pointer is archived as the pointer by default; `lfs=1` hydrates the content-map/native object or a standalone plaintext LFS object (a managed-encrypted standalone object is `503 archive_content_unavailable`). An `lfs=1` manifest is never answered `304` (its ETag does not cover the LFS map); its `200` body is authoritative. LFS object upload is not part of the archive. The sandbox sync client removes a changed **plaintext** non-LFS file above `directReadThresholdBytes` (default 256 MiB) from the archive and reconstructs it from its root/segment/chunk blocks (every BLAKE3 CID verified), so plaintext files are not limited by the archive's per-request cap. A **managed** file can be read through the hub `/content` range read (server-side decrypt), so existing managed files are not archive-capped; sandbox writes to managed are refused. **E2E** has no

server-side read path: sandbox sync is unsupported at every size (archive `409`), not above a 1 GiB cap. Managed/E2E content writes are refused with `content_requires_plaintext`; a path-scoped key cannot direct-hydrate a large file (use a branch-scoped key). Windows-hostile names may fail a Windows extractor. Hosted and self-hosted; both scan at most 20,000 rows, and a self-hosted community server cannot serve managed/E2E tiers. On the community server `X-Swarmfile-Archive-Head` may be a provisional member claim, not a confirmed commit hash: read `X-Swarmfile-Archive-Claim-Status` (`confirmed` / `provisional` / `unknown`) before treating it as repository history. See Hydrating without a mount.

- **The stateless batch checkpoint** (`POST /changesets/batch`) is served by hosted and self-hosted (the self-hosted server is flat-key only and a member key's batch is a provisional claim, not a confirmed-head advance); the details below are the hosted contract: at most 500 path changes per request; content-addressed, or inline `contentBase64` on plaintext `agent/*` branches (no unified patch); file creates/deletes, no directory deletes or renames. A path-scoped key may use the content-addressed form only: every path is glob-checked and every root must be this key's upload, the entry's previous root, or an in-scope branch entry's root; inline content is refused. Symlink creates/retargets/deletes use `{path, linkTarget}` / `{path, delete:true}` (targets are validated, no content). Directory delete/rename is not part of the batch; use the entry metadata API. Create/update use the commit time, while delete preserves the existing mtime. A larger batch uses the async chunk routes (<=500 per request, 4,000 total, content-addressed only); the repository job pins the base and re-checks ACL and path/provenance at execution. Inline content is not accepted there. The community server rebuilds the whole live branch tree per commit, so it caps the base at 100,000 entries by default (`SWARMFILE_SERVER_MAX_BATCH_TREE_ENTRIES`, up to 10,000,000; an over-cap branch is `413 batch_tree_too_large`) and limits batch starts to 60/minute per repository. See Committing a sandbox's changes.
- **Service-minted agent tokens** are Starter+ and hosted-only; a token lasts at most one hour. The community server's short-lived TTL member keys are a separate, narrower capability, not a delegation or an agent token. In an SSO-enforced org the mint needs an owner or interactive session (a non-owner service user's PAT is refused). A path-scoped token cannot use the branch archive. See Service-minted agent tokens.

Git materialization follows your access today

A clone or fetch reads the whole of every commit it walks — each commit's tree exactly as it was archived — and access is evaluated against the access you hold **today**:

- A folder restricted for you **now** means any commit whose tree contains it can't be handed over whole; a clone or fetch that has to walk one is refused with a message saying so, rather than returning a repository with silent holes. Grant read and the same clone succeeds — including commits made while you were restricted; materialization is not frozen to the access of the commit's date.
- Restricting a folder retracts it from clones at once, for new commits and old ones alike.

The mounted drive, `swarmfile materialize` and git all follow the access you hold today. The access that existed when a commit was made is still recorded, and an explicit point-in-time history view can ask for that replay — it is not what git reads. Detail: Clone a repository → What a clone contains.

Anonymous clone stops while a policy hides paths

The anonymous git front door is refused — not filtered — while any active branch's committed tree carries a `.swarmfile/publicignore.yml` that hides a path in it, any tag's committed tree carries one that hides a path, or any ready (published) release hid paths, because git history is content-addressed and a filtered clone at the same commit ids cannot exist.

The check evaluates both committed trees a branch can advertise (its confirmed head changeset's tree and its live head tree — they differ for a fork taken from a dirty source) and every tag's changeset tree.

A release already published keeps the contents it was built with. A policy that hides nothing does not block, and a policy on an archived branch does not either.

An invalid or unreadable policy, or a boundary read/walk error, refuses anonymous clone (fail closed). The boundary is anonymous-only; the policy hides nothing from members, and their reads remain ACL-scoped, so a restricted folder can still refuse a member's clone.

What the policy does not reach

Remaining gaps: a path that exists only in an older commit is not retroactively hidden by the current policy.

The publish wizard's preflight warns when a bounded history audit (up to 100 commits, within the audit's own 800-tree-object and 200,000-file bounds) finds hidden paths that exist only in older commits, with an owner/admin-only path sample; that audit is separate from the boundary's 100-tree/500-object/200,000-file scan caps. The warning never blocks publishing, and it reports unavailable rather than failing the preflight when the audit cannot run.

Rotation or a history rewrite is the remedy — the policy is not a retroactive scrub.

Commit messages are served verbatim

An anonymous clone also serves commit and annotated-tag messages verbatim (filtering a message would change every commit id); the web commit list and the federation activity/commit tails redact names the policy hides, but a commit message in a clone does not, and the committed `.swarmfile/publicignore.yml` itself remains part of the clone history. Do not put secrets in public commit messages.

Caps and caching

The boundary evaluates up to 5,000 active branches; its expensive-work caps are 100 unique policy-bearing trees, 500 tree objects under policy work, and 200,000 files (policy-free root probes are bounded separately), and exceeding one refuses anonymous clone (fail closed). A change takes effect within about a minute (the boundary verdict is cached for up to 60 seconds); the git refs themselves are answered `Cache-Control: no-store` and served live, not from an edge or browser cache.

Detail: Publishing Releases → Keeping paths out of a release, git clone → Limits and caveats.

A release copy is a snapshot of files and directories

Copying a release — the web **Copy** action or `swarmfile clone … --into` — lands the release's files and directories as a new snapshot on the destination's **default branch**. It doesn't track the original: no history, no tags, no later sync. It is recorded as **forked from** that release, though — a public copy's page links back, and the original's page lists its public forks (labeled (claimed) when the copy came from the CLI rather than the web action).

- The web action always creates a **public** repository (it copies a public release). For a **private** destination, copy through the Desktop App's CLI instead — it seals the blocks on your machine before they upload (see Publishing releases → Getting a copy).
- Symlinks are not part of a public release, so a copy contains none; directories are recreated from the manifest's paths.
- A copy adds files; it never overwrites. A same-name file with different content refuses the copy, and a destination whose default branch is protected refuses it (`409`) until the protection is lifted.
- A copy that is canceled or stops at the destination's storage limit cleans up what it stored, except data the destination repository has since come to share or reference. Data from an earlier, retried attempt can also stay, and an organization on dedicated or bring-your-own storage keeps everything the copy stored. What stays is in the partly copied repository, which counts toward your organization's storage. The storage-limit error names that repository (`leftoverRepositoryId`), and deleting the repository frees the space.

Folder-profile columns: shown inline in the file list, authored in the Desktop App

Folder profiles (the typed columns like "Net", "Payment Ref", "Paid") surface values in the web file grid, the Desktop App's file list — the active columns render **inline** in the ordinary folder view, where a **Columns** toolbar menu manages the shared profile (add / edit / remove / reorder columns) and imports/exports values as CSV — and `swarmfile field`, but three parts of the full picture are not built yet:

- **No create/edit UI for the profile on the WEB or in the CLI.** The Desktop App authors the schema (Columns → **Manage columns...**) and applies one profile to many folders (Columns → **Apply an existing profile...**; a folder whose columns are inherited is flagged), but the web grid and `swarmfile field` only read/edit **values** under whatever profile applies; a profile created outside the Desktop App is defined through the API (`POST /folder-profiles` on the repository). Treat profile authoring there as a scripted/admin task for now.
- **Native Explorer columns are OS-gated, and absent on Windows 11 25H2+.** The Windows property handler exists, but Explorer's column chooser does not on build 26200 and later, so the installer deliberately skips the handler registration there — there is nothing native to surface. The web / Desktop App list is the intended column surface on every platform.
- **No Finder, Nautilus (GNOME) or Dolphin (KDE) column provider.** On Linux and macOS the web grid, the Desktop App's inline columns and the CLI are the column surfaces.
- **No org-catalog profiles in the Desktop App.** The org catalog exists server-side, but a profile is bound from the folder it applies to (Columns → **Apply an existing profile...**, or **Use as the repository default** for the whole repository). Deriving a repository profile from the org catalog, or applying one to a folder you are not in, is not built.

Encryption tier is fixed at repository creation

Managed (the default for paid private repositories), end-to-end, and plaintext (public repositories and every Free-plan repository) are chosen when a repository is created and never change afterward.

Upgrading a plan, or deciding later that you want E2E, does not retroactively encrypt an existing repository — you create a new repository at the tier you need and bring the content across. Choose deliberately at creation if encryption posture matters. Detail: Getting Started → Creating a repository, Security Architecture → Unencrypted tier.

SSO sign-in needs your org's slug

There's no email-domain auto-detection. A member signing in with SSO enters their **organization's slug** (the short name in the org's Swarmfile URL), or follows a direct per-org sign-in link an admin can share — rather than typing an email and being routed to the right identity provider automatically. It is a one-step onboarding cost for non-technical staff; the per-org link removes it on a machine an admin provisions. Detail: Getting Started → Signing in, Identity.

Switching accounts requires an app restart

Switching to a different account while the app is running is refused (`account_switch_requires_restart`). Any unsent work from the previous account stays on your machine, quarantined rather than lost. The requested sign-in is kept: restart when the Desktop App offers it and the switch completes. This is most visible to freelancers and contractors working across several client orgs in a day. Detail: Getting Started → Signing in.

Concurrent edits

- **A save over a teammate's edit is kept for recovery, not silently merged.** When two people (or two drives) change one file, the save that lost the race is refused: the file reads read-only until it is resolved, the conflict appears in the app and the drive's status, and history always retains the version that was overwritten.
- **You can make that refusal fail the save itself.** A repository setting, or the per-drive **Conflict handling → Hard: fail the save** option, makes the saving call wait for the server and fail with a stale-version error instead of parking the conflict — useful for scripts and coding agents, whose loops otherwise never learn a save did not become truth. On Linux the failing call is `fsync` (or the close-time flush), and on Windows `FlushFileBuffers` ; on macOS a `close()` cannot carry the error and `fsync` is not reliable for it either — the drive's filesystem may treat it as a local no-op — so check the drive's sync state and refusal surfaces, not just the fsync result, before assuming a save is committed.
- **On macOS drives, a whole-file overwrite is caught only while no newer open has happened since.** A drive on macOS cannot attribute reads to a particular process; it records the last ordinary open of the file instead. An app that opened the file before a teammate's change and later overwrites it receives a conflict — but if anything re-opened the file after that change (an indexer, a preview, the app's own check), the later save is accepted as usual (the previous version stays recoverable in history). On Windows and Linux drives each process is tracked individually, so the same situation is refused. If you script writes on macOS, re-read the file immediately before overwriting it.

swarmfile

`swarmfile` drives Swarmfile version-control operations against a running engine: commit, switch/restore, branches, merge, and the rest of the day-to-day repository commands.

It talks to the engine over its control socket — most subcommands mirror an operation the Desktop App can do, and every subcommand requires a running engine except:

- `completions`
- `demo offline` (it changes the host firewall, not the engine, and must work exactly when the engine is what is being cut off)
- `usage agents --local` (a local transcript scan, uploaded nowhere; the default org-ledger mode needs the engine)
- `processes` (local record/log management, including while the engine is stopped)
- the `git` helper verbs (`git url`, `git clone`, `git preflight` and `git import`, which answer before the engine connection is needed)
- the pure Git-habit shortcuts (`push` / `pull` / `clone` / `rebase` / `cherry-pick` / `rm` / `mv` / `cp` / `mkdir` / `touch` / `worktree` / `grep` / `init` / `remote` / `clean` / `reflog` / `mergetool` / `difftool` / `shortlog` / `submodule` / `bisect` / `gc` / `prune` / `fsck` / `archive`, which work standalone

`add`, `stash`, `reset`, and the executable `clone <id> <path>` / `clone … --checkout` / `clone … --into` forms are the exception among the shortcuts — they map onto real commands, so they need a running engine too (see the Git-habit shortcuts note below).

If the engine isn't running, start it (or the Desktop App) first, or use the standalone `swarmfile-doctor` binary to diagnose connectivity without one.

`swarmfile` comes with the desktop app on every platform: `/usr/bin/swarmfile` from the Linux `.deb`, `/usr/local/bin/swarmfile` on macOS (a symlink into `Swarmfile.app`, so desktop updates carry it too), and `swarmfile.exe` in the Windows install folder (`C:\Program Files\Swarmfile`), which the installer adds to the machine `PATH`.

There is no separate CLI download — install the desktop app and open a new terminal.

A macOS install that came from the `.dmg` rather than the `.pkg` needs one full installer run to get the command-line tools and the `PATH` entry — **Diagnostics → Reinstall** does that from the tray; on macOS the tray also repairs the `/usr/local/bin` symlinks at launch where the directory is writable.

`swarmfile` doesn't include a `search` subcommand — search needs its own hub-authenticated round trip independent of the control socket, so it lives in the standalone `swarmfile-search` binary instead.

Global flags

These apply to every subcommand.

Flag	Description
`--cache-dir <path>`	Engine cache dir (where `control.json` lives). Respects `$SWARMFILE_CACHE_DIR`. Default `~/.cache/swarmfile` (macOS/Linux), `%LOCALAPPDATA%\swarmfile` (Windows).
`--json`	Print raw JSON instead of a human-readable summary.
`--mount <id>`	Target this mount instead of whichever one is `current` — see `mounts` below. Has no effect on `mounts` subcommands themselves, which address by their own `id` argument instead, nor on `workspace switch`, which changes the whole engine's active workspace rather than one mount.
`--no-wait`	Don't wait for a long operation: print its id and exit `3` at once.
`--timings`	Print elapsed time (and phase timings where the command reports them) to **stderr**; commands that print their own JSON body also gain a `timings` object (`elapsedMs` plus named phases) under `--json`. Never changes stdout. Supported on commit, checkpoint, changelist submit, mounts new/open with `--wait`, materialize, hydrate start, `workspace warm`, and the agent hooks; hydrate start and the hook internals print to stderr only.

> These are global flags, so they work on either side of the subcommand: `swarmfile --json status` and `swarmfile status --json` are equivalent, as are `swarmfile --mount m2 status` and `swarmfile status --mount m2`. That holds for the git-habit shims too — `swarmfile rm foo --json` is read as the flag, not a path.

Exit codes

Scriptable commands separate "you need to do something" from "something failed", so a CI or pre-flight script can branch on it:

Code	Meaning
`0`	Success — including the Git-habit shortcuts whose intent the product already satisfies (`push`, `pull`): nothing to do, so nothing failed.
`1`	Any other failure: an unreachable engine, a hub transport error, and the shortcuts whose operation has no equivalent here.
`3`	A long operation is still running — after `--no-wait`, or when you pressed Ctrl-C to detach. It carries on in the engine; follow it with `op status` / `op wait`.
`2`	A refusal a script can act on, **or a CLI usage error** (an unknown command, flag, or argument). With `--json`, refusals print their response body as before, and usage errors print `{"code":"usage_error","kind":"…","error":"…"}` instead of clap's prose — so a script can tell a typo apart from an actionable refusal.
`75`	Paused, not done: the hub accepted the operation (a large `revert`, `checkout` / `restore --to`, `trash restore` / `purge`, folder delete, `repository delete` or field write) but paused its background job because the repository used up today's write budget. It resumes on its own at the time the message gives; nothing failed, but it has not landed yet, so a chained `revert && mr create` stops here. `--json` prints `{"code":"job_paused","jobId":…,"retryAtSecs":…,"message":…}`, and `op status` exits `75` for such an operation. `undo` also exits `75` (`operation_paused`) when the operation it would undo is still finishing behind a paused job; undo it once it has finished, or use `undo --skip` to undo the one before it. If the job later fails, you get a notification.

A command page can assign `3` its own meaning where there is no long operation to follow — `git index` exits `3` when the hub couldn't be reached, for example.

The translated shortcuts (`add`, `stash`, `reset --hard <rev>`, `clone <id> <path>`) return the exit code of the command they map to, not a shortcut code.

Git-habit shortcuts honor `--json`: instead of the sentence they print `{"code":"already_satisfied"|"not_applicable","exitCode":N,"hint":"…"}`.

The confirm-gated verbs all take `-y` / `--yes`, and most also accept `-f` / `--force` as a synonym — except `git enable`, and the verbs with a separate safety-gate `--force` below — `stash drop` / `clear`, `changelist discard`, `branch prune`, `trash purge`, `repository delete`, `sync discard`, `git enable` (one-way, so it never proceeds without it), `ignore-clean`, and (unless `--dry-run`) `cache reclaim`, `cache public --clear` / `scratch gc`.

Without it, a non-interactive stdin or `--json` refuses (`{"code":"confirmation_required"}`, exit 2) rather than destroying work; the estimate-only verbs (`ignore-clean`, `cache reclaim`, `scratch gc`) take `--dry-run` to preview without confirming or deleting.

Two prompt-gated verbs are the exception and proceed without `--yes` in the right context: `hydrate start` and `workspace switch` both proceed under `--json` or a non-interactive stdin (an interactive terminal still prompts).

Both are resumable/reversible operations rather than destructive ones, and `--yes` only skips the interactive prompt.

A few verbs have a separate `-f` / `--force` meaning "proceed despite a safety gate" — `mounts close` (open changelists), `scratch clear` (repository still open), and `ignore-clean` (`-y` skips its prompt, `-f` also overrides its recent-foreign-edit gate).

Commands that currently use exit **2**:

- `mr status` (open, not yet approved — the CI review gate)
- `mr approve` / `mr request-changes` with a source stamp the merge request has moved past (`source_changed`)
- `conflicts resolve --auto` / `--all --auto` (a human must choose — a request that *fails* mid-sweep is exit 1 and takes precedence)
- `lock` (already held)
- `mounts close` (open changelists, or the boot mount)
- `mounts retry` (`mount_retry_unavailable` for a non-boot mount or one that attached earlier)
- `acl mode set` / `acl entry …` (`forbidden` / `not_found` / `plan_restricted`)
- `unlock-request respond` / `cancel` (already resolved, or not yours)
- `workspace switch` (another switch running, a changelist open, or a drive open on another repository — `cross_repository_mount_open`)
- `checkout` with no target
- `blame --lines` (binary/non-UTF-8/oversized, revoked or lease/key refusal, or an engine that predates line attribution)
- `rollback --at` with a synthetic (`d1:`) version (no local restore key)
- `ignore-clean --yes` when a candidate was recently edited by someone else (unless `--force`) or when any delete failed
- `hydrate start` / `fetch` when a job is already running
- any confirm-gated verb without `--yes` (the `confirmation_required` refusal above; `ignore-clean --yes` keeps its own cross-machine gate)

Version control

See Version Control and Branches & Merging for the conceptual walkthrough — this section is the flag-by-flag reference. If you're arriving from Git, Coming from Git maps your habits onto these commands.

> **Git-habit shortcuts.** Typing a Git command that has no Swarmfile equivalent — `push`, `pull`, `clone`, `rebase`, `cherry-pick`, `rm`, `mv`, `cp`, `mkdir`, `touch`, `worktree`, `grep`, `init`, `remote`, `clean`, `reflog`, `mergetool`, `difftool`, `shortlog`, `submodule`, `bisect`, `gc`, `prune`, `fsck`, `archive`, `notes` — prints a one-line pointer to the real workflow instead of an error (e.g. `push` explains that saves sync automatically, `rm` / `mv` point at doing it on the mounted drive, `worktree` points at `swarmfile mounts open`, `grep` points at `swarmfile-search`, `mergetool` points at `conflicts resolve --tool`, and `clone` is argument-aware — a public-repository URL, a `/s/<token>` share link, a `swarmfile://mount?repository=…` deep link, or a lone repository id each get the right next step; `clone <repository-id> <path>` actually opens a mount — add `-b <branch>` to pin it to that branch, exactly like `mounts open --branch`, or add `--checkout` to write a plain directory instead (see `materialize`); `clone <public-repository> --into <org>/<repository>` copies a public release into a repository, encrypting blocks on this machine — the way to get a public release into a **private** repository, since the web Copy action only produces public ones. Add `--tag <tag>` to pick a release, `--path <p>` (repeatable) to copy a subtree, and `--new` (with optional `--public`) to create the destination repository first — a `--new --public` create records the copy as a claimed fork of the source (the destination links "Forked from"; a private `--new` records nothing). It runs as a long operation (`--no-wait`, `swarmfile op status|wait|cancel`; per-file progress; a cancel resumes from what's done), and its refusals (unknown slug or tag, no write access, a protected branch, a name collision) exit `2` while hub or transport failures exit `1`. These shortcuts work even when the engine isn't running (the executable `clone` forms need it).
>
> Three shortcuts have a real destination and **run it** instead of just advising: `stash` (and `push` / `save`) → `changelist park`, `stash list` / `show` → `changelist parked`, `stash pop` / `apply` → `changelist resume`, `stash drop` / `clear` → `changelist discard` (`stash list` numbers entries `stash@{n}`; `stash drop stash@{n}` or `drop n` discards just that one); `add` → opens a changelist if none is open (idempotent — saves then stage into it automatically, and this mount goes into staged mode until you `commit` or `changelist cancel`); and `reset --hard <rev>` → `restore <rev>` (one undoable commit). `add <path>` exits 1 if the path doesn't resolve, so a typo doesn't silently pass. Parked work is a **set, not a stack**: a bare `pop` / `apply` resumes all parked work for the branch, and `pop <n>` (or `pop stash@{n}`) resumes just that entry and leaves the rest parked — an entry parked on another branch is refused, since parked work resumes on its own branch (switch to it instead). If an entry's changeset was closed on the hub while it sat parked, resuming re-creates it and its staged files under a new id, and the result names the new id. A bare `drop` or `clear` abandons all of it — so discards ask first (or take `--yes`). On success these print a sentence, not raw JSON. Bare `reset --hard`, `--soft` / `--mixed`, and bare `reset` still print guidance, since there is no unstaged state to discard. These three need a running engine.

Referring to a commit. Every command that takes a commit (`show`, `restore`, `revert`, `describe`, `diff`, and the commit form of `checkout`) accepts any of: a commit number (`482`), the `#N` form the log prints (`#482`), `HEAD`, `HEAD~N` / `HEAD^` (N commits back in the current branch's log), a **tag name**, a **branch name** (its head), a changeset id, or a 64-hex **commit hash**. A `seq:` / `head:` / `branch:` / `tag:` / `changeset:` / `hash:` prefix forces which kind is meant. Resolved client-side to the commit number before the request.

`commit`

Open a changelist (if none is open) and submit it in one step.

Flag	Description
`-m, --message <String>`	Commit message
`--amend`	Rename the **most recent** commit instead of creating a new one (like `git commit --amend`). Message-only and non-destructive — contents and history are untouched. Requires `-m`. Equivalent to `describe` on the latest commit.

```
swarmfile commit -m "Fix layout regression in the deck"
swarmfile commit --amend -m "Fix layout regression in the title deck"
```

`checkpoint`

Close this mount's open auto-commit changeset **now**, creating a rewindable commit — the same kind of commit `swarmfile log` shows, with no second history. Use it before a risky change, or per agent turn. An explicit staged changelist is left untouched (submit it with `commit`); an `agent/<label>-<6 hex>` workspace also closes after 30 seconds of quiet (`SWARMFILE_AGENT_CHECKPOINT_IDLE_SECS`, `0` disables the shorter window).

A checkpoint of more than about 5,000 files is committed by the hub in the background. The command waits up to about three minutes for it to land.

If it is still running after that, the command says so and exits 0, and the commit appears in `swarmfile log` once it lands; a `-m` message is not applied in that case, so name the commit with `swarmfile describe` afterwards.

If the hub cannot finish the commit (for example, write access was removed mid-way, or the change is too large to record as one commit), the command exits non-zero and says why; nothing is committed, but your saves are kept.

A commit that failed for a temporary reason is retried automatically. Saves made while a large commit is being recorded go into the next commit, which is recorded after it.

Flag	Description
`-m, --message <String>`	Name the commit this checkpoint creates.
`--if-changed`	Exit 0 with no output when nothing changed (for tool hooks and scripts).
`--wait <seconds>`	Wait for in-flight saves to land before checkpointing (default 5; `0` posts immediately).

```
swarmfile checkpoint
swarmfile checkpoint -m "green build"
swarmfile restore --to 482
```

`undo`

Undo this device's newest local operation, or inspect the journal. **It is local and control-route only:** it cancels a pending create or inverts a confirmed delete, rename, move or restore.

A confirmed create and a restore are undone as a guarded delete: the engine locks the entry, re-reads the hub's current version (a teammate's newer version is refused `superseded`), refuses while the entry has unsaved or queued local changes (`blocked`, try again once they sync) and deletes with the hub's version check, so nothing newer is ever deleted.

A directory is refused. A content edit or teammate's change is refused `not_undoable`; `--skip` explicitly targets the next invertible row. VFS/mount namespace ops are journaled read-only; undoing a queued VFS op is not offered (let it sync, then undo the rename or restore it from Trash). It is not a history rewrite.

A multi-entry gesture undoes as one. A multi-select delete or move from the app is recorded as one gesture, and `undo` (or `undo --op` on any of its rows) reverses all of it, reporting each entry. If some entries are refused, the answer says which and why and exits `2`; the refused entries keep their state, so running `undo` again retries only them.

An entry that can never be reversed — a new file deleted before it finished uploading is gone for good — is listed under `skipped` with the reason, without failing the rest; if every entry in the gesture is skipped, the undo is refused `not_undoable` (exit `2`), never reported as done. `undo --list` marks such rows `(one gesture of N)`. A single-file delete or move is not a gesture and undoes on its own.

A large gesture can take a while: `undo` and `redo` run as long operations (`--no-wait`, `op status` / `op wait`), cannot be canceled part-way, and finish in the engine even if the command is interrupted.

Pressing Ctrl-C during `undo` or `redo` detaches rather than stopping it: the command exits `3` and prints the operation id, and `op wait <id>` then exits the way `undo` / `redo` would have (`2` when something was refused, `75` when the operation is still finishing behind a paused job). `--dry-run` answers straight away, without an operation id; if another undo or redo in the repository is still running, it says so (`in_progress`) instead of waiting.

Operations recorded before gestures existed, and VFS/mount operations, are still undone one entry at a time.

An operation whose hub job is still paused behind the daily write budget is refused `operation_paused` (exit `75`). So is any undo or redo while a paused revert, checkout or field commit in the repository is still finishing, and an undo of something inside a folder whose paused restore, folder delete or purge is still running.

`--skip` passes over a paused row, but it never undoes an older operation inside a folder whose paused delete is still running. When every candidate is paused it exits `75` instead of reporting nothing to undo.

A folder operation that only partly finished (its paused job later failed) stays `partial`: a plain `undo` stops on it and tries again (a folder still in Trash comes back), and if the folder is already back it answers `partial` (exit `2`) and keeps the row for a later retry.

A refused undo (for example a confirmed-create delete the hub refused, `delete_failed`) exits non-zero, never `0`.

Flag	Description
`--list`	List this device's recent journal rows (newest first).
`--dry-run`	Show the inverse plan and precondition verdict without applying it.
`--op <OP_ID>`	Undo this journal row instead of the newest.
`--skip`	Explicitly skip a newest non-invertible user action (or one still finishing behind a paused hub job) and target the next invertible row.
`--before <SEQ>`	Page `--list` rows before this journal sequence (a newest-first cursor).

```
swarmfile undo --list
swarmfile undo --dry-run
swarmfile undo
```

`redo`

Re-apply the newest undone delete, rename, move or restore, or a row named with `--op`. A cancelled pending create is terminal and cannot be redone. Redoing a delete is guarded like `undo`'s deletes: a file that changed since the undo is refused `superseded`, and so is one the hub has already deleted (the redo has nothing left to do); a local cache miss that the hub does not confirm is retryable `blocked`. A gesture redoes as a whole. `--dry-run` shows the forward plan.

```
swarmfile redo --dry-run
swarmfile redo
```

`changelist`

Explicit changelist control (Mode B: staged atomic submit) — for when you want to stage multiple saves before committing them atomically, rather than using `commit`'s one-step flow. A mount can hold several open changelists at once (Perforce's `-c` model) — see Multiple changelists at once for the conceptual walkthrough.

Subcommand	Flags	Description
`open`	`-m, --message <String>`	Open a new changelist and make it current; subsequent saves stage into it until submit/cancel/switch. Never fails if one is already open — opening a second doesn't close the first.
`submit`	`--id <changeset>`, `-m, --message <String>`	Atomically apply a changelist. Defaults to the current one; `--id` targets a specific one without switching to it first. `-m` sets the commit message at submit time (git-style) instead of naming the changelist at `open`; omitted, the changelist keeps its open-time label.
`cancel`	`--id <changeset>`	Abandon a changelist. Defaults to the current one; `--id` targets a specific one without switching to it first.
`status`	—	Show whether the current changelist is open and what's staged, plus any files blocked by an unresolved conflict — the same changelist + conflicts pair `status --changes` renders.
`list`	—	List every open changelist, `*` marking which one is current.
`current`	`<id>`	Switch which open changelist is current.
`reopen`	`[path]`, `--to <changeset>`, `--from <changeset>`	Move staged file(s) to a different open changelist without unstaging/restaging. With `path`, moves that one file. Without it, moves every staged file — from the current changelist, or from `--from <changeset>` if given — to `--to`. `--from` applies only without a path (a path with `--from` is a usage error).
`park`	—	Set aside every open changelist on this mount (kept, and kept alive on the hub) so the branch/repository can be left without abandoning the work.
`parked`	—	List parked changelists with age + files.
`resume`	—	Restore parked work for the mount's current branch.
`discard`	`--changeset <id>`, `-y, --yes`	Abandon parked work (one with `--changeset`, else all). Asks first (it cannot be undone); `--yes` skips the prompt.
`staged-base`	`--changeset <id>`, `--entry <id>`	Return the exact hub version the staged row was based on. Hunk selection must diff against this, not the live head.
`commit-plan`	`--changeset <id>`, `--entry <id>`, `--path <path>`, `--partial <FILE>`, `--remainder <FILE>`, `-m, --message <String>`	Commit exactly one staged file's selected hunks (the exact content in `--partial`), leaving the remaining hunks staged in a fresh changelist against the new head. `--remainder` is the complete content to stage in the follow-up changelist (normally the full working file); it may be empty when every hunk was selected. The source must be the changelist's only staged file; it must be a regular non-LFS file, and the selected result must be non-empty. No working-tree write is made by the engine; the extension saves the open buffer first. `--path` names the staged file; `--changeset` is its source changelist (from `changelist list`).

```
swarmfile changelist open -m "Reorganize asset folders"
# ...edit and save files...
swarmfile changelist status
swarmfile changelist submit
```

```
swarmfile changelist open -m "Swap all Act 2 plates to final grade"
swarmfile changelist open -m "Quick fix: wrong LUT on shot 12"
swarmfile changelist list
swarmfile changelist current a1b2c3d4
swarmfile changelist reopen "Renders/Act2/shot12.mov" --to e5f6a7b8
swarmfile changelist submit --id e5f6a7b8
```

`agent`

A coding-agent session is a per-mount changelist labeled `Agent: <name>` (a header, not a trailer). `agent start` reuses an open same-name session or opens one and makes it current; saves stage there. `agent done` submits that session atomically and prints its changelist id, giving the agent a stable baseline without waiting for the other agents. It refuses to submit a current/`--id` changelist that is not an agent session. `agent wait` blocks until matching sessions are submitted. This uses the existing per-mount changelist and adds no server contract.

One mount per agent is the supported topology; a shared mount is one writer identity. For exact line-level attribution run one engine per agent with a distinct `SWARMFILE_MACHINE_ID` (a per-agent API key still needs its own engine). The label is claimed; the key/machine is the authoritative identity. Only open changelists are listed — parked and in-flight work is not included.

Subcommand	Flags	Description
`start <name>`	`-m, --message <String>`	Reuse one open `Agent: <name>` session (make it current) or open one; more than one open match is refused, and a parked match must be resumed first. One to 64 characters with no control characters. The note applies only to a newly opened session.
`done`	`--name <name>`, `--id <changeset>`, `-m, --message <String>`	Submit an open agent session atomically and print its changelist id. Defaults to the current changelist but refuses one that is not an agent session; `--name` requires exactly one match. A final note keeps the `Agent:` header.
`status`	`--name <name>`	List open agent sessions, their changeset, staged file count and current marker.
`wait`	`--name <name>`, `--timeout <secs>` (default 600, 1–86400), `--interval <secs>` (default 2, 1–3600), `--require`	Block until no matching open session remains; exits 1 on timeout, or when a matching session is parked (`changelist parked resume` first). Without `--require`, a name that is not open yet completes immediately; `--require` makes that a failure (with no name it requires at least one open session).

```
swarmfile agent start claude-1 -m "tighten the parser"
# ...the agent works...
swarmfile agent done --name claude-1
swarmfile agent status
swarmfile agent wait --name claude-1
```

`agents`

`agents show` prints the current mount's `.swarmfile/agents.yml`; `agents init` writes a commented starter (refusing an existing file unless `--force`); `agents check` validates the TTL, conflict mode, branch prefix, concurrency, image, deny paths and upstream values and exits 2 on a bad file. The GitHub Action and MCP server read the same policy; the engine applies it on `mounts new` (TTL, conflict mode, `paths_deny`, max, checkpoint idle), the hooks use its TTL, and `merge` uses `merge_diff3` when `.swarmfile/merge.yml` is absent.

Subcommand	Flags	Description
`show`	`--json`	Print the policy as JSON instead of YAML.
`init`	`-f, --force`	Write the commented starter; overwrite an existing file.
`check`	`--json`	Validate the policy; JSON output when requested; exit 2 on errors.

```
swarmfile agents show --json
swarmfile agents init
swarmfile agents check
```

`hosted-agents`

Reads the hosted agent control API for the active organization: the non-human agent identities, their live delegations, and the branch each delegation is scoped to. This is separate from the local `agent` changelist sessions and the `agents` policy file. The control surface is the hosted hub only — a community or self-hosted server serves none of it. The same commands mint and revoke agents and delegations and start, exec, log and stop a hosted session.

Subcommand	Flags	Description
`list`	`--json`	List the organization's hosted agents with a live-delegation count.
`show <agent-id>`	`--json`	One agent and its delegations (branch scope, tier, role, creator, max-live, expiry, revoked).
`keys <agent-id>`	`--json`	The agent's live scoped keys (id, kind, delegation, expiry); members see only keys on their own delegations.
`create`	`--name <n>`, `--kind <k>`, `--json`	Create a non-human agent identity (owner/admin on the hub; the name must be unique).
`delegate <agent>`	`--branch <b>` or `--prefix agent/`, `--repository <id>`, `--tier branch`, `--role read|write`, `--ttl-days <1-30>`, `--deny-glob <g>` ..., `--max-live-keys <1-100>`, `--service-mintable`, `--service-user <sub>`, `--max-token-ttl <60-3600>`, `--json`	Delegate a repository branch or the reserved `agent/` workspace prefix. The repository defaults to the mounted one. An owner/admin may delegate any repository in the org; a member or guest needs the repository root write (be its creator or hold a whole-repository grant) and is capped at seven days.
`sessions <agent-id>`	`--json`	List the agent's sessions (state, delegation, stop reason); members see only sessions on their own delegations.
`key <agent>`	`--delegation <id>`, `--name <n>`, `--json`	Mint a durable scoped key for the delegation. The plaintext key is printed exactly once. Owner/admin only on the hub; a member instead uses `token` on their own service-mintable delegation.
`token <agent>`	`--delegation <id>`, `--branch <b>`, `--ttl-seconds <60-3600>`, `--name <n>`, `--deny-glob <g>` ..., `--json`	Mint a short-lived service token (shown once) on a service-mintable delegation. The delegation's own ceiling caps the request; it defaults to 900 seconds unless the delegation was created with a larger `--max-token-ttl` (the server's hard maximum is 3600).
`revoke-token <agent>`	`--delegation <id>`, `--token <id>`, `-y, --yes`, `--json`	Revoke one service token.
`session start <agent>`	`--delegation <id>`, `--budget-micro-cents <n>`, `--ports <a,b>`, `--json`	Start a hosted session (owner/admin; materialize the branch, mint a session key, start the runtime). The session key is printed once.
`session status <agent> <session>`	`--json`	The session state and stop reason (owner/admin, or the session's own session key).
`session stop <agent> <session>`	`-y, --yes`, `--json`	Checkpoint the workspace, revoke the key, stop and settle. Allowed for an owner/admin, the session's own key, or the delegation's creator/service user; a failed or absent checkpoint is reported and exits nonzero.
`session exec <agent> <session> -- <cmd> [args]`	`--json`	Run one command in the workspace (owner/admin, or the session's own key); prints stdout/stderr (or the hub's JSON) and exits 0 when the command succeeded, 1 on a failure or timeout, in both modes. It does not pass the command's own exit code through.
`session logs <agent> <session>`	`--after <seq>`, `--json`	The bounded session event log, oldest first (owner/admin, or the session's own key).
`disable <agent>`	`-y, --yes`, `--json`	Disable the agent; the hub revokes its live keys immediately.
`revoke-delegation <agent>`	`--delegation <id>`, `-y, --yes`, `--json`	Revoke a delegation; the hub stops its sessions and revokes its keys.
`revoke-key <agent>`	`--key <id>`, `-y, --yes`, `--json`	Revoke one scoped agent key.

```
swarmfile hosted-agents list
swarmfile hosted-agents show <agent-id>
```

Human rows print the agent, delegation, session and key ids in FULL, because those are exactly the values the next command takes; only informational ids such as a creator are shortened. `--json` prints the hub's object untouched. The engine's proxy passes the hub's real 401/403/404 through, so a permission or missing-agent refusal is reported as itself rather than a gateway error. A `402 plan_restricted` (for example service-token minting on the Free plan) exits 2, the same convention as the other CLI plan refusals.

A session checkpoint is clean only when the hub answers a 2xx `{ok:true}`. The web Agents view applies the same rule, so the terminal and the dashboard never disagree about whether a stopped session's changes landed.

`hooks`

Coding-agent harness integration: give each agent session its own mounted `agent/*` workspace without the agent knowing Swarmfile exists. `hooks install` merges the hooks into a harness's settings; `hooks run` is the wrapper form for any command line. A session that cannot mount (no engine, no FUSE, a refusal) says why on stderr and exits 0; `SessionStart` may spend up to 120 seconds creating the workspace and 60 seconds waiting for it to attach (the installed hook timeout is 240 seconds), and `hooks run` exits 2 if it cannot mount.

`hooks install`

Merge Swarmfile's hooks into a harness's own settings file: `SessionStart` opens the workspace and returns its path as context; `PostToolUse` (Write/Edit/MultiEdit/NotebookEdit/Bash/PowerShell; Codex also `apply_patch`) checkpoints after each mutating tool call and names the commit after the step; `Stop` is a turn-end backstop; and `SessionEnd` waits briefly, checkpoints, then archives the branch (or opens an MR).

Codex's 3-second SessionEnd closes the mount and leaves the archive to the TTL sweeper; `--mr` is refused for Codex hooks (use `hooks run --mr -- codex`). It also writes the harness's instructions file if absent.

Re-installing replaces Swarmfile's own hooks and leaves every other hook and an edited skill alone.

Argument / Flag	Description
`<harness>`	`claude-code`, `cursor`, `codex`, or `copilot`. Claude uses `.claude/settings.json` + a skill; Cursor uses `.cursor/hooks.json` + a rule; Codex uses `.codex/hooks.json` + a skill (which it asks you to review/trust via `/hooks`); Copilot has no lifecycle hooks, so it gets a `.github/agents/swarmfile.agent.md` profile (repository-scoped).
`--repository`	Write the repository-level file (default). Check it in; teammates need `swarmfile` on PATH and the default cache.
`--user`	Write the user-level file instead (`~/.claude/...`, `~/.cursor/...`, `~/.codex/...`); commands pin the absolute binary and cache. Not valid for `copilot`. Cursor user rules are UI-managed, so only its hooks file is written.
`--mr`	At session end, open a draft merge request and keep the branch instead of archiving it. Refused for `codex` hooks; use `hooks run --mr -- codex`.
`--label <L>`	Readable part of the workspace branch name (default: `claude`, `cursor`, `codex` or `copilot` for the chosen harness; 1–40 safe characters).
`--ttl-days <1-30>`	Workspace lifetime (default `.swarmfile/agents.yml`, then 7).
`--print`	Print the settings this would write, without touching the file.

For a self-contained prompt you can paste into an agent (version-pinned, with the verification steps), see Coding Agents § Copy-paste install prompt.

```
swarmfile hooks install claude-code
swarmfile hooks install cursor
swarmfile hooks install codex
swarmfile hooks install copilot
swarmfile hooks install claude-code --user --mr
```

`hooks run`

Mount a workspace, run the command with its cwd **inside** it, then checkpoint and finish the workspace. This is the reliable cwd path: a hook cannot change the agent's working directory. Installed hooks stand down while it runs (`SWARMFILE_AGENT_MANAGED`). It also exports the repository's `.swarmfile/cache.yml` redirects to the child, so installs/builds run by the agent use the shared dependency cache automatically.

Flag	Description
`--label <L>`	Readable part of the workspace branch name (default `agent`).
`--ttl-days <1-30>`	Workspace lifetime (default `.swarmfile/agents.yml`, then 7).
`--mr`	Open a draft merge request instead of archiving the workspace.
`-- <command…>`	The command to run (everything after `--`).

```
swarmfile hooks run -- claude
```

`hooks uninstall`

Remove Swarmfile's own hooks from a harness's settings (foreign hooks are preserved) and delete the generated instructions only if they were not edited. `--repository` (default) / `--user` select the scope; the harness argument is `claude-code`, `cursor`, `codex` or `copilot`.

`hooks doctor`

Check that the harness's settings contain Swarmfile's lifecycle hooks and that the instructions file exists. `--repository` (default) / `--user` select the scope. Exits `1` with the missing pieces; it does not contact the engine.

The internal hook entry points (`hooks session-start`, `hooks session-end`, `hooks checkpoint`) are installed into the settings, not typed by hand.

`switch`

Switch which branch this mount works on (like `git switch`) — hot and in-process, no engine restart. Same operation as `branch switch`. "This mount" is the `--mount` one, else the current mount (`mounts current`): the branch checks, `-c`'s create, and the switch itself all act on that same mount. A mount that shares another mount's branch context can't switch on its own and is refused (`shared_branch_context`) — `-c` refuses before creating anything. The success line names the branch the engine reports it switched to.

Argument / Flag	Description
`[name]`	Branch to switch to. Omit to return to the repository's default branch (usually `main`).
`-c, --create`	Create the branch first (forking from the current branch), then switch to it — like `git switch -c`. Errors if it already exists.

Refused while any changelist is open on that mount (exit 2 on the default mount; on any other mount the switch fails with the same reason, exit 1); submit or cancel all of them first.

```
swarmfile switch look-dev-b
swarmfile switch            # back to the default branch
swarmfile switch -c feature-x   # create + switch
```

`restore`

Restore a scope (the whole repository, a folder, an entry, or a mount-relative path) to a past commit — a true tree snapshot, landed as one undoable commit (like `git restore`, but whole-repository by default). Locked files → exit 2, and a file the restore would change that has an unresolved conflict refuses with `conflict_unresolved` — resolve it first.

Argument / Flag	Description
`[seq]`	A commit to restore to — `#N`, `HEAD~N`, a tag, a changeset id, or a 64-hex hash (see "Referring to a commit"). A **branch** name restores this mount's branch to that branch's head tree (it does not switch branches — that's `switch`).
`--to <ref>`	The same commit ref as `[seq]`, in the checkpoint spelling — `swarmfile restore --to 482`. Not with `[seq]` or `--at-seq`.
`--at-seq <N>`	Commit number to restore to — instead of `[seq]`, not with it (both at once is a usage error).
`--path <path>`	Restore only this mount-relative path.
`--folder-id <id>`	Restore a specific folder by ID.
`--entry-id <id>`	Restore a specific entry by ID.

```
swarmfile restore 482                    # whole repository back to commit #482
swarmfile restore 482 --path shots/seqA  # just that folder
```

`--json` prints the engine's response plus `targetSeq` — the commit number the CLI resolved your ref to; the engine's own `seq` is the NEW commit the restore landed as.

`checkout`

A Git-habit shim mirroring `git checkout`'s overloading — prefer `switch` / `restore`, which say exactly what they do.

You type	What happens
`checkout <branch>`	Switches to that branch (→ `switch`).
`checkout <N>` (a number, `HEAD`, `HEAD~N`)	Restores to that commit (→ `restore`).
`checkout <tag>` / `<hash>` / `<changeset-id>` / a prefixed ref (`tag:x`, `seq:5`, `hash:…`)	Restores to that commit — resolved exactly as `restore` resolves it. `branch:x` switches to `x`.
`checkout -b <name> [<target>]` (`-b, --branch <name>`)	Creates a branch and switches to it — like `git checkout -b`. Forks from `<target>` (a branch, tag, or commit — same rules as `branch create --from`) or `--at-seq <N>`, else from the current branch. `--path` / `--folder-id` / `--entry-id` don't combine with `-b` (a branch is whole-repository).
`checkout --at-seq <N> [--path …]`	Restores the version at that sequence number (the `--at-seq` form). Not combined with a `<target>`.

> A bare number restores that commit **unless a branch of that name exists** — an existing branch wins even when its name is commit-shaped, as in git, and the CLI prints a note when it does. Force the branch with `switch <name>`, or force the commit with a `hash:` / `seq:` prefix.

`show`

Show one commit: its message, author, time, and the files it changed (like `git show`). Whole-file — Swarmfile versions binaries, so there's no line-level diff.

Argument	Description
`<rev>`	A commit — number, `#N`, `HEAD`, `HEAD~N`, or a tag (see "Referring to a commit" above).

```
swarmfile show 482
swarmfile show HEAD        # the latest commit
swarmfile show HEAD~2      # two commits back
swarmfile show v1.0        # a tag
```

`show-tree`

Fetch and print a tree object by its content-addressed cid (like `git ls-tree`) — one level, not recursive: each entry's kind (`d` directory, `f` file, `l` symlink), name, and content cid. Get a cid from `swarmfile show <rev>`'s `tree` line.

A directory large enough to be stored segmented is listed a segment at a time — each segment is its own fetch, never one unbounded request for the whole directory.

A **repository member's** listing is filtered against the selected mount's branch view (the same ACLs a normal listing applies), so a cid taken from an old commit can come back partial or empty; owners and service callers see the object unfiltered.

Argument/flag	Description
`<cid>`	A tree object's content-addressed cid (64 lowercase hex characters).
`--audit`	Walk the tree and its child tree objects at their real depth and report symlinks that would escape. A tree stored before the object-PUT guard can carry one; a content address cannot be rewritten, so this is the read-only audit an operator acts on. Exits 1 on a finding.

```
swarmfile show-tree a1b2c3...
swarmfile show-tree a1b2c3... --audit   # audit a pre-guard tree's links (walk all child trees)
```

`revert`

Undo a commit by landing a **new** commit that restores the prior versions of the files it changed (like `git revert`) — non-destructive, itself undoable, and it doesn't touch newer commits. Locked files → exit 2, and a file the revert would change that has an unresolved conflict refuses with `conflict_unresolved` — resolve it first.

Argument	Description
`<seq>`	The commit to undo — a number (`#N`), `HEAD~N`, a branch, a tag, a changeset id, or a 64-hex hash (see "Referring to a commit").

```
swarmfile revert 482
```

`rollback`

Restore **one file** to an earlier version, keeping the current version in history — the CLI form of the Desktop App's version picker. File-scoped and non-destructive, unlike `revert`, which undoes a whole commit.

Requires the file's write lock, and refuses (exit 2) if the file has uncommitted local changes, is locked, has an unresolved conflict, or changed since the version was chosen. Get the version id from `swarmfile blame`.

Argument / Flag	Description
`<path>`	Mount-relative (or absolute) path to the file.
`--at <id>`	The version's local history id — the `#N` column from `swarmfile blame <path>` for a local row. Synthetic (`d1:<seq>`) rows have no local restore key and are refused; cold-archived rows keep theirs.

```
swarmfile blame shots/seqA/plate_v3.exr   # find the version id
swarmfile rollback shots/seqA/plate_v3.exr --at 4182
```

`describe`

Name, rename, or clear a commit's message after the fact — including an auto-saved commit that landed with no message (like `git commit --amend -m`, but for any commit and non-destructive). Author-or-admin only.

Argument / Flag	Description
`<ref>`	The commit to describe — the full ref vocabulary (`482` / `#482`, `HEAD~N`, a branch, a tag, a changeset id, or a 64-hex hash).
`-m, --message <String>`	The new message.
`--clear`	Remove the commit's message.

Exactly one of `-m` / `--clear` is required. Alias: `reword`.

```
swarmfile describe 482 -m "v2 delivery to client"
swarmfile describe 482 --clear
```

`diff`

Show what changed. With no argument (or a branch name) it previews what merging a branch into its base would change (like `git diff main...`); with two commits it shows the files that changed between them (like `git diff A B`); with one commit (any ref that isn't a branch) it shows what changed from that commit to the current branch head — git's `diff <commit>` compares against the working tree, and since saves sync as they happen here, the head is the closest equivalent. Whole-file: there's no line-level diff for a binary. Read-only — it writes nothing.

Argument	Description
(none) / `[branch]`	Branch preview — the given branch (or this mount's) vs its base. A branch preview is only meaningful for a non-main branch (main has no base → 400).
`<commit>`	That commit vs the current branch head — any non-branch ref (`HEAD~3`, a tag, a hash, `tag:x`, …).
`<A> <B>`	Two-commit diff — files changed between commit A and B. Each accepts the full ref vocabulary: a number, `#N`, `HEAD`, `HEAD~N`, a branch, a tag, a changeset id, or a 64-hex hash (prefixes force a kind).

`--format=unified` / `side-by-side` renders a branch preview's or commit range's content in the terminal; `auto` selects unified on a TTY and `stat` on a pipe. Side-by-side underlines the changed words within paired lines. Content reads stop at 8 MiB per file / 64 MiB total / 50k lines, and binary, encrypted-unavailable or oversized files are listed with the reason. Commit ranges carry both content CIDs in the hub response, so they render the same way.

```
swarmfile diff              # this branch vs its base
swarmfile diff look-dev-b   # a named branch vs its base
swarmfile diff HEAD~3 HEAD  # what changed over the last three commits
swarmfile diff v1.0 v2.0    # between two tags
```

`blame`

Who last changed a file, plus its version history with authors. The default is whole-file (one row per saved version). With `--lines`, Swarmfile derives line-level attribution from the saved versions: each line is credited to the version/writer that last introduced or replaced it. Content is read client-side, so this works on managed and end-to-end encrypted repositories. For E2E repositories the key never leaves the engine; managed repositories are decrypted by the engine with the repository key, which the hub manages.

Argument	Description
`<path>`	Mount-relative (or absolute) path to the file.
`--branch <name>`	Read this path's history on another branch (read-only) instead of the selected mount's — that branch's metadata is synced on demand, and the path must exist there.
`--lines`	Line-level attribution. Text files only (UTF-8, no NUL bytes).
`--max-versions N`	How many saved versions the line walk reads (default 100, range 1–500; out-of-range is refused). It is enforced at page granularity — a page can add up to 50 versions beyond the limit. Older lines outside the window are reported as unknown/truncated.
`--summary`	Print per-writer line counts instead of every line range. Requires `--lines`; ignored with `--json` (which prints the raw ranges).
`--include-text`	With `--lines --json`, also return the attributed head file content (`headText`) — the baseline an IDE uses for a quick diff. Requires `--lines`.
`--include-versions`	With `--lines --json`, also return every walked version's text (`versions`) — the historical sides an IDE diffs against. Requires `--lines`.

```
swarmfile blame shots/seqA/plate_v3.exr
swarmfile blame --lines src/foo.ts
swarmfile blame --lines --format=gutter src/foo.ts
swarmfile blame --lines --summary src/foo.ts
swarmfile blame --lines --json src/foo.ts
```

Line attribution is exact to the version's writer. Within one version pair, a rewrite larger than the bounded line alignment is credited to the newer version as a whole. To attribute separate agents exactly, run one engine per agent with a distinct `SWARMFILE_MACHINE_ID` (or a per-agent API key; a key is process-wide, so it still needs its own engine). A shared mount is one writer identity: the engine cannot tell which process inside it saved a line.

Lines present in a recorded merge/rollback parent inherit that parent's author; lines present in no parent belong to the merge/resolution actor.

On the hosted hub, clean branch merges record target/source CIDs and copy up to 30 source content-history rows (authors and parent links) into the target entry as local side rows for modified files and branch-only adds. The rows are only served when the caller can read the source entry.

The community server's history has no copied source rows, so a merged line there is credited to the merge actor. Source copy requires the retained window to reach a create/copy base; history beyond that window falls back. Branch-only add copies are base-aware; their old source history is capped at 30 versions. Staged changelist work is not included until it is submitted; `--lines` says so when it detects an open changelist on the requested mount. Parked and in-flight work is not detected by that flag.

The line view works against the hosted hub and against a community server's changeset history. The line view is capped at 8 MiB per version, 64 MiB total, 50,000 lines and 20,000 attribution ranges; a version open and each 4 MiB read slice have a 30-second timeout, and the whole read is bounded at 170 seconds. A deep/cold history may return `blame_timeout` — lower `--max-versions` and retry.

The first older version that cannot be read is listed under `unavailable` with a code (`binary_file`, `not_utf8`, `too_many_lines`, `file_too_large`, `content_unavailable` — including a timed-out version read — or `key_unavailable` when that version names a key generation this device does not have yet).

Synthetic versions (`d1:<seq>`, no local history row) are line-blame only and are not locally restorable; cold-archived rows keep their local restore key.

`changesets list`

The commit log, newest first, scoped to the branch this mount is on (or another named branch, read-only).

Flag	Description
`--limit <N>`	Default `50`
`--offset <N>`	Default `0`
`--branch <name>`	Read another branch's log without switching mounts. Omitted = the selected mount's branch.

`branch`

Branch management.

Subcommand	Flags	Description
`gc`	`--older-than-days <DAYS>`, `--dry-run`, `-y, --yes`, `--limit <N>`, `--repository <ID>`	Archive expired ephemeral workspaces now (manual expiry sweep). Owner/admin only; only `ephemeral` rows are touched. `--dry-run` lists the due rows; a real sweep requires `--yes`; `--older-than-days` (default 0), `--limit` (1–500, default 20) and `--repository` (defaults to this mount's repository) bound it.
`list`	`--all`	Active branches for this mount's repository. Each row carries its head commit's hash (`headCommitHash`; `null` when no head hash is recorded), creator, age/expiry and `kind`. Ephemeral `agent/<label>-<6 hex>` workspaces are hidden unless `--all`; a bare `agent/foo` stays visible.
`create`	`<name>`, `--from <REF>`, `--from-commit <seq>`, `--from-tag <tag>`	Fork a new branch (defaults to forking from the selected mount's branch's live head, like `switch -c`). `--from` takes the full ref vocabulary: a branch forks that branch (an empty one included), a tag forks its commit, and a seq/hash/changeset id forks that commit with the source branch derived from it. `--from-commit` / `--from-tag` are the explicit forms; combining a **branch** `--from` with one of them is allowed and the hub refuses the fork if the point belongs to a different branch (`branch_mismatch`), while a second explicit point (a tag/seq `--from` plus `--from-commit` / `--from-tag`) is refused up front.
`switch`	`[name]`	Switch which branch this mount (`--mount`, else the current mount) works on. Omit `name` to switch back to the repository's default branch (usually `main`).
`archive`	`<name>`	Soft-archive a branch (the repository's default branch cannot be archived). Exit 2 on an actionable refusal: unknown branch, the repository's default branch, a protected branch archived by a non-admin, a branch another **active** branch is based on (`branch_has_active_children`), one still named by an **open** merge request (`branch_has_open_mr`), a branch with a merge in progress (`merge_in_progress`, 409), or a branch in an archived repository (`repository_archived`, 403). A merged/closed MR doesn't block, so `mr merge --delete-branch` still works.
`merged`	—	Branches with nothing left to land — a branch-vs-base diff with no changed files and no conflicts (the same computation `diff <branch>` shows). The repository's default branch is never listed, a branch whose diff can't be computed is excluded rather than assumed merged, and a branch an open mount is working on is marked *in use*.
`rename <name> <new-name>`	—	Rename/promote a branch. The branch id is stable, so history, comments, MRs and locks follow; an ephemeral workspace becomes a named branch and loses its TTL. A protected branch requires an org admin (exit 2). The hub refuses a rename into or out of the reserved `contrib/` namespace (fork proposals) with `contrib_branch_reserved`.
`restore <name>`	`--ttl-days <1-30>`	Restore an archived branch. A restored ephemeral workspace gets a fresh TTL (the hub's configured `EPHEMERAL_TTL_SECS`, otherwise 7 days; `--ttl-days` overrides); a named branch is TTL-free.
`prune`	`-y, --yes`, `--include-ephemeral`, `--older-than <30m|12h|7d>`, `--owned-by <user-id>`	Archive every prunable branch `merged` lists, after naming them and confirming. **Never archives a branch an open mount on this machine is on.** Ephemeral `agent/<label>-<6 hex>` workspaces are skipped unless `--include-ephemeral` is given (their normal cleanup is `mounts close --archive`). `--older-than` uses last activity/creation time; `--owned-by` takes a user id or `me`; both fail closed when metadata is missing. `--yes` skips the prompt; `--json` without `--yes` returns a `confirmation_required` error instead. The result lists `archived` / `refused` / `failed` by name. Exits 2 if any branch was refused (protected, etc.), 1 on a transport error.
`protect`	`<name>`, `--required-approvals <N>`	Refuse a direct merge into this branch **and every direct write to it** (saves/edits/renames/moves/creates/deletes, staging/committing, rollback, conflict resolution, LFS reflect) — changes land only through an approved Merge Request. `--required-approvals` (1-10, default 1) sets this branch's approval floor — an unprotected branch otherwise requires none; a matching protection rule's own count can raise it further (the effective count is the MAX of both). The CI-checks-required gate (`--require-checks-to-pass`) is a `protection-rule`-only flag, not available per-branch here — protect via a single-branch (no-wildcard) rule instead if you need it without a pattern. Org-admin-gated.
`unprotect`	`<name>`	Clear a branch's protected flag. Org-admin-gated, same as `protect`.
`backfill`	`--dry-run`, `--ttl-days <1-30>`, `-y, --yes`, `--json`, `--repository <ID>`	Classify existing `agent/<label>-<6 hex>` branches that predate the expiry lifecycle as ephemeral so the expiry reaper manages them. Dry-run by default; `--yes` writes the classification and `--ttl-days` sets the new expiry (default 7, or the deployment's `EPHEMERAL_TTL_SECS` / `SWARMFILE_EPHEMERAL_TTL_SECS`). Classification is capped (repository plus per-user/per-key, `overflow`); the CLI follows `nextAfter` and `--json` emits one aggregate envelope with `truncated`. Owner/admin only.
`set-commit-mode`	`<name>`, `[mode]`	Lock (or unlock) this branch to a commit mode — overrides the repository's own lock, if any, while set. `mode` is `auto-commit`, `staged`, or omitted/`unset` to clear it back to unenforced-at-branch-scope (the repository's lock, if any, still applies). Already-open sessions/changelists are grandfathered: this only affects the *next* `commit` / `changelist open`, never one already in progress. Org-admin-gated: exit 2 on `forbidden` (403) or an unknown branch (404); any other failure is exit 1.

> `branch switch` is hot — no engine restart. It's refused (exit 2) while any changelist is open; submit or cancel all of them first. With the global `--mount <id>`, it retargets **that** mount only (a mount sharing another mount's branch context can't switch on its own and is refused with `shared_branch_context` — open an independent mount instead); a branch that doesn't exist can't be switched to. `branch create` never touches the running mount either, even with `--from-commit` / `--from-tag` — creating a branch doesn't change which branch this mount is checked out to. `protect` / `unprotect` exit 2 on `forbidden` (non-admin caller), `not_found` (unknown branch), or (`protect` only) an out-of-range `--required-approvals` (400) — distinguishable business states, not a crash — and exit 1 on anything else.

```
swarmfile branch create recover --from-commit 482
swarmfile branch create recover --from-tag v1.0
```

`protection-rule`

Glob-pattern branch protection — additive on top of `branch protect` / `unprotect` 's per-branch flag. A rule like `release/*` protects every branch whose name matches it, including ones created after the rule was added; a pattern with no wildcards is just another way to protect one exact branch.

Subcommand	Flags	Description
`list`	—	Protection rules for this mount's repository. Org-admin-gated, like `create` / `delete`: exit 2 on `forbidden`.
`create`	`<pattern>`, `--required-approvals <N>`, `--require-checks-to-pass`	Create a rule. `*` matches any run of characters, `?` matches exactly one. `--required-approvals` (1-10, default 1) is this rule's contribution to the approval floor for every branch it matches — see `branch protect`'s row above. `--require-checks-to-pass` refuses a merge unless every CI check that has reported a run at the request's exact current head shows its latest run passing. A check whose latest run is at an earlier commit is not carried forward as still-required; if no check has reported at the head at all, the gate fails closed rather than passing vacuously. This flag exists only here, at the rule level; there's no per-branch equivalent on `branch protect` itself. Org-admin-gated: exit 2 on `forbidden` (403), `invalid_pattern` / `invalid_required_approvals` (400), or `pattern_taken` (409); exit 1 on anything else.
`delete`	`<pattern>`	Delete a rule by its exact pattern text. Exit 2 on `forbidden` or `not_found`, same convention as `create`.
`add-reviewer`	`<pattern>`, `<principal-id>`	Add an auto-reviewer — every merge request opened against a branch this rule matches automatically requests a review from this principal. Additive to manually-requested reviewers, never a replacement. Idempotent. Org-admin-gated.
`remove-reviewer`	`<pattern>`, `<principal-id>`	Remove an auto-reviewer from a rule. A no-op if they weren't one. Org-admin-gated.
`list-reviewers`	`<pattern>`	List a rule's configured auto-reviewers. Human output shortens each id (`user 1a2b3c4d…`); `--json` carries the full ids that `add-reviewer` / `remove-reviewer` take. Org-admin-gated.

```
swarmfile protection-rule create "release/*" --required-approvals 2 --require-checks-to-pass
swarmfile protection-rule list
swarmfile protection-rule add-reviewer "release/*" <principal-id>
swarmfile protection-rule list-reviewers "release/*"
swarmfile protection-rule delete "release/*"
```

`tag`

A repository-scoped, immutable named pointer to a commit — "branch from tag" is "branch from commit, resolved via a name." A tag also pins the commit it names against history retention, so checkout/restore to it keeps working while the tag exists; deleting the tag releases the pin. There is no move/repoint subcommand: to repoint a name, delete it and create it again. (With git access on the repository, a lightweight tag can also be moved or deleted over the git remote — see Cloning a Repository with Git.)

Subcommand	Flags	Description
`list`	—	Tags for this mount's repository. Each row carries the tagged commit's hash (`commitHash`; `null` when no hash is recorded for the tagged commit).
`create`	`<name>`, `[REF]`, `--at <ref>`, `--at-seq <seq>`, `--on-branch <branch>`	Tag a commit. `REF` takes the full vocabulary (`N` / `#N`, `HEAD~N`, a branch, a tag, a changeset id, a 64-hex hash; prefixes force a kind); `--at-seq N` is the bare-number shorthand. Omit the ref to tag the CURRENT head of the selected mount's branch, or of `--on-branch` (which must name an existing branch — checked with every form, `--at-seq` included). Errors if the branch has no commits yet. Needs repository-wide write (see Permissions); exit 2 on that refusal (403).
`delete`	`<name>`	Delete a tag. Real delete, not soft — a tag has no archived state. Same access and exit 2 on 403 as `create`.

```
swarmfile commit -m "final render pass"
swarmfile tag create v1.0
swarmfile tag create v0.9 --at-seq 482
swarmfile tag list
swarmfile tag delete v1.0
```

`query`

A fixed-filter query over branches, commits, changes, or entries. Predicates are space-separated: `type:branch|commit|change|entry`, `branch:<glob>`, `author:<me|id>`, `since:<unix>`, `until:<unix>`, `text:<substring>`, `path:<glob>`, `limit:<N>` (1–500), and `order:name|time`. ACLs stay hub-side.

Predicates are type-checked: `branch:` applies to `type:branch` and `type:commit` (a commit glob expands through the branch catalog, bounded to 50 branches, and merges by commit sequence); `text:` applies to commits and changes; `path:` applies to entries only.

`type:change` is the repository activity feed (every activity kind), has no branch/path field, and its `since:` / `until:` are forwarded to the hub's window. `type:entry` is the mounted repository's **current (root) folder** listing; `path:` on it is a name glob (there is no recursive path walk yet).

`author:<id>` asks the hub for the enriched `lastModifiedById` (the raw idp subject); `lastModifiedBy` is a display name for rendering only. `author:me` needs a signed-in OIDC identity — an API-key session has no user id and exits with `author_me_unresolved` (use `author:<id>`). Entry `author:` needs the hosted enriched listing; a community or older hub returns `enrichment_unavailable` (use `type:commit` there; the community cut does not serve the authenticated activity feed).

`order:name` applies to branches and entries; `order:time` sorts entries by mtime/ctime, changes by their activity time, and commits by their commit time.

`--json` emits `{kind, items, nextCursor, truncated}`; the human form prints the matching rows. `nextCursor` is the last item's key (informational — the grammar has no cursor predicate yet). One local changeset page is capped at 200 and one activity page at 100, so `truncated` can be true even when fewer than `limit:` rows are returned; a 50-branch glob that drops matches also sets it. Entry `since:` uses the entry's mtime/ctime fields, which are best-effort metadata. `type:entry` lists one page (up to 500 rows); `truncated` reports the cap. `type:change` uses the hub's default 7-day activity window unless `since:` / `until:` is given.

```
swarmfile query "type:commit branch:main since:1790000000 limit:20"
swarmfile query "type:change since:1790000000 limit:50"
swarmfile query "type:entry path:*.rs author:<id>"
swarmfile query "type:branch author:me order:time" --json
```

Flag	Description
`--json`	Emit the machine-readable result instead of the summary.
`--mount <id>`	Query through this mount's repository instead of the current one.

`ref`

A thin read-only grouping over branches **and** tags — for the two questions that need both listings at once. `branch` and `tag` remain the full commands for acting on either.

Subcommand	Flags	Description
`list`	—	Every named pointer: branches (the current one marked `*`) and tags, together — each carrying its head/tagged commit hash.
`resolve <name>`	—	Say what a ref resolves to under the same resolver every other command uses — the kind (`branch`, `tag`, `seq`, `head`, `hash`, `changeset`) and commit number `restore` / `checkout` / `log` would act on — with a note (`alsoMatches` in `--json`) when the name also matches another kind: a branch wins over a same-named tag, and `tag:<name>` selects the tag. Exit 1 if it doesn't resolve.

```
swarmfile ref list
swarmfile ref resolve v1.0
```

`release`

Publish a tag as an immutable, public release — a static, cached snapshot of that tag's tree, served anonymously for browsing and for full downloads (a quick proof-of-work check per download, no account needed).

Requires the repository to already be `plaintext`-tier and `public` (every public repository is plaintext-tier, on any plan — see Billing and plans); publishing enqueues a combined-archive build, so a fresh release starts `pending` and moves to `ready` (or `failed`) asynchronously.

`list` / `create` / `preflight` are scoped to this mount's own repository; `fetch` is not — it pulls from ANY public repository by org/repository slug, since a public release needs no org membership and no account.

A release can also carry **assets**: named binaries attached to it without living in the repository tree (no manifest entry, no combined archive, no clone history). They are served anonymously at `/public/<org>/<repository>/assets/<name>` on the hub, which is how build artifacts ship alongside the tagged source; `attach` uploads one, `assets` lists them, `detach` removes one, and re-attaching the same name replaces it.

Subcommand	Flags	Description
`list`	—	Releases published for this mount's repository, each with the `id` `delete` takes, its `status` (`pending` / `ready` / `failed`), and the tagged commit's `commitHash` (`null` when no hash is recorded for the tagged commit).
`preflight`	`<tag>`	Report exactly what `create <tag>` would do, publishing nothing: the gates (public/ `plaintext` , your authority, verified email), the file count and size that would become public, the paths `.swarmfile/publicignore.yml` keeps out, the paths `.gitignore` / `.swarmfileignore` hide from the web browser that would *still* publish, and whether the tag's policy differs from the default branch's current one (the fixed-the-policy-but-didn't-re-tag trap). Exit 0 when the tag is publishable, 2 when it isn't (the reason is printed, including `publicignore_clone_exposure` when the release would hide paths while anonymous cloning is on — turn anonymous cloning off, then retry), 1 when the check itself failed.
`create`	`<tag>`	Publish a tag. Re-running this on an already-published tag is a no-op unless the prior attempt failed non-permanently, in which case it retries. Exit 2 on a defined refusal: not `plaintext` + `public` (409), the tag's prior publish failed permanently (409 — publish a new tag instead; a terminal `publicignore_invalid` from a committed `.swarmfile/publicignore.yml` lands here, so fix the file and publish a new tag), publishing a release that hides paths while anonymous cloning is on (409 `publicignore_clone_exposure` — turn anonymous cloning off with `swarmfile git anonymous-clone off` , then retry; once the release is ready, turning it back on does not re-expose those paths), the tag doesn't exist (404), the repository isn't provisioned on the hub yet (501), or you lack the access to publish (403 — see Permissions; this includes a folder in the tag you're denied read on, and an unverified email).
`delete`	`<id>`	Unpublish. Real delete, not soft, and needs the same access as `create` (see Permissions). Find `id` via `list` 's output first. Exit 2 on `404` (already unpublished, or a typo'd id) or `403` (you lack the access to publish, so you can't unpublish either).
`attach`	`<tag> <file>` , `--name <name>` , `--content-type <type>`	Upload a local file as an asset of the release named by `tag` (see `list`). The file's basename is the asset name unless `--name` overrides it; `--content-type` overrides the type derived from the file name. The engine computes the SHA-256 the hub displays. Re-attaching the same name replaces it. Files up to 100 MiB stream through the hub; larger files (up to 5 GiB) upload directly to the repository's storage with a presigned URL. Exit 2 on a defined refusal: a bad name (400), no authority (403), a missing release (404), too many assets (409), a file over the 5 GiB cap (413), or a repository bucket that cannot accept a direct upload when the file is over 100 MiB (409 `presign_unavailable`).
`assets`	`<tag>`	List the release's attached assets (name, size, content type, SHA-256).
`detach`	`<tag> <name>`	Remove one attached asset. Exit 2 on `404` (no such asset) or `403` (no authority).
`fetch`	`--org <slug>` , `--repository <slug>` , `--tag <name>` , `--path <path>` , `--out <path>` , `--into <org>/<repository>`	Fetch one file from the release's manifest, or (if `--path` is omitted) the whole release as a combined `.tar` . No account is needed: the download passes a one-time proof-of-work check. `--org` / `--repository` are slugs, matching the public URL shape, not internal ids. With `--into` , nothing is written to disk: the release is copied straight into that repository by the engine — the same client-side copy as `clone … --into` (see Version control), which seals the blocks on this machine so a **private** destination works — `--out` is then not required, and `--path` narrows the copy exactly as it narrows an archive. The copy runs as a long operation (`--no-wait` , `swarmfile op status`

Browsing a published release anonymously (no account, no CLI) is a web-hosted page, not a CLI surface — the same split `share` describes for share links.

```
swarmfile tag create v1.0
swarmfile release preflight v1.0
swarmfile release create v1.0
swarmfile release attach v1.0 dist/widgets-1.0-macos-arm64.tar.gz
swarmfile release assets v1.0
swarmfile release detach v1.0 widgets-1.0-macos-arm64.tar.gz
swarmfile release delete rel_9f2a
swarmfile release list
swarmfile release fetch --org acme --repository widgets --tag v1.0 --out widgets.tar
swarmfile release fetch --org acme --repository widgets --tag v1.0 --path docs/readme.md --out readme.md
swarmfile release fetch --org acme --repository widgets --tag v1.0 --into myorg/private-widgets
```

`merge`

Merge this mount's branch into the branch it was forked from — its parent (usually main, but a branch forked from a non-main branch merges back into that one). The target is implicit: a branch already knows its own parent, so there's no target flag. A merge that would change an entry with an unresolved conflict is refused with `conflict_unresolved` — resolve it first, then re-run.

Flag	Description
`--from <branch>`	Merge FROM this branch instead of the selected mount's own — land a branch you aren't standing on. The target is still the source's parent (main, or whatever it was forked from).
`--record-conflicts`	Record each conflict for interactive resolution (the Conflicts page or `conflicts`) instead of only listing them.
`--diff3`	On a merge conflict, batch-resolve every text-file content conflict with the engine's three-way merge before giving up, then retry the merge once. Never lands a partial merge on its own. Overrides a repository's `.swarmfile/merge.yml` `diff3: false` . Mutually exclusive with `--no-diff3` .
`--no-diff3`	Force diff3 auto-resolve **off** for this invocation, even if the repository's `.swarmfile/merge.yml` declares `diff3: true` . Mutually exclusive with `--diff3` .
`--dry-run`	Print what the merge would change — added, modified, deleted, renamed, and conflicting files, the same read-only diff `swarmfile diff <branch>` shows — and stop. Nothing is staged and no merge lock is taken.
`--wait`	Large merges apply in the background; the command otherwise returns as soon as the merge is accepted. Block and poll until the apply finishes, printing each phase. Gives up after ten minutes with exit 2 and explains how to resume or roll the merge back.

```
swarmfile branch create feature-x
swarmfile branch switch feature-x
# ...work, commit...
swarmfile branch switch
swarmfile merge --record-conflicts
```

`mr`

Open, list, inspect, review, and land Merge Requests — the propose → review → approve → merge gate that sits on top of `merge` above.

Subcommand	Flags	Description
`create`	`--title <title>`, `--from <branch>`, `--body <text>`, `--draft`	Open a merge request from `--from` into its base branch. `--from` defaults to whatever branch this mount is currently on. Prints the branch-vs-base diff preview (same as `diff <branch>`) before opening it, so you see what a reviewer will see. `--draft` opens it as a draft — reviewable and commentable, but `mr merge` refuses it, and the usual "new merge request" notification never fires for it at all (not deferred — marking it ready later doesn't retroactively send the one it skipped).
`list`	`--status <open|merged|closed|all>` (default `open`), `--limit <N>`, `--label <label-id>`, `--assigned-to <principal-id>`	Merge requests for this mount's repository. `--limit` is a hint the hub caps at 500 regardless. `--label` / `--assigned-to` filter server-side — see `label list` /assignees, requested reviewers, and labels for where the ids come from. `--assigned-to` matches the `assignee` role specifically, not a requested reviewer.
`status`	`<number>`	Branch names, who opened it, approval state, per-reviewer verdicts, assignees/requested-reviewers/labels, and changed-file count for one merge request, by its display number (the "3" in "MR-003"). Exit code doubles as a CI gate: `0` approved, `2` open but not yet approved, `1` any other error. It also prints the source stamp (`h1:…`) for the content it showed; `--json` carries it as `sourceContentStamp`.
`diff`	`<number>`	The merge request's live diff — the files it would change (A/M/D/R) plus any conflicts, rendered exactly like `diff` above. Read-only, recomputed fresh by the hub on each call. If the MR's source branch was archived/deleted the hub has no diff and this says so, rather than printing an empty list that reads as "no changes."
`approve`	`<number>`, `--note <text>`, `--acknowledge-no-preview`, `--source-stamp <h1:…>`, `-y, --yes`, `--no-bind`	Approve a merge request. Refused (`self_review`) if you opened it yourself. When the merge request is too large to preview (`mr diff` says so), approving means approving changes you haven't seen, so it is refused (`preview_too_large`, non-zero exit) unless you pass `--acknowledge-no-preview`; otherwise review it in the web app. `--source-stamp` binds the approval to the content `mr status` showed you: if the source branch changed since, it is refused (`source_changed`, exit `2`: the merge request changed since you viewed it, so review it again). Without the flag, an interactive run prints which merge request and branches it is approving and the stamp it binds to, then asks `Send this approval? [y/N]` (`--yes` skips the question). If the read fails it warns and refuses (exit `2`) unless you pass `--no-bind`, so an approval is never sent unbound silently; `--yes` sends it unbound only when the hub reports no stamp at all. `--no-bind` sends it unbound on purpose. A script (no terminal, or `--json`) without `--source-stamp` or `--no-bind` is refused (exit `2`) unless `SWARMFILE_MR_ALLOW_UNBOUND=1` is set.
`request-changes`	`<number>`, `--note <text>`, `--source-stamp <h1:…>`, `-y, --yes`, `--no-bind`	Request changes on a merge request — blocks `merge` below until a later review from the same reviewer approves it, or an org owner/admin dismisses it (`dismiss-review`); a new commit does not lift it. Same self-review restriction as `approve`, and the same `--source-stamp` / `--yes` / `--no-bind` binding rules (the stamp is recorded with the review).
`review-comment`	`<number>`, `--note <text>`	Leave a non-blocking "Comment" review — shows up in the review history but never counts toward the approval floor and never blocks `merge`. Unlike `approve` / `request-changes`, allowed on your own merge request. Distinct from `comment` / `comments` below, which post to the discussion thread, not the formal review history.
`dismiss-review`	`<number>`, `<review-id>`, `--reason <text>` (required)	Dismiss one approval or request for changes so it no longer gates `merge` — a "request changes" otherwise blocks until that reviewer reviews again (a new commit does not lift it). Org owners and admins only (`dismiss_forbidden`), on any open merge request, their own included. The review id is `reviews[].id` in `status --json`.
`ready`	`<number>`	Mark a draft merge request ready for review — lifts `merge`'s draft refusal, and notifies the merge request's current assignees and requested reviewers (`mr_ready`). Opening it as a draft suppressed the one-time "new merge request" notification, which is not re-sent. One-directional: there's no way back to draft. Refused (`not_draft`) if it isn't currently a draft, or (`invalid_state`) if it isn't open.
`merge`	`<number>`, `--record-conflicts`, `--delete-branch`, `--diff3`, `--no-diff3`	Land an approved merge request — same conflict detection as `merge` above, including the `--diff3` / `--no-diff3` override flags. Refused with `not_approved` unless enough reviewers have approved (zero on an unprotected branch; a protected target branch or rule raises the floor, 1-10), nobody's latest verdict is "request changes," and (if the target requires it) CI checks pass. Refused with `is_draft` while the merge request is still a draft — run `mr ready` first. `--delete-branch` archives the source branch once the merge has actually landed — opt-in, off by default; a failed archive (e.g. the branch got protected in the meantime) never fails the merge itself, just prints a warning.
`close`	`<number>`	Close a merge request without merging it.
`reopen`	`<number>`	Reopen a closed merge request. Refused (`invalid_state`) unless it's currently closed — a merged merge request can never be reopened.
`comment`	`<number>`, `-m, --message <text>`	Post a comment to a merge request's discussion thread — the same thread the web dashboard's merge request page shows.
`comments`	`<number>`	List comments on a merge request's discussion thread.
`assign`	`<number>`, `<principal-id>`	Assign a principal (user or group id — see `admin nodes` /the web dashboard's people picker for ids) to land this merge request. Idempotent: assigning someone already assigned is a no-op, not an error.
`unassign`	`<number>`, `<principal-id>`	Unassign a principal. A no-op if they weren't assigned.
`request-review`	`<number>`, `<principal-id>`	Request a review from a principal — distinct from `assign` (who lands it). Idempotent.
`unrequest-review`	`<number>`, `<principal-id>`	Withdraw a review request. A no-op if none was pending.
`label`	`<number>`, `<label-id>`	Attach a label (from `label list` below) to this merge request. Idempotent.
`unlabel`	`<number>`, `<label-id>`	Detach a label. A no-op if it wasn't attached.

```
swarmfile mr create --title "Warmer facade material"
swarmfile mr create --title "WIP: new roof profile" --draft
swarmfile mr ready 4
swarmfile mr list
swarmfile mr list --label <label-id> --assigned-to <principal-id>
swarmfile mr status 3
swarmfile mr diff 3
swarmfile mr approve 3
swarmfile mr request-changes 3 --note "needs another pass"
swarmfile mr merge 3
swarmfile mr close 3
swarmfile mr reopen 3
swarmfile mr comment 3 --message "Looks good once the fascia texture is swapped."
swarmfile mr assign 3 <principal-id>
swarmfile mr request-review 3 <principal-id>
swarmfile mr label 3 <label-id>
```

`approve` / `request-changes` / `merge` / `close` put every review action a terminal command away. The hub refuses a review from the merge request's own creator; an approval from a second credential counts, so treat review-capable credentials like any other approval authority in your org. `branch protect --required-approvals` raises the bar for a unilateral change.

`label`

A repository's label registry — the tags `mr label` / `mr list --label` above attach to and filter by. See assignees, requested reviewers, and labels.

Subcommand	Flags	Description
`list`	—	This mount's repository's label registry.
`create`	`<name>`, `--color <#RRGGBB>`	Define a new label. Org admin/owner only.
`update`	`<label-id>`, `--name <name>`, `--color <#RRGGBB>`	Rename and/or recolor a label in place — the alternative to delete-and-recreate, which would drop every existing merge request's attachment. Both flags optional; an omitted one is left unchanged. Org admin/owner only.
`delete`	`<label-id>`	Delete a label — cascades off every merge request it was attached to. Org admin/owner only.

```
swarmfile label list
swarmfile label create bug --color "#FF5733"
swarmfile label update <label-id> --name defect --color "#000000"
swarmfile label delete <label-id>
```

`rfi`

RFIs and Submittals — the AEC request-for-information / approval workflow, the same feature the dashboard's RFI panel drives. Every command here (other than `create` / `list` / `search` / `by-file`) takes an RFI, and you can name it either way: the friendly display number (`RFI-003`, case-insensitive — resolved through the mounted repository) or its internal `id` from `rfi create`'s or `rfi list`'s output. Resolving a number needs a mounted repository and the same read access the detail view has.

Subcommand	Flags	Description
`create <path>`	`--title`, `--body` (required); `--priority <low|normal|high|urgent>`; `--due-at <unix-s>`; `--assigned-to <id>`; `--reviewer <id>` (repeatable); `--kind <rfi|submittal>` (default `rfi`); `--draft`	Open an RFI (or, `--kind submittal`, a Submittal) rooted at a mount-relative folder — same path resolution as `comment` / `watch`. Needs `write` there. `--draft` creates it unsubmitted; call `submit` when ready.
`list`	`--status`, `--priority`, `--ball-in-court <originator|reviewer|both>`, `--kind`, `--assigned-to`, `--due-before <unix-s>`, `--outstanding`, `--limit`, `--offset`	RFIs/Submittals matching the filters. `--outstanding` narrows to the same still-needs-attention set (excludes `closed` / `void` / `draft`) the file-browser's badge uses.
`status <id>`	—	Full detail, including decision history. Exit `0` if `answered` / `closed` (someone has responded), exit `2` for every other state (`draft` / `open` / `in_review` / `void`) — a CI/pre-merge gate can check "has this been answered."
`update <id>`	`--title`, `--body`, `--priority`, `--due-at`, `--assigned-to`, `--ball-in-court`	Change mutable fields — every flag optional, only the ones given change. Originator-or-owner only.
`submit <id>`	—	`draft → open`. Originator-or-owner only.
`acknowledge <id>`	—	`open → in_review` — a reviewer marking they've seen it. Assigned-reviewers only.
`close <id>`	—	`answered → closed`. Originator-or-owner only.
`void <id>`	—	`any → void` — abandon it (terminal, leaves the number gap). Originator-or-owner only.
`decide <id> <decision>`	`-m, --message <text>` (required); `--signature-kind <click|drawn>`; `--signature-blob`; `--comment-id`	Record a decision — an RFI's answer, or a Submittal's review outcome. `decision` is one of `approved`, `rejected`, `revisions_requested`, `answered`, `commented`, `approved_rev_a`, `approved_rev_b`, `approved_as_noted`, `rejected_as_noted` — the last four are Submittal-only; the hub doesn't itself reject a mismatched kind, so check `status <id>`'s `kind` first. Comment-or-above ACL.
`decisions <id>`	`--limit`, `--offset`	The decision audit trail, oldest first.
`assign <id>`	`--add <principal-id[:role]>` (repeatable), `--remove <principal-id>` (repeatable)	Add/update or remove reviewer-role assignments. `role` is `reviewer` (default)/ `observer` / `originator`. Admin-gated.
`attachments <id>`	—	Files attached under the RFI's root.
`search <query>`	`--repository-id`, `--limit`	Full-text search over title and body, ACL-filtered to what you can read. Defaults to the mounted repository; pass `--repository-id` to search a repository this engine isn't mounted on.
`by-file <path>`	—	Resolve the RFI (if any) that owns a file — the CLI equivalent of the dashboard's right-click "Open RFI thread."

```
swarmfile rfi create ./Repositories/tower-a/foundations --title "Slab thickness at grid C4" --body "Drawing S-201 shows 300mm, spec calls for
350mm — which governs?" --priority high --reviewer user_88
swarmfile rfi list --outstanding
swarmfile rfi status rfi_9f2a
swarmfile rfi decide rfi_9f2a answered -m "350mm per spec — drawing will be revised"
swarmfile rfi status rfi_9f2a   # now exits 0
```

`ec-placement`

Deliberate erasure-coded shard placement across LAN peers.

Subcommand	Description
`status`	Whether this mount currently has deliberate LAN shard placement on.
`enable`	Turn on deliberate LAN shard placement.
`disable`	Turn off deliberate LAN shard placement.

> `ec-placement enable` / `disable` restart the engine to take effect.

Placement spreads shards across your **LAN peers**, which are normally found by mDNS. If your network blocks mDNS multicast (common on corporate/VLAN'd networks), the office won't be discovered and every shard falls back to this machine — set `lan_from_office` (see Engine Config File) so same-office peers count as LAN.

`shards <path>`

Where a file's erasure-coded shards live across the office — for each piece, the machines it's assigned to and whether this one holds it. Requires `ec-placement enable` and a live P2P fabric. The command opens with a per-file summary: how many chunks and shards the file has, whether this machine holds enough shards (k per chunk) to reconstruct it alone, and whether placement actually assigned every shard to K peers — so "is this file protected in the office, and from where" is one line instead of counting rows.

Day-to-day

`status`

This mount's engine, peers, and settings at a glance, including a `drive:` line (`mounted`, `starting`, or `failed` with its reason). The text is human-oriented — use `--json` for scripts. It also prints the mounted branch's head commit pin when one is known (`head commit: <12-char> (<full hash>)`) so a script can capture the exact commit this workspace is on.

An engine started with `SWARMFILE_CONFLICT_MODE` pinned prints the mode it pinned (`conflict mode: hard (pinned by SWARMFILE_CONFLICT_MODE)`); per-drive pins show in `mounts list` / `mounts status` instead.

`pending uploads` counts this mount's repository. When another repository still has saves uploading (one you switched away from keeps uploading in the background), the line adds `(N more in other repositories)` and one indented line per such repository follows, named when the engine has seen the repository's name and by id otherwise, with any saves held there by a refusal. `--json` carries the same as `pendingUploadsAllRepositories` and `pendingUploadsByRepository`.

`--changes` swaps that for a Git-status-like summary of what's outstanding on this mount: the current changelist's staged files and any blocked conflicts. There's no "modified but unsaved" state to show — saves either stage into the changelist or commit immediately — so "staged" plus "conflicted" is the whole picture.

```
swarmfile status           # engine, peers, drive, settings
swarmfile status --changes  # staged files + conflicts
```

`storage`

Storage accounting for the mounted branch: the live file tree's logical bytes (including local-only rows such as an imported dependency layer), this machine's local block cache (bytes, block count, pinned count, cache ceiling), and their ratio. The ratio is `null` when this machine has no local blocks yet. **This is not the repository's billed hub storage** — the local cache only holds what this engine has read or written, so a cold mount can show a small cache against a large tree, and a warm one can show a high ratio from versions it has seen.

```
swarmfile storage
swarmfile storage --json
```

`events`

The host-call journal: one JSON line per CLI control call, written beside `control.json` (`trace.jsonl`, rotated at 1 MiB). Each line carries `atMs`, `method`, the path with its query string removed, `status`, `ms`, and an optional `actor` from `SWARMFILE_ACTOR` in the calling process's environment. No body, header or bearer is recorded. It covers buffered CLI calls and the SSE stream's open (its lifetime is not recorded); it does not cover the tray's or the web app's requests, and not engine-internal work.

```
swarmfile events                # last 50 calls, human lines
swarmfile events --limit 100    # a larger window
swarmfile events --json         # raw JSONL
swarmfile events --follow       # keep printing new calls
```

`host state`

One JSON object for an agent host, so it does not need three calls: `{schemaVersion, engine, mounts, storage}` where `engine` is the `status --json` payload and `mounts` is the `mounts list` array. The same contract is documented in the repository's host-integration design note; this is a CLI/JSON surface, not an SDK.

```
swarmfile host state
```

`stats`

Where this machine's read bytes actually came from — per file and in total: **this computer**, **a colleague's computer**, or **the cloud** — plus first-read timings. The engine counts each block **once per file per counter window**, at the moment that file is first served it (a cache hit is "this computer"; a LAN/WAN peer transfer is "colleague"; a fetch from cloud storage is "cloud"), including read-ahead, so the cloud and colleague figures are the bytes really downloaded and re-reading or scrubbing a timeline never inflates them.

Counters are per repository and per engine process; `--reset` starts a fresh window (use it between takes so each recording's numbers stand alone — it zeroes both this repository's breakdown and the engine's process-wide bandwidth counters), and the tray's **Data sources** page shows the same counters live with its own reset button.

`--json` prints the full evidence report — engine version, machine, OS/arch, the mount path/branch, file sizes, per-file source split, read counts and first-read latency — one object you can paste into an evidence log or a journalist can reproduce against. `--csv` prints the same as spreadsheet rows (a header, a summary row and one row per file, each carrying date/version/machine/repository) for the evidence log.

```
swarmfile stats             # human table
swarmfile stats --json      # the evidence report
swarmfile stats --csv       # spreadsheet rows, one take per machine
swarmfile stats --reset     # zero the counters after printing
```

`usage`

Estimated token spend for coding-agent sessions. It has two modes:

- **Org ledger** (the default): the organization's collected reports, opt-in, read through the engine. This is what the web dashboard's **Usage → Agent spend** card shows. Collection is off by default; an owner or admin enables it, and a member's engine then posts token counts (never content) at most every five minutes.
- `--local` : a scan of this machine's harness transcripts. It works with the engine stopped, reads only the local session log, and uploads nothing. From each line it uses the message id, model, timestamp and token counts; the line's content is discarded immediately and is never retained, logged, printed or sent. The transcript's own path is kept locally (and never shown) so a later scan can find it.

Both are estimates against a versioned price table — the 5-minute and 1-hour cache-write tiers are priced separately — never an invoice, and a model the table does not know is reported `unpriced` with its tokens kept. One row is one harness session: a wrapper workspace and a later direct resume of the same session do not double-count.

The org ledger also reports today's spend against the owner-set daily budget (UTC day; a new agent workspace is refused `agent_budget_reached`, exit 2, at the budget; the text output warns at 80%). For sessions with checkpointed changes it reports landed / in-review / exploration / unknown cost: landed is the hub's default-branch commit DAG, in-review is an open merge request, and a session with no checkpoint is unknown.

The split covers the newest 10 repositories, 200 ids per repository and the newest 200 sessions, and says when it is capped. A plain member sees their own attribution and their spend, not the org budget; a member-keyed commit row keeps two members' identical harness session ids apart.

Subcommand	Flags	Description
`agents`	`--local`, `--session <id>`, `--transcript <path>`, `--harness <auto|claude-code|codex|opencode|cursor|copilot>`, `--since <when>`	Per-session, per-model tokens and estimated cost. Without `--local`, read the org ledger (your own sessions as a member; the org's as an owner/admin; `--session` / `--transcript` are local-scan options). The org text shows per-session totals and a by-model cost line; its `--json` carries the per-model token counts and cache-write tiers. A member's view is their own attribution. With `--local`, scan every agent workspace this machine recorded, one session, or a transcript file. `--since` takes a duration (`7d`, `12h`) or a timestamp; a local session with no recorded time is always shown. `--harness` selects the local transcript format (default `auto`; Cursor and Copilot report `unreported`). A workspace whose harness left no transcript is `unreported`; Cursor and Copilot are `unreported` even when a path is recorded because they have no local token source; another harness's transcript that is gone is an `error` and the path is not printed. `--json` emits a stable object: the org ledger adds `local: false` and `mode: "org"` to the summary fields (including `priceVersions`, `ingestTruncated` and per-row `truncated` markers); a refusal in org mode is a JSON object too; `--local` emits `local: true` with `sessions`, `totals`, `priceVersion`, `pricedAt` and `caveats`. Per-model token counts include the 1-hour cache-write subset, and a transcript with no usable usage is flagged `usageMissing`.

```
swarmfile usage agents
swarmfile usage agents --since 7d --json
swarmfile usage agents --local
swarmfile usage agents --local --session claude-abc123 --json
swarmfile usage agents --local --transcript ~/.claude/repositories/-work-app/claude-abc123.jsonl
```

The price table's date and version are printed with the total, and an estimate below one cent is shown as `<$0.01` rather than a misleading `$0.00`.

`demo reset`

`swarmfile demo reset [--repository <id>] [--force] [--yes]` — start a demo take cold on **this machine**: empty the repository's local cache (the same whole-repository "Free up space" sweep, including "available offline" marks) and zero the `stats` counters — this repository's window and the process-wide bandwidth counters, so every surface the recording shows starts at zero. Run it on every machine a recording uses; a second take that quietly read from the cache is the failure this exists to prevent.

Files with saves that have not committed yet are **refused** (`409`) — their bytes may be the only copy — unless `-f` / `--force`, which proceeds while still keeping them. Cached content re-downloads on demand and nothing is deleted at the hub. The confirmation prompt is skipped with `-y` / `--yes`.

```
swarmfile demo reset --yes
swarmfile demo reset --repository prj_… --force
```

`demo offline`

`swarmfile demo offline on [--include-lan] [--for <duration>]`, `swarmfile demo offline off`, `swarmfile demo offline status [--json]` — cut **this machine's** network for real, on cue, for a recording of "the internet drops mid-work". It installs host firewall rules (macOS pf, Linux nftables or iptables, Windows Firewall), so the engine sees exactly what a dead office uplink looks like; it is not a simulation, and it works with no engine running.

- **The office LAN stays up by default.** Everything is blocked except this machine itself, private addresses (10/8, 172.16/12, 192.168/16), link-local, IPv6 unique-local and multicast — so office computers keep serving each other while the cloud is out of reach. `--include-lan` blocks the LAN too (everything but this machine); a remote shell to it drops.
- **Existing connections are cut too**, including the engine's long-lived connection to the hub — not just new ones.
- **It always switches itself back off.** `on` arms an automatic `off` after 30 minutes (`--for 90s`, `--for 10m`, `--for 1h30m`; at most 24h) that survives the command, the terminal, and on Windows a restart. `off` ends it sooner and cancels the timer; running it when nothing is on is harmless.
- **Needs administrator rights** for `on` and `off`: on macOS and Linux run it with `sudo`; on Windows, from an elevated terminal. Without them it exits `2` and prints the exact command to run. `status` needs none; `--json` reports `offline`, `scope`, `lanStaysUp`, `since`, `restoreAt` and `remainingSecs` for scripts.

```
sudo swarmfile demo offline on --for 10m   # internet off, office LAN up
swarmfile demo offline status
sudo swarmfile demo offline off            # back online
```

`op`

Some commands do work proportional to your data — `merge`, `commit`, `checkout`, `revert`, `materialize`, `branch create` / `archive`, `mr merge`, trash `restore` / `purge`, `repository delete`, `ignore-clean`, `hydrate free-space`, `undo` / `redo` (except `--dry-run`), and opening or closing a mount. They run as **operations** in the engine: the command waits and prints the result as usual, however long that takes, but the work no longer depends on the terminal staying open.

- **Ctrl-C detaches** instead of stopping anything: the command exits `3` and prints the operation's id, and the work carries on.
- `--no-wait` returns the id at once (exit `3`).
- An operation's outcome is kept, so you can collect it later — even after the engine restarts. One still running when the engine stopped is reported as interrupted.
- When an operation follows a hub background job — an over-cap folder delete/restore/purge or `repository delete` (see Trash, History & Rollback) — `op status` also shows its `done` / `total` progress while it runs.

Subcommand	Description
`op list`	Recent operations, newest first.
`op status <id>`	One operation's state and, once finished, its result. Exits `0` succeeded, `1` failed, `2` canceled, `3` still running.
`op wait <id>`	Wait for it to finish and print its result; exits the way the original command would have.
`op cancel <id>`	Stop an operation that can safely stop part-way (`materialize`, `hydrate free-space`, `git backfill-oids`, `git index`, and `checkout` / `revert` while still staging). A cancel that arrives once the staged apply has begun is declined (`too_late`) and the operation ends with the commit's real outcome. Commits, merges and deletes complete or fail on their own and refuse (`not_cancellable`).

```
swarmfile --no-wait merge --from feature-x   # prints the operation id, exit 3
swarmfile op wait 1b2c…                      # the merge's own result, when it lands
```

`mounts`

Manage this engine process's concurrent mount points. One engine can hold several full mounts open at once — of the same repository, or (same org only) a genuinely different one via `--repository` — each with its own changelists, metadata, and lock manager — for running multiple isolated mounts without paying git's one-worktree-per-branch tax (the motivating case: several AI coding agents, each working against the same repository without stepping on each other's saves).

Most installs only ever run the one boot mount; this is for when you deliberately want more than one. Multiple live mounts are supported on every platform, with a bounded number of FUSE-T mounts per machine (18; a FUSE-T helper-port limit, not an OS limit): an `open` fails with `fuse_t_helper_ports_exhausted` once that limit is reached — close a mount or quit the app holding a slot.

A boot mount that never attached can retry in place (`mounts retry`); one that attached earlier needs a restart, and `swarmfile doctor` names the holders. For the headless, per-agent recipe (API key, labeled mounts, shared cache), see Coding Agents.

Subcommand	Flags	Description
`list`	`--all`	Every mount this engine currently has open, `(current)` marking the one id-less commands (and any command run without `--mount`) target. `--all` adds each mount's workspace branch kind/expiry (ephemeral workspaces); it is not a branch inventory.
`retry`	—	Re-attempt a **boot** mount that failed with a coded FUSE-T exhaustion/probe error **before it ever attached**, in this engine process (the exhausted-slot remedy): reaps leaked FUSE-T helpers and retries without restarting the engine. A drive that attached once, a plain key/repository/config refusal and runtime mounts are not retried (restart an attached boot mount; re-open a failed runtime mount).
`new [mount_point]`	`--label <text>`, `--from <REF>`, `--repository <id>`, `--policy-root <DIR>`, `--ttl-days <1-30>`, `--set-current`, `--exclusive`, `--conflict-mode <soft|hard>`, `--wait`	Create an exact `agent/<label>-<6 hex>` ephemeral branch and open a mount pinned to it in one step. `--ttl-days` overrides the hub's configured workspace TTL (otherwise 7 days); the hub archives an expired workspace on its daily sweep. Omitted `mount_point` = `auto`. `--policy-root` reads the policy from that directory instead of the current mount (the MCP server passes its operator-owned root so a prompt-injected agent cannot weaken `paths_deny` or the checkpoint cadence). `--repository` targets another repository (same org); with `--repository`, `--from` accepts `branch:`, `tag:`, `seq:`, `changeset:`, `hash:` or a bare branch name. The branch is hidden from the default branch picker (`branch list --all` shows it) and is **not** an isolation boundary — the workspace shares the repository's ACLs, caches and P2P visibility. A mount failure archives the just-created branch only when the mount list proves no mount holds it; an unreadable list or an actual mount leaves it in place. If the engine reports the branch was created in a different repository, the command refuses before opening a mismatched mount.
`promote <id> <new-name>`	`-f, --force`	Close an ephemeral workspace mount and rename its `agent/*` branch to a named branch (stable branch id; TTL cleared). Refused if the mount has open changelists unless `--force`; a non-workspace branch is a no-op (exit 2).
`open <mount_point>`	`--label <text>`, `--set-current`, `--branch <name>`, `--repository <id>`, `--exclusive`, `--conflict-mode <soft|hard>`, `--wait`	Open a new mount at a filesystem path, a drive letter (Windows: `P:`), or `auto` — the next free drive letter on Windows, or a new folder beside the boot mount on macOS and Linux, named after the label or branch. On Windows a folder must be empty or not exist yet (Windows creates the mount folder itself); a folder with anything in it is refused with `mount_point_not_empty`, and a drive letter something else holds with `drive_letter_in_use`, naming what holds it. The command returns once the drive has attached, or with the reason it couldn't (`mount_failed`, `fuse_t_helper_ports_exhausted` when FUSE-T's helper-port range is full, or `fuse_t_helper_port_probe_failed` when it could not probe for a free port) — a mount that fails to attach is not left registered. It is a full, symmetric mount, equally capable to the boot mount, not a lightweight second view of it. Does **not** become current unless `--set-current` is given, so opening one never retargets a command already running elsewhere against the existing mount. `--branch` is optional; omit it to mount the repository's currently-active branch — but a mount opened without `--branch` shares the engine's boot branch context and can't be switched independently (`mounts branch` refuses it with `shared_branch_context`). Naming a *different* branch opens a divergent `(repository, branch)` scope: the mount works that branch while every other mount stays on its own, and it can be switched later with `mounts branch`. `--repository <id>` opens a mount on a DIFFERENT repository than this engine's active one (same org only); it combines with `--branch`, and a cross-repository mount pinned to a branch gets that branch's own live-update feed. `--exclusive` opts the mount into **exclusive-write mode**: for as long as a file is open for writing through it, a *sibling* mount opened with the same flag on the same repository+branch has its write-open refused (the app sees `EAGAIN`, the same refusal another machine's lock produces) — a rename, delete, or atomic-save replace of that file is refused too — instead of both editing the file and letting whichever closes last win. A mount that does not set the flag neither takes nor honors claims, so existing multi-mount setups are unchanged; open every agent mount with it when several agents share one tree. The flag is fixed for the mount's lifetime and shown by `mounts list` / `mounts status` as `exclusive`. `--conflict-mode` picks how a concurrent-write refusal (the hub rejecting a save built on a stale version) reaches the writer: `soft` (the default) accepts the save locally and surfaces the conflict asynchronously — the tray, the file reading read-only until resolved, and a `(conflict-…)` copy on the save lanes; `hard` makes the triggering `fsync` (Linux also the close-time `flush`; Windows `FlushFileBuffers` — a macOS `close()` cannot carry the error by design) wait for the hub's answer and fail it instead, with `ESTALE` (Windows: "the file was externally altered") so an automated writer's loop learns its save didn't become truth and re-reads. Omitted = the repository's setting, else soft. A mode pinned per drive is shown by `mounts list` and `mounts status` (`conflict:soft` / `conflict:hard`).
`close <id>`	`-f, --force`, `--archive`	Close a mount. Refused (exit 2, `changelists_open`) if it has open changelists, unless `--force` is given — never auto-cancels staged work. Refused (exit 2, `is_default_mount`) for the boot/default mount specifically: closing it shuts down the whole engine, not just one mount, so it isn't reachable through this command at all — quit the engine itself (the Desktop App's Quit, or stopping the `swarmfile-engine` process) if that's what you actually want. `--archive` archives the mount's branch afterwards when it is an exact `agent/<label>-<6 hex>` workspace, using the mount's own repository; any other branch is left in place and exits 2. A branch still used by another registered mount on this engine is not archived (exit 2), and an archive failure exits 1, so a script can tell the branch is still present.
`current <id>`	—	Switch which mount id-less commands target, from then on.
`branch <id> [branch]`	`--wait`	Switch one mount's branch independently of every other mount (omitted = the repository's default branch). Only works on a mount that was opened with its own `--branch`; a mount sharing the engine's boot branch context is refused with `shared_branch_context`. `--wait` blocks until the (asynchronous) switch lands or fails.
`status <id>`	—	One mount's own detail — the same fields `list` shows, for just this id, without scanning the whole list.
`scratch-dir <id>`	—	That mount's repository's machine-local scratch/dependency-cache directory, plus whatever tool-cache redirects its `.swarmfile/cache.yml` declares (see `scratch` below). Creates the directory — and every declared subdirectory — if it doesn't already exist.

> **Anonymous public mount.** `mounts open <path> --public <org>/<repository> [--hub <url>]` opens the same anonymous read-only public snapshot as `public-mount` instead of a signed-in repository mount. It runs in the foreground until Ctrl-C; `--repository` is refused (the flags conflict), while `--label`, `--set-current`, `--exclusive`, `--sparse`, `--exclude` and `--conflict-mode` do not apply.

> `mounts new --wait` / `mounts open --wait` block until the drive is attached (and, with the global `--timings`, print the create/open/attach phases). Without `--wait` the command returns as soon as the engine accepted the open, which may be `mountState: pending`.

> The global `--mount <id>` flag (see "Global flags" above) is how every OTHER subcommand targets a specific mount instead of `current` — e.g. `swarmfile --mount mount-abc123 status` runs `status` against that mount regardless of which one is current. It has no effect on `mounts` subcommands themselves, which already take the id as their own argument, nor on `workspace switch` (below), which retargets the whole engine's active workspace; combining the flag with either prints a warning and is otherwise a no-op for the flag.

```
swarmfile mounts new --label agent-3                        # agent/agent-3-<suffix>, auto mount point
swarmfile mounts new ~/agent-3 --label "Agent 3" --from feature-x --exclusive
swarmfile mounts new --label "Agent 3" --repository proj_abc123  # another repository's own default branch
swarmfile mounts open ~/mount-2 --label "Agent 2" --set-current
swarmfile mounts open auto --label agent-3 --branch feature-x   # Windows: next free drive letter
swarmfile mounts open ~/agent-3 --exclusive                     # exclusive-write mode for agent fleets
swarmfile mounts open ~/agent-4 --exclusive --conflict-mode hard  # automated writer: a stale save fails ESTALE
swarmfile mounts open ~/src-only --sparse "src/**"              # a sparse mount — only src/ is visible
swarmfile mounts open ~/no-vendor --exclude "vendor/**"         # everything except vendor/
swarmfile mounts open ~/other-repository --repository proj_abc123   # different repository, same org
swarmfile mounts list
swarmfile mounts status mount-abc123
swarmfile mounts current mount-abc123
swarmfile --mount mount-abc123 status       # target it without switching current
swarmfile mounts close mount-abc123 --archive
```

Sparse mount. `mounts open --sparse <glob>` (repeatable) exposes only the paths matching those include globs — the same dialect as `materialize --sparse` (a glob matching a directory includes its subtree, a leading `!` excludes an earlier match, a bare name matches at any depth).

`--exclude <glob>` is sugar for the `!` negations; with no `--sparse` it means "everything except".

An excluded path is invisible at that drive: no listing entry, an open fails as if it weren't there, and a write-create/mkdir/rename onto it is refused. Empty = the whole branch, and the scope is per mount (a sibling mount is unaffected); a scoped mount's offline (`hydrate` /Pack & Go) set matches the scope.

`public-mount`

Mount a **public** repository read-only with no account: an anonymous snapshot of the repository's derived git head. It resolves the refs through the anonymous public clone gate (the repository must be public, git-enabled, and have anonymous cloning on), fetches tree pages and file blocks on demand through the public PoW ticket, and runs in the foreground until Ctrl-C.

```
swarmfile public-mount swarmfile/swarmfile-lfs ~/public
swarmfile public-mount swarmfile://hub.swarmfile.com/swarmfile/swarmfile-lfs ~/public --branch main
```

Every write returns `EROFS`; there is no lock, queue, comment, issue, MR or share surface. No API key or cookie is sent, and the engine does not need to be running.

The snapshot does not appear in `mounts list` and does not touch a signed-in repository cache. `--branch` selects another derived head; omitting it uses the repository's default branch.

`--hub <url>` overrides the configured hub; a `swarmfile://<host>/<org>/<repository>` source defaults to `https://<host>`. macOS (FUSE-T) and Linux (FUSE) are supported; Windows is not implemented yet.

If a repository turns anonymous cloning off, a fresh mount within the 60 s boundary-verdict window may still attach, but its first content read is refused; outside that window it fails with the same uniform refusal as the public clone surface (the git refs are served live, not from the edge cache).

A file whose stored content is an LFS **pointer blob** appears as that pointer text; a native large file streams its content through the mount (LFS objects are never materialized). Use the public clone/LFS path for the actual bytes, or keep LFS off the default branch of a repository you intend to show this way.

`workspace`

Show or change the org/repository this engine is scoped to — the CLI equivalent of the Desktop App's org and repository pickers. A same-org repository switch usually takes the hot path (no restart), and an org switch does too while the mount is live; switching to a repository the running mount can't be re-scoped to, or switching with no live mount to work with, drains and restarts the engine, so the command may briefly drop its connection.

A target repository whose key hasn't been released yet doesn't force that restart — the switch waits for the grant and completes in place.

A same-org switch *with creates still queued* can also take that restart fallback — queued creates survive it and land in the repository they were created under. Refused (exit 2) while a changelist is open, or while another switch is already running.

Subcommand	Flags	Description
`status`	—	The current org and active repository for this mount.
`list`	—	Every org this account can reach, plus every repository in the current org, with `(active)` marking the current pair. Use these ids with `switch`.
`switch`	`--org <id>`, `--repository <id>`, `-y` / `--yes` (`--force`), `--park`	Move the engine to that org, and optionally that repository in it. `--org` is required; omitting `--repository` clears the stored repository so the engine adopts the target org's default. A switch is refused (exit 2) while a changelist is open — `--park` sets the mount's open changelists aside first so the switch can proceed; `--yes` skips the confirmation prompt (which names staged files and queued/failed writes).
`deps gc`	`--repository <path>`, `--cache-dir <dir>`, `--kind <share-deps|local>`, `--older-than-days <n>`, `--keep <key>`, `--apply`, `-f` / `--force`	List the share-deps and machine-local dependency layers and their age; a dry run by default. `--apply` removes layers unused for at least `--older-than-days` (default 14) — a live repository marker is still kept unless `--force`; `--keep` pins a key. `--cache-dir` reads the machine-local layers with the engine stopped.
`warm`	`--share-deps`, `--local`, `-- <command…>`	Run an install/build/test through the repository's shared dependency caches (the `swarmfile run` redirects) and print how long it took. Run it **from the mount** (or use `hooks run`, which sets the cwd); the first workspace on a machine warms the cache and later workspaces reuse it. `--share-deps` builds a hub-traveling installed tree; `--local` builds a machine-local cache layer (Unix only; conflicts with `--share-deps`).

```
swarmfile workspace status
swarmfile workspace list
swarmfile workspace switch --org org_abc --repository proj_9f2a
swarmfile workspace switch --org org_abc        # adopt that org's default repository
swarmfile workspace warm -- pnpm install         # warm the shared cache
swarmfile workspace warm --share-deps -- pnpm install  # also share the installed tree
swarmfile workspace deps gc                             # list the layers (dry run)
swarmfile workspace deps gc --apply                     # remove unused layers
swarmfile workspace deps gc --kind local --older-than-days 30 --apply
```

`--share-deps` runs an **install** command inside `.swarmfile/deps/<OS-arch-Node-ABI>-<libc>-<lockfile-hash>/tree`, stages the installed `node_modules` copy as `tree/.packages.partial` and renames it to `tree/packages` with a completion marker (an interrupted copy is rebuilt, not half-linked), and leaves a relative `node_modules` symlink to it. Use it for `npm/pnpm/yarn install`, not build/test, which need the workspace's source files; run it from the repository root.

`--local` builds/reuses a machine-local installed tree under the engine cache (never the hub) and imports it as a real local-only `node_modules` directory (no absolute link; a new workspace imports it in ~12–15 s, measured 14.6 s first / 12.6 s reuse, and the same branch reuses it); it preserves executable bits, so `node_modules/.bin` tools run, refuses a strict E2E/static mount (its blocks are plaintext), and rolls back a partial import. It is Unix-only for now, and conflicts with `--share-deps`.

A branch forked from the warming branch (for example `swarmfile mounts new --from <branch>`) inherits the tree and can run tests without reinstalling. The tree is ordinary repository content: it syncs to the hub and counts against repository storage.

`.npmrc` is never copied into the shared tree (registry credentials stay in place). Native modules are only portable across the same OS/CPU/Node ABI; nested `node_modules` from pnpm and old trees need manual cleanup. On a Windows **WinFsp mount** the host cannot create the `node_modules` symlink/junction, so `--share-deps` currently refuses at the link step (the install/copy still exists); use the hub-backed tree on a Unix mount. A junction works only on an ordinary NTFS directory.

Old layers are listed and removed by the dependency-layer commands below. The import is bounded at 20,000 entries and 512 MiB per file.

`--local` is the machine-local variant: it runs the install in `<engine-cache>/layers/<OS-arch-Node-ABI>-<libc>-<manifest+command-hash>/` (never the hub), then **imports** that tree into the workspace as ordinary `local_layer` metadata entries and local block-store content. `node_modules` is a real local-only directory (no absolute symlink, so the cache and its siblings are not reachable through the mount and a mount sandbox cannot walk into them).

The layer is visible/readable and is not synced; a fresh workspace attaches it by re-running `workspace warm --local -- npm install` (the cached tree is imported/reused; no install; the CLI says the command was not run).

It is install-only, must run from the **mount root**, supports single-package npm/yarn/pnpm installs (pnpm's relative `.pnpm` store links are imported), and still refuses workspace/monorepo installs; use a plain install in the checkout (a `--share-deps` tree is single-package too, so it does not carry workspace package `node_modules`).

It merges the repository `.npmrc` with the user config (relative `cafile` / `certfile` / `keyfile` values resolve against the file that declared them; the merged 0600 temp is removed after the install, and one abandoned by a killed process is swept after a day). It is Unix-only for now; old cache layers are listed and removed by the dependency-layer commands below (a layer whose recorded repository still exists is kept; `--keep` / `--force` control the exceptions).

The CLI prints the imported key; the share-deps GC refuses any share-deps `tree` rather than partially deleting it.

> `workspace switch` changes the whole engine's active workspace, not one mount, so the global `--mount` flag has no effect on it (the CLI prints a note if you combine them).

`workspace deps`

Inspect and reclaim this machine's dependency layers. Engine-free: it reads the repository tree and the engine cache, so it works while the engine is stopped and never resolves a session.

Subcommand	Flags	Description
`gc`	`--repository <DIR>`, `--cache-dir <DIR>`, `--kind <share-deps|local>`, `--keep <KEY>` (repeatable), `--older-than-days <N>` (default 14), `--apply`, `-f, --force`	List `.swarmfile/deps` and `<cache>/layers` layers and remove old, unused ones. A dry run unless `--apply`. Keeps the current `node_modules` link, a local layer whose recorded repository still exists, `--keep`, a live install, and any share-deps layer with a `tree` (a normal/fork share layer is reported BLOCKED: its tree is synced repository content and must be removed through the mount/hub, not by a host-side GC); `--force` also reclaims an old live local layer. Another branch/worktree's use is not auto-detected, so pass `--keep`. A dead install marker does not block. A symlinked `.swarmfile` / `deps` /cache `layers` root or layer is refused. For `--kind local`, the GC removes the cache-layer source only; the imported local-only metadata/blocks stay pinned until the repository detaches or the cache is resynced. Exits 2 rather than applying without a current link or `--keep`; re-checks before every removal.
`materialize <key> <checkout>`	`--cache-dir <DIR>`, `-f, --force`	Copy an installed local layer into a plain (unmounted) checkout. The copy is staged and renamed (no partial `node_modules`), executable bits and relative symlinks are preserved, an existing `node_modules` is refused without `--force`, escaping/absolute links are refused, and the checkout is recorded as the layer's repository.
`export <key> <archive.tar.gz>`	`--cache-dir <DIR>`	Package a local layer for another machine. The archive includes a format/key manifest and the `node_modules` tree (symlinks and exec bits preserved); the key must match this machine's OS/arch/Node-ABI/libc prefix. It is structurally validated with 20k-entry/512 MiB-per-file/2 GiB-total caps; an archive inside the source is refused.
`import <archive.tar.gz>`	`--cache-dir <DIR>`	Validate and seed a layer archive into this machine's cache. Paths must stay under `node_modules`, symlinks must stay inside, the format/platform key and the 20k/512 MiB/2 GiB caps are checked; a healthy existing key is kept, a corrupt one is replaced, and a live/in-progress one is refused. Then run `materialize`.

```
swarmfile workspace deps gc                    # dry-run: list layers and what is removable
swarmfile workspace deps gc --apply            # remove old, unused layers
swarmfile workspace deps gc --kind local --keep <key> --apply  # reclaim a local layer
swarmfile workspace deps materialize <key> /path/to/checkout    # plain clone / CI workspace
swarmfile workspace deps export <key> layer.tar.gz              # move a layer to another machine
swarmfile workspace deps import layer.tar.gz                    # seed the cache there
```

`gc` is a dry run by default. The current `node_modules` link, a local layer whose recorded repository still exists, a `--keep` key, a layer with an install in progress, and any share-deps layer with a `tree` are never removed; `--older-than-days` is the retention threshold and `--force` also reclaims an old live local layer. It exits 2 rather than applying without a current link or `--keep`, and re-checks the filesystem before every removal.

`materialize` copies an installed local layer into a plain checkout (a regular `git clone` or a CI workspace with no mount). The copy is staged inside the checkout and renamed into place, so an interrupted run never leaves a partial `node_modules`; executable bits and relative symlinks are preserved, an existing `node_modules` is refused without `--force`, and the layer records the checkout as its repository so `gc` keeps it.

It reads a layer already in **this machine's** engine cache. For another machine, `export` the layer, copy the archive, `import` it there (the OS/arch/Node-ABI/libc prefix must match), then `materialize`; the archive validation rejects unsafe paths/symlinks and an existing key is kept rather than replaced.

`conflicts`

Lists files that are blocked because the same file was changed in two places and the two versions haven't been reconciled yet, and resolves them.

While a file is in that state, writes to it are refused, and an application sees a plain I/O error — a filesystem has no way to explain a cross-machine change. This command is where the reason lives.

Subcommand	Flags	Description
`list` (default — bare `conflicts` is the same)	`--state active|resolved|all`	Files blocked by an unresolved conflict, by path. `--state resolved` lists recent history (including `resolution: local` rows written by `.swarmfile/conflict-policy` auto-wins); `all` shows both. A row on another branch (a branch-merge conflict is recorded under the merge TARGET's entry id) prints the branch name, or `(not on this mount's branch)` when the hub couldn't name it.
`show <path>`	`--state open|active|resolved|abandoned|all`	One file's conflict history (all states by default): derived lifecycle, origin and resolver. Read-only.
`resolve <path>`	`--winner local|remote|keep-both`	Accept one side non-interactively, without running any tool.
`resolve <path>`	`--tool [name]`	Resolve via an external mergetool speaking git's own `mergetool` protocol. Mutually exclusive with `--winner` and `--auto`.
`resolve <path>`	`--auto`	Run the engine's three-way diff3 auto-merge and accept the result if it's clean — no external tool, no `gitconfig` needed. Exits **2** when a real conflict remains (a text collision, or a binary).
`resolve --all [--auto | --winners]`	—	Attempt every conflicted file on this mount's branch, printing one summary at the end. Requires `--auto` (engine merge) or `--winner <local|remote|keep-both>` (accept that side everywhere). Rows on other branches are skipped and counted unless `--include-other-branches` is given — pass it when landing a merge (the conflict is recorded under the target branch), or use `merge --diff3`. Exits **2** if any file still needs a human (auto only), **1** if any request failed.
`resolve --all --include-other-branches`	—	With `--all`, also act on conflicts the hub reports for other branches.

```
$ swarmfile conflicts
1 file(s) blocked by an unresolved conflict:

  /Repositories/plan.dwg

Writes to these files are refused until the conflict is resolved.
A namespace observation is still addressed by its occupant path; add
`--conflict-id <id> --winner keep-both|abandon` (`swarmfile conflicts show`
prints the id) to name the namespace row instead of the content tracker.
Resolve a path with `swarmfile conflicts resolve <path> --auto`, `--winner
local|remote|keep-both`, or `--tool`, in the Desktop App, or in the web dashboard.
```

`--winner` picks a side directly — same three outcomes as the Desktop App/web picker (keep mine, keep theirs, keep both), just from the CLI:

```
swarmfile conflicts resolve /Repositories/plan.dwg --winner local
```

For a conflict recorded by `merge --record-conflicts`, `local` is the incoming branch's version and `remote` is the version already on the branch being merged into — see Branches & Merging.

When `--winner remote` keeps a version different from yours, your edit is preserved in the file's history (it shows in `blame` and the History views) and the command prints a note saying so; `swarmfile rollback --at <id>` can bring it back.

`--auto` runs the engine's own three-way diff3 merge and submits it when it comes back clean. It needs nothing installed and no `gitconfig`, which makes it the resolution path to reach for from a pre-flight or CI script:

```
swarmfile conflicts resolve /Repositories/plan.dwg --auto
```

It exits **2** (not 1) when auto-merge can't produce a clean result — a genuine three-way text collision, or a binary / non-diff3-eligible file — so a script can tell "a human must choose" apart from "the engine was unreachable" and fall back to `--winner` or `--tool`.

Add `--all` to sweep every conflicted file in the repository in one pass and clear the text ones automatically, leaving only the files that genuinely need a person:

```
swarmfile conflicts resolve --all --auto
```

`--all` also accepts `--winner`, to accept one side across every conflicted file in one pass (useful when a whole branch should keep yours, or theirs):

```
swarmfile conflicts resolve /Repositories/plan.dwg --conflict-id <id> --winner keep-both  # namespace row
swarmfile conflicts resolve --all --winner local
```

`--tool [name]` resolves a whole-file conflict — including binary files the Desktop App/web picker's "try automatic merge" can't touch — via an external mergetool speaking git's own protocol: `$BASE` / `$LOCAL` / `$REMOTE` / `$MERGED` temp-file substitution, same as `git mergetool`. This is the one resolution path that's CLI-only; it needs a real terminal/GUI session to run the tool in, which only the foreground CLI process has.

Resolution order is: swarmfile's own `mergetool_cmd` config (see the config reference) if you've set one — it wins over gitconfig without even reading it — else your git config's `[merge] tool` + `[mergetool "<name>"] cmd`, so anyone with `git mergetool` already configured and no swarmfile-specific override gets it for free. Name one explicitly to bypass both:

```
swarmfile conflicts resolve /Repositories/plan.dwg --tool          # configured default
swarmfile conflicts resolve /Repositories/plan.dwg --tool kdiff3   # a specific tool
```

A nonzero exit from the tool aborts before anything uploads — nothing is resolved, and the CLI tells you where the scratch files (`$BASE` / `$LOCAL` / `$REMOTE` / `$MERGED`) are for a manual look or retry.

`throttle`

Bandwidth throttling — applies immediately, no restart.

Subcommand	Flags	Description
`status`	—	Whether throttling is on, the rate in effect right now, and why (office or off hours, and that window's percentage of the site link).
`enable`	`--download-mbps <N>`, `--upload-mbps <N>`, `--office-hours-start <HH:MM>`, `--office-hours-end <HH:MM>`, `--office-hours-wan-pct <0-100>`, `--off-hours-wan-pct <0-100>`	Turn throttling on. Omit a flag to leave that value at whatever it already resolves to.
`disable`	—	Turn throttling off.

The `--office-hours-*` / `--off-hours-*` flags shape a time-of-day schedule: WAN traffic is capped to a percentage of the site link during the office-hours window (`--office-hours-start` / `--office-hours-end`, `HH:MM` local time) and to a different percentage outside it, so background sync can back off while people are working and open up overnight. The window's start and end must be different times: to use one rate all day, set both percentages to the same value. These same values are file-settable as `office_hours_start` / `office_hours_end` / `office_hours_wan_pct` / `off_hours_wan_pct` (see Engine Config File).

```
swarmfile throttle enable --office-hours-start 08:00 --office-hours-end 18:00 \
  --office-hours-wan-pct 20 --off-hours-wan-pct 80
```

What the cap covers. Bulk background transfer: uploads, `hydrate`, and `fetch`. A read an application is actually waiting on — opening a file through the drive — is **not** throttled, so a cap set to protect the office link can't turn somebody's file open into a stall. The practical consequence is that a large `hydrate` or `fetch` is bounded by this setting: both are pre-fetches filling the cache ahead of use, not somebody sitting in front of a spinner.

`cache`

Local disk cache size cap — applies immediately, no restart. This is the block cache (the repository's own synced content); for machine-local package-manager/build caches shared across mounts, see `run` and `scratch` below — a different, unrelated cache.

Subcommand	Flags	Description
`status`	—	Current cache cap and usage.
`set`	`<gib: u64>`	Set the local disk cache cap, in GiB.
`public`	`--clear`, `--dry-run`, `-y, --yes`	Report the anonymous public-mount cache (`<cache>/public/<org>/<repository>`), or remove it with `--clear` (asks first; a repository with a live mount is skipped; `--dry-run` previews). Needs no engine.
`reclaim`	`--dry-run`, `-y, --yes`	Remove a leftover unscoped block-cache directory if one exists (its blocks re-download on demand); does nothing otherwise. `--dry-run` reports without deleting; otherwise it asks first (`--yes` skips).
`resync`	—	Re-fetch this repository's hub change feed from the beginning — a repair for a mount whose feed cursor advanced past a change it never applied (an entry missing on that machine alone while the hub serves it correctly). Applies are idempotent; local rows and cached bytes are kept, and the walk runs in the background with the engine retrying until it catches up.

`run`

Run a command with this repository's declared tool caches (see `.swarmfile/cache.yml`, under `scratch` below) redirected into environment variables — e.g. `swarmfile run -- pnpm install` points pnpm's store at a directory every mount of this repository, on this machine, shares, so a second mount (or a second AI agent's mount) doesn't pay to re-download what the first one already fetched. Put `--` before the command so its own flags aren't parsed as swarmfile's.

Always safe to prefix onto anything: with no `.swarmfile/cache.yml`, a repository that declares none, a config with a bad entry, or even an unreachable engine, `run` just warns on stderr and runs the command completely unmodified — it never refuses to run the command over a cache-resolution problem. The wrapped command's stdio and exit code are passed straight through. For wiring several agent mounts onto one shared cache, see Coding Agents.

```
swarmfile run -- pnpm install
swarmfile run -- cargo build --release
```

`--detach` starts the command in the background and returns at once with its id and log path. The record and log live under the engine cache (`processes/`), are machine-local, and are not repository content; the process is not checkpointed when it exits. It is for a dev server, a test watcher or a long build, not for work whose output must be committed:

```
swarmfile run --detach -- npm run dev
# started 1761…-8123-… (pid 4812)
# log:  …/processes/1761…-8123-….log
swarmfile processes list
swarmfile processes log <id> --tail 100
swarmfile processes stop <id>
```

`processes`

Manage commands started with `run --detach`. These commands read local records/logs and work while the engine is stopped or restarting.

- `list` shows running/exited state, pid, cwd and argv (newest first)
- `log <id> [--previous]` prints the tail of the captured log (the log rotates at 10 MiB and keeps one previous file; `--previous` reads `<log>.1`)
- `stop <id>` verifies and kills the command and its process group
- `stop <id> --force` (short `-f`) signals the recorded group after its leader has exited (a shell that left a child running) — use it only for a record you recognize; this recovery is Unix-only (Windows has leader-only group semantics), and a live leader whose start token could not be captured also requires `--force`; a live leader with a mismatched token is refused
- `prune [--older-than-days N]` removes stopped/exited records and their logs older than 7 days (running commands are kept)

Records survive the CLI exiting but not an engine-cache wipe, and nothing here is synced or committed. The log rotates at 10 MiB (one previous file kept as `<log>.1`, so a chatty server cannot fill the cache); `log` reads at most the last 512 KiB of the current file. A detached process is not owned by the engine after a restart — records/logs remain but the engine does not restore the command; `processes prune` cleans them up.

```
swarmfile processes stop <id> --force
swarmfile processes prune --older-than-days 14
```

Subcommand	Flags	Description
`log <id>`	`--previous`	Print the tail of the captured log; `--previous` reads the rotated `<log>.1` file.
`stop <id>`	`-f, --force`	Verify and kill the command and its process group; `-f` signals the recorded group after its leader has exited (Unix only; Windows is leader-only).

`scratch`

This machine's per-repository scratch/dependency caches — the machine-local, non-versioned directories `run` (above) redirects tool caches into. One directory per repository, shared by every mount of that repository on this machine regardless of branch, the same way a developer's real `~/.cargo/registry` isn't per-checkout either. The worked example (agent mounts sharing one cache, with `gc`) is in Coding Agents.

Subcommand	Flags	Description
`list`	—	Every repository's scratch cache on this machine: size on disk, how long since it was last used, and whether that repository is currently open in this engine.
`clear <name>`	`-f, --force`	Delete ONE repository's scratch cache outright, by the name `list` shows. Refuses if that repository is currently open, unless `--force`.
`gc`	`--older-than-days <N>` (default `30`), `--dry-run`, `-y, --yes`	Delete every scratch cache that's both closed and untouched for at least the given number of days. Reports what it removed — or, with `--dry-run`, what it *would* remove — and the total space reclaimed. Asks first (`--yes` skips; `--dry-run` never asks).
`init`	—	Write a starter `.swarmfile/cache.yml` into this mount's repository root. Refuses to overwrite an existing file.

```
swarmfile scratch init
swarmfile scratch list
swarmfile scratch clear <name> --force
swarmfile scratch gc --older-than-days 14 --dry-run
```

Nothing here is unrecoverable repository data: everything under a scratch cache is a re-buildable tool cache (a pnpm store, a cargo registry, ...), never repository content, so clearing one only costs a slower next build, not lost work.

`.swarmfile/cache.yml`

A versioned, repository-root config file — commit it alongside the repository, like `.swarmfileignore` — declaring which tool caches `run` should redirect:

```
caches:
  - name: pnpm
    env: PNPM_HOME
    subdir: pnpm
  - name: cargo-registry
    env: CARGO_HOME
    subdir: cargo
```

Each entry needs a `name` (used only in messages), the `env` variable the tool reads for its cache location, and a `subdir` (relative, no `..`) under the shared scratch directory. A single bad entry — an unsafe subdir, a name collision with an earlier entry, or one of a handful of env vars this refuses to redirect on principle (`PATH`, `HOME`, `LD_LIBRARY_PATH`, and similar) — is dropped with a warning; it never takes the other, otherwise-valid entries down with it. Run `swarmfile scratch init` if you'd rather start from a commented example than write one by hand.

> Only redirect READ-MOSTLY, CONTENT-ADDRESSED download caches here — a pnpm store, `~/.cargo/registry`, a pip wheel cache. **Not** build *output* directories (`target/`, an in-progress `node_modules` install): sharing one of those across concurrent mounts risks lock contention, not just staleness.

`changeset-idle`

Changeset session-minting idle timeout — how long the engine waits after your last write before sealing the open changeset and starting a fresh one on the next write. This groups a burst of related saves into one commit rather than one commit per save. Only applies on a single-repository mount. Distinct from `changesets list` (the commit history) — this is the settings command, not a log.

Subcommand	Flags	Description
`status`	—	This mount's current changeset idle timeout, in seconds.
`set`	`<idle_secs: u64>`	Set the idle timeout. `0` disables commit grouping (writes land ungrouped).

> `changeset-idle set` restarts the engine to take effect (the changeset manager is only constructed at startup). Persists to the config file's `changeset_idle_secs` — see Engine Config File.

`attr-cache`

How long (in seconds) the mount may trust cached file/directory attributes before revalidating — this controls how quickly a teammate's newly-saved file stops showing a stale directory listing. `0` means "leave the platform's own default alone" (not "disabled").

Subcommand	Flags	Description
`status`	—	This mount's current attribute-cache timeout, in seconds.
`set`	`<attr_cache_secs: u64>`	Set the timeout. `0` keeps the platform default.

> `attr-cache set` restarts the engine to take effect, and applies only to macOS (`fuse-t`) / Linux (`vfs`) builds — it has no effect on a Windows/`winfsp` mount. Persists to the config file's `attr_cache_secs`.

`hydrate`

Bulk-materialize a scope of this mount into the local cache, pinned against eviction until released — for a machine that needs to **read** that scope with no network, or that wants everything local before a job starts rather than fetching lazily during it. It does not reserve anything for writing: to also keep a scope editable with no hub, use `offline prepare`, which reserves the whole scope.

Subcommand	Flags	Description
`start`	`[path]`, `-y, --yes`, `--wait`, `--sparse <glob>`, `--exclude <glob>`	Download and pin every file under `path` — relative to the mount, or a full path inside it (default: the whole repository on the selected mount's branch — use the global `--mount <id>` to target another mount). Prints a size estimate and a warning if it exceeds your cache cap, and asks for confirmation, unless `--yes`. `--sparse` / `--exclude` (the same globs as `mounts open`) narrow it to the matching paths; on a sparse mount the mount's own scope already applies, and a request can't widen it. Polls progress until done, except in `--json` mode, which returns as soon as the job starts unless `--wait` is given.
`status`	—	Progress of the current (or most recent) hydrate job.
`cancel`	—	Cancel a running hydrate job. A no-op if nothing is running.
`release`	`[path]`	Unpin everything hydrated under `path` on the selected mount's branch, making it evictable again. With no path, unpin everything this machine holds for the repository — every branch, since the pin map is per machine and repository-wide.
`free-space`	`[path]`, `--dry-run`	Free up space: delete the downloaded copy of everything under `path` so it's cloud-only on this computer, and stop keeping it offline. Files stay listed and download again when opened. Files with changes that haven't synced yet are skipped. With no path it covers the whole repository and also clears older cached versions. `--dry-run` only reports what would be freed.

```
swarmfile hydrate start ./Repositories/shots/seqA --yes
swarmfile hydrate start ./src --sparse "src/**" --exclude "src/fixtures/**"
swarmfile hydrate status
swarmfile hydrate release ./Repositories/shots/seqA
swarmfile hydrate free-space ./Repositories/shots/seqA --dry-run
```

Pinned blocks are never evicted by the cache's normal LRU sweep, so a large hydrate can push disk usage past your configured `cache` cap — `hydrate start` warns about this up front rather than silently growing past it. Pins are permanent until you run `hydrate release`, not time-limited.

`sync`

Saves that aren't reaching the cloud on their own — the cloud keeps refusing them, or their upload gave up — and a way to give one up.

Subcommand	Flags	Description
`stuck`	—	List stuck saves, one per file: the file's entry id, name and path, why it's stuck (`commit_refused` or `upload_failed`) and since when. A new file refused at a protected repository's root is held, not lost: ask an owner to grant repository-wide write, or create a folder to put it in — the full reason sentence is in `swarmfile status` and the Desktop App's **Why did my save fail?**
`discard`	`<entryId>`, `-y, --yes`	Drop the file's stuck saves and put it back to its cloud version. **The unsynced change on this computer is lost**, so this asks first unless `--yes`.

```
swarmfile sync stuck
swarmfile sync discard 6f1c…e2 --yes
```

`uploads`

Every **live in-flight upload** on this mount — your own saves and other machines' — from the hub-synced pending pointers. This is the "what is still on its way" view that the Desktop App's **Uploading** badge summarizes, across everyone, not just this computer.

Flag	Description
`--wait`	Poll until nothing is uploading, instead of printing once. Useful before a checkout or a mirror in CI, where acting on a file that is still landing would read the old version.

```
swarmfile uploads
swarmfile uploads --wait   # block until every in-flight upload has landed
```

`swarmfile uploads retract <path>` clears one file's pending pointer when the machine that was saving it is gone for good — the case where no upload will ever finish and nobody should keep waiting on it. It needs **write** access to the file, reads the live pointer first, and clears against that pointer's own manifest: if the upload has re-announced since, the retract is a no-op rather than an interruption, so it can never yank a genuinely live upload out from under its writer.

```
swarmfile uploads retract ./Repositories/reel-04.r3d
```

A pointer nobody refreshes for 30 minutes stops blocking branch creates, merges from its branch, and copies even without this; retracting is for making that explicit and for the readers that wait on the pointer.

An upload that has **stalled** (a save this machine keeps failing to land) is a different list — see `sync stuck`.

`materialize`

Write a ref's tree to a plain directory — **no mount, no FUSE**, so it works on a CI box with no kernel driver. The ref resolves through the same vocabulary as `show` / `restore` (`N` / `#N`, `HEAD~N`, a branch, a tag, a changeset id, or a 64-hex commit hash; a prefix like `hash:` or `tag:` forces one kind), and it defaults to the selected mount's branch head.

A commit ref — a number, `HEAD` / `HEAD~N`, a tag, a hash, or another branch's name — writes **that commit's tree**, exactly as it was committed. Only the default (or the mount's own branch, e.g. `branch:main` on a main mount) writes the branch as the mount sees it right now, including saves that haven't been committed yet. A commit archived before it had a stored tree can't be materialized (`no_commit_tree`); pick a newer commit or a branch.

On a **protected** repository, a member's checkout of a historical ref is filtered by the permissions that were in force at that commit — the hub replays its ACL log at the commit's own time, so files they could read then are written even if they're restricted now, and files they couldn't read then are left out. If the hub has no ACL history for that commit it prints a note that the listing was instead filtered against *current* permissions — never silently. Owners and service/CI callers (see Runner) are never filtered.

The destination must be new or empty, or a directory this tool already wrote: it keeps a `.swarmfile-materialize.json` marker and updates in place, pruning only paths it wrote before — so files your job created between runs (build outputs, caches) survive, and it never overwrites or deletes a directory it doesn't own.

A 64-hex hash is the most reproducible pin (tags can be repointed, branches move). Executable files (see `chmod`) are written executable on macOS and Linux. Exits non-zero if any file failed to materialize.

`--sparse <glob>` (repeatable) checks out only the paths that match — for a huge repo where a job needs one subtree. A path is written when it matches a glob or a directory in its ancestry does; `*.md` matches by name at any depth, `src/**` anchors to the root, and a leading `!` excludes a path an earlier glob included (gitignore-style).

Everything else is neither fetched nor written, and a directory with no matching file isn't created. At most 64 globs, each at most 1024 characters, at least one non-`!`.

`--exclude <glob>` is sugar for the `!` negations (with no `--sparse`, "everything except"). Omit either for a full checkout. The `clone --checkout` form takes both flags.

```
swarmfile materialize v1.0 --out "$PWD/src"
swarmfile materialize hash:8f14e45fceea167a5a36dedd4bea2543ed10d0f8f9d8f9d8f9d8f9d8f9d8f9d8 --out ./checkout
swarmfile materialize HEAD~2 --out ./prev
swarmfile materialize hash:$COMMIT --out ./checkout --sparse "src/**" --sparse "*.toml"
```

See Runner (Headless CI) for the full external-CI recipe, including the marker's ownership rules.

`git status`

How ready this repository is to be served to git. Files at or above the git-LFS pointer threshold (default 1 MiB) would reach git as LFS pointers, and a pointer needs the file's SHA-256 object id. `git status` walks the selected mount's branch as the mount sees it and reports how many such files there are, how many still have no recorded id, and the first 50 of those with their sizes. It's the readiness check for turning on git access, which refuses while any counted file is missing an id.

It reports the access state first, before the readiness counts: **enabled and frozen** (with the conversion version and pointer threshold), **not enabled** (with the command that turns it on), or **not available** on an end-to-end-encrypted repository. When the hub can't be reached, that line is omitted rather than guessed at.

Flag	Meaning
`--threshold-bytes <N>`	Count files of at least `N` bytes instead of 1 MiB.

Files saved through the desktop app (or imported with `swarmfile-migrate`) on a managed or unencrypted repository get an id as they're saved. Files added through the web dashboard's uploader or produced by resolving a conflict get one too. Files with no recorded id show as missing until `swarmfile git backfill-oids` fills them in. An end-to-end encrypted repository never has ids — Swarmfile doesn't store a hash of E2E content — and says so; neither does content encrypted with a static key you supply yourself (`SWARMFILE_ENCRYPTION_KEY`).

It also reports how many commits on the branch are still **unconfirmed (unsigned)** — with how many of those are disputed and the oldest one's age — when there are any. A commit becomes servable to git as soon as it's derived, even before it's confirmed: a fetch returns the provisional hash and, from a second machine, confirms it. A teammate's app and an API key used in CI can confirm one the same way. A commit that stays unconfirmed for a long time is still worth a look; the warning never changes the exit code.

Exits `0` when every counted file has an id, `2` when some don't (or on an E2E repository, or while the hub doesn't yet know the repository's encryption tier). `--json` returns the counts and paths.

```
swarmfile git status
swarmfile git status --threshold-bytes 10485760 --json
```

`git backfill-oids`

Records the missing ids `git status` reports. For each file at or above the threshold that has no id, it reads the file through the desktop app's normal read path (downloading it if it isn't cached), computes its SHA-256, and sends the ids to Swarmfile. It works through the files in batches and prints progress as it goes. Repository owners and admins only; an end-to-end encrypted repository, or one encrypted with your own `SWARMFILE_ENCRYPTION_KEY`, never records ids and is refused.

Flag	Meaning
`--threshold-bytes <N>`	Count files of at least `N` bytes instead of 1 MiB.

Exits `0` when nothing is left missing, `2` when some files couldn't be read (they're listed) or the repository doesn't record ids. Re-running it is safe: files that already have an id are skipped.

```
swarmfile git backfill-oids
swarmfile git status   # now reports every counted file as having an id
```

`git enable`

Turns on git access for the selected mount's repository — the same one-way switch as the **Git access…** dialog in the web dashboard. Repository owners and org admins only; needs a running engine with the repository mounted. Enabling freezes the rules the repository is converted to git with: the conversion version, the size above which a file becomes an LFS pointer, and where history starts. The record is immutable afterwards (changing any rule would change commit ids a clone has already seen), so the command prints exactly what was frozen and on which repository.

Because the freeze cannot be undone, the command asks for confirmation first: `-y` / `--yes` skips the prompt, and a scripted run (`--json`, or no interactive terminal) must pass it rather than freeze silently.

Flag	Meaning
`--threshold-bytes <N>`	Freeze the LFS pointer threshold at `N` bytes instead of 1 MiB.
`--no-backfill`	Refuse instead of recording missing object ids first. For scripts that want the fail-fast behavior; interactive runs don't need it.
`-y` / `--yes`	Skip the confirmation prompt. Required with `--json` and when stdin isn't a terminal.

It refuses, with the reason, while the repository doesn't keep every saved version (turn that on from the repository's history settings), when an older branch or tag head can no longer be read, when history is too long to check in one pass, and on an end-to-end-encrypted repository — git access isn't available there.

If any at-or-above-threshold file still has no object id, the command records the missing ids itself first (the same work as `swarmfile git backfill-oids`: it hashes each file's content through the mount, fetching what isn't cached) and retries the enable once. `--no-backfill` skips that and surfaces the coverage refusal directly.

Exits `0` when the record was frozen, `2` on any refusal (the hub's sentence says which) or when the confirmation is missing, `1` when the hub couldn't be reached.

```
swarmfile git status          # everything has an id? then:
swarmfile git enable
swarmfile git clone acme/website
```

`git clone`

Clone a repository with git through the `swarmfile://` remote helper — the same as `git clone swarmfile://<hub-host>/<org>/<repository> [path]`. The relative `swarmfile://<org>/<repository>` is used when no hub is configured, when the configured hub is a path-prefixed reverse proxy, or when it is a plaintext host outside the loopback/`SWARMFILE_GIT_INSECURE_HOSTS` allowlist; the printed URL is the one the helper will reach as written.

It finds the `git-remote-swarmfile` helper next to this binary even when it isn't on your `PATH`. `<repository>` is a repository id, a bare repository name (qualified with your configured organization), or `<org>/<repository>`; `[path]` is the destination directory (git's own default when omitted). It runs `git clone` for you, so no running engine is needed.

The repository must have git access turned on — see Cloning a Repository with Git for what a clone contains, how it signs in, how `git push` works from it, and what isn't supported.

```
swarmfile git clone acme/website
swarmfile git clone acme/website ./website
```

`git anonymous-clone`

Show or set whether a **signed-out visitor** may clone a public, git-enabled repository. Anonymous cloning is **on by default**; while it is on, a signed-out visitor can clone, fetch and `git lfs pull` (never push) only while no active branch's committed tree hides a path through `.swarmfile/publicignore.yml` or a ready (published) release hid paths.

While one does, anonymous clone is refused outright — within about a minute — and the boundary is anonymous-only, so member ACLs still apply. An invalid or unreadable policy, or a boundary read/walk error, refuses anonymous clone (fail closed).

A release hiding paths cannot be published while anonymous cloning is on; once it is ready, turning anonymous cloning back on does not re-expose those paths. Turn it off to refuse all anonymous git reads (clone tickets stop immediately; an already-minted ticket is re-checked on its next read, and the 1-hour TTL only bounds the ticket row).

Needs a running engine with the repository mounted; the write is owner/admin only. With no argument it prints the current state.

```
swarmfile git anonymous-clone         # on or off?
swarmfile git anonymous-clone off     # refuse anonymous clones
swarmfile git anonymous-clone on      # allow them again
```

Exits `0` normally, `2` on a refusal (not an owner/admin — the hub's sentence says so), `1` when the hub couldn't be reached.

`git preflight`

Inspect a source repository — GitHub, GitLab, a local path — **without creating anything**. It fetches into a temporary bare mirror with your own `git` (so your configured credentials and `git-lfs` apply), then reports the default branch, every branch's verdict (`import`, `skip` or `refuse`), the tag policy, the LFS/submodule/octopus blockers, and the largest historical blobs against the literal-blob threshold. Nothing is created and no Swarmfile credential is read, so it works with the engine stopped and before you decide to migrate.

Exit `0` when `import` would create the repository, `2` when it would refuse (the report names the branch, tag or blob and the flag that changes it), and `1` on a fetch or git failure. `--json` prints the structured report; `--keep` keeps the temporary mirror.

Flag	Description
`--branches <GLOB>`	Assume the `--branches` filter you would pass to `import` (repeatable; the default branch is always kept).
`--no-tags`	Assume `--no-tags`.
`--annotated-tags <keep|convert|skip|fail>`	Assume the annotated-tag policy; `fail` reports the refusal, as `import` would.
`--submodules <skip|fail>`	Assume the side-branch submodule policy.
`--octopus <skip|fail>`	Assume the side-branch octopus-merge policy.
`--threshold-bytes <BYTES>`	Compare historical blobs against this literal-blob threshold instead of the hub's 1 MiB default.
`--max-blobs <N>`	How many of the largest at-or-above-threshold blobs to list (default 20; `0` skips the blob scan).
`--keep`	Keep the temporary mirror.

```
swarmfile git preflight https://github.com/acme/mono.git
swarmfile git preflight ./old-repo --submodules fail --json
```

`swarmfile git import --dry-run` is the same report without a second command.

Each source `git` child is bounded by `SWARMFILE_GIT_IMPORT_TIMEOUT_SECS` (default 300 seconds; `0` disables it). A timeout is a failed preflight, not a compatibility verdict; the command kills the child's process group rather than hanging.

`git import`

Bring an existing git repository — GitHub, GitLab, a local path — into a new Swarmfile repository, or into an existing one with `--into`. The repository is created with git access already on and every version kept, and the source's default branch becomes the repository's default branch (so upstream commit ids are preserved exactly, including a history whose default branch is `master`, `trunk` or anything else).

The source is fetched into a temporary bare mirror with your own `git`, so the same credentials a normal `git clone` would use — and a configured `git-lfs` — apply; each branch and tag is then pushed through the same `swarmfile://` helper `git clone` uses. It runs without a running engine, like `git clone`.

The repository name from the URL becomes the repository name unless `--name` overrides it. Every branch the git view can hold is imported; one it cannot (submodules, Git-LFS, an octopus merge) is skipped and listed, while the same problem on the source's default branch refuses the import before anything is created — nothing can land without it.

`--branches` narrows which branches are attempted (wildcards; the default branch is always imported) and `--no-tags` skips tags; `--submodules fail` / `--octopus fail` widen the refusal to side branches too.

Annotated tags keep their exact tag object by default — message, tagger, signature and id survive (`--annotated-tags keep` ; `convert` imports them as lightweight tags instead and the report says how many, `skip` leaves them out, and `fail` stops the import before creating anything).

Tags pointing at something other than a commit, or living only on a branch this import did not push, are skipped and listed.

`--threshold-bytes N` (positive) freezes the created repository's LFS threshold at N before the first push — needed when the repository's history already contains a blob above the 1 MiB default (it is creation-only, so it is refused with `--into` ; enable an existing target first).

`--dry-run` runs the full preflight and creates nothing (the same report as `git preflight`).

If anything fails to land, the temporary mirror is kept and the report names the retry: `swarmfile git import <url> --into <org>/<repository>` resumes against what already landed; the repository is resolved and its git access verified before the source is fetched, so a typo'd or disabled target is refused up front.

The import records the source it came from, so re-running the same command without `--name` / `--into` recognizes the repository it made and syncs new upstream branches and commits into it instead of failing on the name. `--new` forces a fresh repository anyway (a second repository from one source), and an explicit `--name` always creates the named repository.

The importer refuses, before creating anything, a source whose default branch uses Git LFS (the importer does not fetch LFS objects), contains submodules, or has an octopus merge — each is a limit of the importer and git access. A repository-name collision after the source is fetched is refused too; pass `--name` , or `--into` the existing repository.

Flag	Description
`--name <NAME>`	Repository name (default: the repository name from the URL).
`--org <ORG>`	Organization slug or id (default: your configured organization).
`--template <ID>`	A creation template from the repository-creation list (default: the `git` configuration without its starter `.gitignore` — an import needs the default branch to start empty, and the repository brings its own files, so a template that seeds files can't be imported into).
`--public`	Create a public plaintext repository instead of the default private managed one (private repositories need a plan with them).
`--into <ORG>/<REPOSITORY>`	Push into an existing repository instead of creating one; the repository is resolved (before the source is fetched), so an unknown name is refused up front.
`--keep`	Keep the temporary mirror after a successful import.
`--dry-run`	Run the full preflight and create nothing; prints the same report as `git preflight` .
`--annotated-tags <keep|convert|skip|fail>`	Annotated-tag handling: `keep` the exact tag object (default; message, tagger, signature and id survive), `convert` to lightweight, `skip` , or `fail` before creating anything.
`--new`	Create a fresh repository even when this source was already imported (the default without `--name` / `--into` is to sync into the repository this source made).
`--submodules <skip|fail>`	A branch containing submodules: `skip` (default) lists it and imports the rest, `fail` refuses the whole import.
`--octopus <skip|fail>`	Same for a branch containing an octopus merge.
`--branches <GLOB>`	Import only branches matching the wildcard (`*` matches any characters, `/` included); repeatable, and the source's default branch is always imported.
`--no-tags`	Skip every tag (branches still import).

Exits `0` when every ref landed or was deliberately skipped (a tag pointing at a blob or tree is listed as skipped, not failed), `2` on a refusal (the sentence names the fix) or when a branch or tag push failed (the report names the `--into` retry), and `1` on a failure such as the source being unreachable. `--json` prints the structured report. The creation-only flags (`--name` , `--public` , `--template`) are refused alongside `--into` , which already names the repository.

```
swarmfile git import https://github.com/acme/website.git
swarmfile git import git@github.com:acme/site.git --name website --org acme
swarmfile git import ./old-repo --into acme/website   # retry a partial import
swarmfile git import https://github.com/acme/big.git --threshold-bytes 50000000
```

`git url`

Print a repository's `swarmfile://` URL and exit, for wiring a remote by hand. `<repository>` is a repository id, a bare repository name (qualified with your configured organization), or `<org>/<repository>` .

The URL is host-qualified (`swarmfile://<hub-host>/<org>/<repository>`) whenever the configured hub can be written that way, so a copied link means the same repository on every machine; a path-prefixed hub, or a plaintext hub outside the loopback/ `SWARMFILE_GIT_INSECURE_HOSTS` allowlist, keeps the relative form.

URL segments are printed ASCII-lowercased and percent-encoded where a character needs it (a display name with a space comes out as `%20`) — the canonical web-address form, which also repairs a caps-typed UUID (a full `swarmfile://` URL is passed through unchanged; the remote helper decodes an escaped segment before resolving it).

```
git remote add origin "$(swarmfile git url acme/website)"
```

`git force-forward`

The append-only form of `git push --force`, run inside a clone. A branch's tree can't be rewritten on Swarmfile, but the end state a force push wants — "the branch's tree is my tree" — can be landed as a *new* commit on top of the current head whose tree is your tip's, with your tip as its second parent.

This command fetches the branch (which also snapshots uncommitted drive saves, like any fetch), builds that commit with git's own `commit-tree` (so its id is derived on this machine), and pushes it through the normal helper path: every commit stays reachable, teammates fast-forward instead of diverging, and your tip's history rides the `merged/<sha>` side branch the helper creates and archives.

Only committed work is landed (a dirty worktree is ignored, like `git push`), and `--dry-run` still fetches — the dry run skips the push, not the fetch, so a fetch's usual effects apply.

Your local branch is deliberately not moved — the new commit's id cannot match your tip's — so the command prints the `git fetch <remote> && git reset --hard <remote>/<branch>` that aligns your clone.

Pass a `BRANCH` (default: the checked-out branch), `--message <MESSAGE>` for the new commit's message (default `force-forward: <your tip's subject>`), `--remote <REMOTE>` (default `origin`), or `--dry-run` to run every check and push nothing.

When your tip simply continues the remote head along its first parents (an ordinary fast-forward), a normal `git push` lands the same end state, and the command says so instead of building a graft; the diverged *merge* shape (merging the fetched branch into your line) does need the graft, and the push refusal for it names this command.

When the branch already carries your tree there is nothing to do either, and the command just prints the realignment.

A clone created through the helper enables the same translation for a plain force push by default (`swarmfile.force-forward=true`); an older clone, or an explicit `git config swarmfile.force-forward false`, keeps force pushes refused. With it on, `git push --force` (and `--force-with-lease`, whose lease check still runs client-side against the advertised head) pushes the tip as-is when it is an ordinary fast-forward, else builds the same forward commit; a note names the new id and asks you to `git fetch` and reset, since git's own bookkeeping points your remote-tracking ref at the tip you sent.

Exits `0` when the push landed (or there was nothing to do), `2` on a refusal — detached `HEAD`, an unknown local branch or branch name, a fetch that couldn't find the branch, a push the helper refused — and `1` on a transport failure. A rewritten **root** commit cannot be landed this way: the replaced history shares no commit with Swarmfile, so the push refuses naming the unrelated root — push that as a new branch instead. `--json` prints the outcome — `pushed`, `fast-forward`, `already-at-tip` or `already-carries-tree` — with the commit ids; `pushed` and `already-carries-tree` carry `align`, the realignment command for scripts.

```
swarmfile git force-forward                   # this clone's branch
swarmfile git force-forward release -m "keep our tree"
swarmfile git force-forward --dry-run
git config swarmfile.force-forward true       # let plain --force use the same path
```

`git index`

Derives commits' Git-ready trees on this machine and hands them to Swarmfile, so a committed version can be served to git directly. For each commit it reads the branch state as of that commit's version, builds the tree objects (directories, symlinks and the large-file pointers, splitting very large directories into segments), uploads anything the hub doesn't already have, and claims the result. A second machine — a teammate's app, a fetch, or an API key used in CI — confirms the same value.

With no argument it works on the selected mount's branch head and every unconfirmed commit behind it (through merges too), back to the nearest commit that is already confirmed or that you claimed yourself. Each commit is derived independently and the claims go oldest first, so a chain someone else pushed is confirmed in one run. A run handles up to 1,000 commits and prints one line per commit. Given a commit's sequence number (`#42` or a bare number), it handles just that commit.

The git remote helper also does this by itself for a small unclaimed chain: when a git clone, fetch, push or `ls-remote` meets a branch head that has never been derived, it derives and claims up to 200 commits inline (a clone that set `swarmfile.confirm=false` opts out). This command is for longer chains, or a machine without the Desktop App.

Requires a running engine and a managed or unencrypted repository. Exits `0` when every claim was accepted (including ones still waiting for a second machine to confirm), `2` when the hub disputed or rejected one (later commits in the run aren't claimed), and `3` when the hub couldn't be reached.

A commit printed as `rejected` (`resolved_against` in `--json`) was already settled to a different value when the hub resolved a dispute over it, so this machine's claim is kept only as evidence; compare the two with `git disputes SEQ --all` and report both hashes, since it is a client bug unless the repository's history changed. If the settlement left the commit provisional, an owner who believes it is wrong can override it with `git clear-claim`; a settlement that confirmed the commit is final.

A commit printed as `provisional (your earlier claim was kept)` (code `self_disagreement` in `--json`) means this machine derived a different value than its **own** earlier claim for the same commit. One identity deriving two values for one commit is a client bug, not a dispute: the hub keeps the first claim, records the disagreement as evidence (readable with `git disputes SEQ --all`), and does not alert owners. Report it with both hashes.

```
swarmfile git index                # the branch head and every unconfirmed commit behind it
swarmfile git index '#42'          # one commit
swarmfile git index --json
swarmfile git index --verify-full  # ignore the cache; re-derive and check every object
```

`--verify-full` re-derives without the incremental/sparse caches and checks each derived object against its content address before uploading — a slow but complete self-check for a suspicious or freshly upgraded machine.

`git clear-claim`

Owner/admin only: resets the commit claim at a sequence number on the selected mount's repository. Most claims are never cleared by hand: Swarmfile's server (for a pushed commit) or a teammate's fetch confirms a provisional one, and a disputed one is settled by Swarmfile deriving it again on the server, usually within a minute or two.

This is the escape hatch for a claimed commit that can't be served: a **disputed** row (two derivations disagree), a claim whose root tree is gone, or a **stranded snapshot-mint head** — `git push` mints the branch's live tree as a snapshot when it has drive saves, and if the push is then refused, that snapshot stays as the provisional head; clearing it also rolls the branch head back to the snapshot's parent, so the next fetch mints afresh from the live tree.

A healthy claim — including a member's own — is never discarded: the hub answers `cleared: false` and says why.

Clearing resets the hub's record of the claim, so it is confirmation-gated: `-y` / `--yes` skips the prompt, and a scripted run (`--json`, or no interactive terminal) must pass it. Dispute evidence is kept, not deleted: an owner or admin can read both values, their trees, parents and claimants before or after a clear with `git disputes` (add `--all` once it is cleared), or in the **Commit disputes** section of the repository's Commits page on the web.

Argument/flag	Meaning
`SEQ`	The changeset sequence number to clear.
`-y` / `--yes`	Skip the confirmation prompt. Required with `--json` and when stdin isn't a terminal.

Exits `0` when the claim was cleared (an already-confirmed or archived seq prints that instead), `2` on a refusal (not an owner/admin) or when the confirmation is missing, `1` when the hub couldn't be reached.

```
swarmfile git index '#42'          # see what the hub holds first
swarmfile git clear-claim 42       # reset it; a stranded snapshot head rolls back
swarmfile git clear-claim 42 -y
```

`git disputes`

Owner/admin only: lists the commit disputes on the selected mount's repository, newest first. A dispute means two derivations of the same commit produced different hashes, so git access can't serve that commit until Swarmfile settles the dispute (usually within a minute or two, by deriving the commit again on the server) or an owner or admin clears its claim.

Each block shows the commit's sequence number, when the dispute was recorded, the claim's current status, and both sides: the challenging claim and the one it contradicted, each with its commit hash, tree, parents and claimant. A claimant is a user, an API key, or Swarmfile's server-side derivation.

By default only open disputes are listed: not yet cleared, on a commit that is still disputed or provisional. Resolve one Swarmfile could not settle, or override a settlement that left the commit provisional, with `git clear-claim`; the evidence is kept, and Swarmfile derives the commit again on its own (unless its own derivation was part of the dispute), as does the next `swarmfile git index` or a teammate's fetch.

Argument/flag	Meaning
`SEQ`	Only the disputes for this commit sequence number.
`--all`	Include cleared and evidence-only disputes.

`--json` prints the hub's list unchanged. Exits `0` on success (an empty list included), `2` on a refusal (not an owner/admin), `1` when the hub couldn't be reached.

```
swarmfile git disputes             # open disputes on this repository
swarmfile git disputes 42 --all    # every record for commit 42, cleared ones included
swarmfile git disputes --json
```

`offline`

Pack & Go: materialize a scope **and** reserve the whole scope with one long-lived offline reservation, so it stays editable with the hub unreachable. `hydrate` above only makes content readable offline; this is the form that also makes it writable. It runs as a background job, and the scope follows the selected mount's branch (target another mount with the global `--mount <id>`).

Subcommand	Flags	Description
`prepare`	`[path]`, `--duration-secs <N>`, `--offline-days <D>`, `--no-wait`	Materialize and reserve. The lease defaults to 7 days, clamped at 30; a lease is only renewed while online, so `--offline-days`, when longer than the lease, **extends the lease to cover the trip** (an explicit shorter `--duration-secs` is never overridden — it warns instead). Prints how many files the reservation covers. If another machine holds an overlapping reservation or is editing a file inside the scope, the prepare is refused naming the holder and nothing is reserved. Then polls to completion unless `--no-wait`.
`status`	—	Progress of the current (or last) job — `interruptedScope` names one a restart interrupted — plus what this engine holds right now (file edit locks and folder/repository reservations, with an expiring-soon and renewal-failure warning).
`cancel`	—	Ask a running prepare to stop — reservations already taken are kept, and the content download this prepare started is stopped too (only that job, never a newer one).
`resume`	—	Complete a prepare a restart interrupted; re-reserves the scope idempotently.
`return`	`[path]`	Release reservations: at or under one path, or with no path every reservation this engine holds (per-file locks and scope reservations alike). In-scope saves are given a few seconds to land first; any still syncing are reported.

```
swarmfile offline prepare ./Repositories/shots/seqA --offline-days 14
swarmfile offline status
swarmfile offline return ./Repositories/shots/seqA
```

`fetch`

Pull a byte range of one file into the local cache — including from a version somebody else is **still uploading**.

On a slow link a large upload takes hours or days, and until it finishes the new version isn't readable at all. This asks the uploading machine to send the part you need *first*, then downloads it as it lands. What you get back is the time it takes to send that range, not the time it takes to send the file.

Flag	Description
`--tail <bytes>`	Fetch the last N bytes. The common case for media, where the part you need is at the end. Resolved against the *in-flight* size, which is why it's a flag rather than arithmetic you do yourself — see below.
`--start <bytes>`	First byte of the range. Default `0`. Conflicts with `--tail`.
`--len <bytes>`	How many bytes from `--start`. Clamped to the end of the file. Conflicts with `--tail`.
`--version <which>`	`newest` (default — the in-flight version if there is one, otherwise the committed one), `pending` (fail unless something is uploading), or `head` (the committed version only).
`--follow`	Track a moving reader instead of fetching a fixed range: keep a buffer ahead of wherever it is and re-target when it seeks. `--start` becomes the initial position; `--len` and `--tail` are rejected together with `--follow`.

```
# The last 200 MB of a file a colleague is still uploading
swarmfile fetch ./Repositories/reel-04.r3d --tail 209715200

# A specific window of the committed version
swarmfile fetch ./Repositories/site.rvt --start 0 --len 52428800 --version head
```

The command polls until every block has landed, printing progress as it goes:

```
$ swarmfile fetch ./Repositories/reel-04.r3d --tail 209715200
19/160 blocks, 25.8 MiB fetched (waiting on the upload)
...
fetch complete: 160/160 blocks, 200.0 MiB now local
```

"waiting on the upload" is the normal state, not a stall. The blocks you asked for may not exist yet — the other machine is still sending them at its uplink speed — so this command can legitimately run for a long time. It gives up only when nothing is arriving *and* the hub reports that the upload has stopped, and says so rather than hanging.

Why `--tail` rather than working out the offset yourself: the size to measure back from is the **in-flight** version's, and the file's committed size is still the old version's. A number you compute locally would measure from the wrong end of the wrong file.

`--version pending` fails outright when nothing is uploading, rather than quietly giving you the committed version. That is deliberate: somebody who asked for the version in progress and silently received yesterday's bytes has no way to tell.

Fetched blocks are pinned exactly like `hydrate`, so they aren't evicted by the cache's LRU sweep before you open the file — and they are released the same way, with **`swarmfile hydrate release <path>`**. One job runs at a time; starting a second while one is running is refused rather than queued.

You can also do this without the CLI: reading the file normally through the mount always works, and gets the committed version. `fetch` is for the case where the version you want is still on its way. (A mount also streams a file that has never finished uploading — see the config reference.)

Following a moving reader

A fixed range is the right shape for "give me the last 200 MB". It's the wrong shape for watching a file as it uploads, because a viewer plays forward and seeks. `--follow` keeps a buffer ahead of a read position you report as it moves, and re-targets when it jumps.

Command	Flags	Description
`fetch <path> --follow`	`--start <bytes>`, `--version <which>`	Start following from `--start` (default `0`). Returns once the job is running; it does not complete on its own.
`fetch-seek`	`<position: u64>`	Report where the reader has got to. The buffer re-targets there immediately. A no-op if nothing is following.
`fetch-status`	—	Progress of the current (or last) `fetch`. The way to watch a `--follow` job.

```
swarmfile fetch ./Repositories/reel-04.r3d --follow --version pending
swarmfile fetch-seek 5000000
swarmfile fetch-status
```

For a follow job, `fetch-status` reports `position`, `bufferedBytes` and `readRateBps` rather than a block count. **`bufferedBytes` is the number worth watching**: it counts contiguous bytes from the read position, so it answers "will this play" — where a block total does not. A hole two blocks ahead stalls a reader however full the rest of the window is.

Buffer depth is derived from how fast the reader is actually consuming, not from a fixed byte count: 30 seconds of a 6 Mbit/s proxy and 30 seconds of a 200 Mbit/s master are very different numbers of bytes, and a single figure would be useless for one of them. Set the seconds with `SWARMFILE_STREAM_BUFFER_SECS` (default 30). A seek is not counted as playback, so scrubbing doesn't inflate the estimate.

A follow job ends when you cancel it, when the upload finishes and the buffer reaches the end of the file, or after five minutes with no `fetch-seek` — a paused viewer reports nothing, and one job runs at a time, so an abandoned follow would otherwise block every later fetch.

`seed`

NAS/seed-node mode.

Subcommand	Description
`status`	Current seed-mode state.
`enable`	Turn on seed mode. Requires the Pro plan or above, and is mutually exclusive with Cloud-only mode.
`disable`	Turn off seed mode.

> `seed enable` / `disable` restart the engine.

> A headless cache is easiest to set up from the dashboard: **Settings → Office caches** generates the complete `seed.env` (scope and credential included) and lists the fleet's liveness and served bytes. `seed enable` is the toggle for an engine whose scope is already configured.

`watch` / `unwatch` / `watches`

Command	Description
`watch <path>`	Watch a file or folder.
`unwatch <path>`	Stop watching a file or folder.
`watches`	Everything you're currently watching in this repository.

`lock` / `unlock` / `locks`

An explicit, up-to-30-day edit lock — like `git lfs lock` / Perforce's `p4 edit`. Not the same as the short-lived, automatic lock a plain write takes (from an app's first write to the file until it closes it); this one you take and release yourself.

Command	Flags	Description
`lock <path>`	`--duration-secs <N>`	Reserve exclusive edit rights on a file, even offline. Defaults to 7 days, clamped to a 30-day maximum. Refused (exit 2) if someone else already holds it. Re-running `lock` on a file you already hold extends it, and a held lock is auto-renewed every 6 hours while the engine is online.
`unlock <path>`	—	Release a lock taken with `lock`.
`lease <path>`	`--duration-secs <N>`, `--release`, `--token <T>`	Hold or extend the short session lease (default 120s; the companion renews at half) on the ordinary standard lock. Unlike `lock`, this is not a Pack & Go checkout: the engine heartbeats it, it is not kept for offline use, and it releases within its deadline if the lease holder stops renewing. The acquire response carries a `sessionToken`; `--release --token T` drops only that caller's reference.
`stems check`	`--manifest <path>`, `--song <id>`, `--all`, `--project <dir>`, `--strict`, `--quiet`, `--max-set-mb <n>`, `--quality`	Native Ableton handoff validator (no Node): manifest, Set media references, WAV sample rate/channels/bit depth, data length and duration alignment. `--max-set-mb` raises the scan cap; `--quality` reports sample peak/clipped samples (not LUFS/true peak).
`stems setup`	`--manifest <path>`, `--song <id>`, `--project <dir>`, `--add-role <id>`, `--write`	Native per-song setup checklist; optionally add roles/folders. Never edits the ALS.
`stems companion`	`--manifest <path>`, `--song <id>`, `--project <dir>`, `--watch-only`, `--no-watch`, `--interval`, `--stems-interval`, `--duration`, `--title-match`, `--lease-token`, `--process-only`, `--parent-pid <pid>`, `--once`, `--dry-run`, `--no-notify`	Native companion: short session lease, current stem content-CID (`rootCid`) watcher and notifications. `--process-only` leases a non-Live DAW project and skips Live detection; `--lease-token` resumes the same per-caller lease after a companion restart; `--parent-pid` watches a parent tray PID and exits/releases the lease if it disappears. `--dry-run` prints the lease commands without acquiring or releasing a lease. No Node.
`stems status`	`--manifest <path>`, `--song <id>`, `--project <dir>`, `--json`	Native dashboard: the Set and each role's engine sync state and current `rootCid`.
`locks`	—	Every file currently locked — for editing, or being written somewhere — in this repository's current branch.

```
swarmfile lock ./Repositories/tower-a.rvt
swarmfile lock ./Repositories/tower-a.rvt --duration-secs 86400   # 1 day
swarmfile locks
swarmfile unlock ./Repositories/tower-a.rvt
```

If someone else needs the file back before you release it, they can ask via `unlock-request` below — there's no force-release from another user's side.

Multi-song music repositories use one manifest per song at `Songs/<song>/stems.json`; pass that manifest path (the Desktop App's Music page does this automatically). A repository with the `Song/` layout keeps `Song/stems.json` as the default.

`chmod`

Mark a file executable or not — the same bit `chmod +x` sets on the mounted drive, and the one git records as mode `100755` (a script, a build tool) versus `100644`. Swarmfile stores only this one bit: there are no separate read/write bits per user.

Command	Description
`chmod +x <path>`	Make a file executable.
`chmod -x <path>`	Make it a plain, non-executable file.

```
swarmfile chmod +x ./tools/build.sh
swarmfile chmod -x ./docs/notes.txt --json
```

- Files only: a folder or symlink is refused (`not_a_file`).
- Works offline. The change shows at once on this machine and is sent when the hub is reachable again; the summary line (or `"queued": true` under `--json`) says when that's still pending.
- On the macOS and Linux drive, `chmod` does the same thing: any execute bit (`u+x` , `g+x` or `o+x`) makes the file executable, and none makes it not. The drive shows executable files as `rwxr-xr-x` and every other file as `rw-r--r--` .
- Windows has no execute bit, so `swarmfile chmod` is how you set it there. Editing a file on Windows keeps its bit, and so does an app that saves through a temporary file and renames it over the original.

`comment` / `comments`

Command	Flags	Description
`comment <path>`	`-m, --message <String>` (required)	Post a comment on a file.
`comments <path>`	`--limit <N>` , `--offset <N>`	List comments on a file, newest first. The hub caps one page at 199; `--offset` pages through the rest.

`field`

Folder profiles — the typed columns ("Net", "Gross", "Payment Ref", …) a folder subtree can carry, defined org-wide with optional per-repository overrides. Values are hub-side, ACL-gated metadata (whoever can read the file can read its values), so each command resolves the path locally on this machine and relays the hub.

Read display formatting matches the web app's spreadsheet view, and `export` / `import` share the grid's CSV shape — including its formula-injection guard on text cells and its canonical number form.

Each write command (`set` , `unset` , `import`) applies through **one** metadata commit: an aborted batch commits nothing (never a partial write), and a later command always starts a fresh one.

All three write commands answer the same summary — `applied` , `skipped` , `errors` , plus the hub's `message` when a step was refused and `aborted` for the two refusal classes that name themselves (`conflict` on a 409/ `field_value_conflict` , `forbidden` on a 403; a 400-class refusal carries the hub's message with no `aborted`) — which `--json` prints raw; any skip, error or abort exits 2, with the per-item reasons on stderr.

Subcommand	Flags	Description
`profile [PATH]`	`--root`, `--json`	The profile that applies to a folder — the nearest bound ancestor's, else the repository default — printed as its name then one `heading (type)` line per field. `--root` asks for the repository default directly (the tree's root level has no entry id, so no path reaches it); otherwise PATH is required and may be absolute or relative to the current directory. A folder with no profile prints the one line `No folder profile is bound` and still exits 0 — an explicit state, not an error.
`show <path>`	`--json`	One entry's **set** field values under its resolved profile, one `<heading>: <value>` line per set field (absent fields are omitted): numbers grouped with thousands separators, `currency` pinned to 2 decimals (a profile field carrying an ISO 4217 currency code renders with its symbol in the web grid and the Desktop App's inline columns; this line stays the plain 2-decimal number), `bool` as `Yes` / `No`, dates and text as-is — the same formatting as the web grid. A file or folder with no bound profile exits 2 (`No folder profile is bound to <path>`); a profile with nothing set yet prints nothing and exits 0. `--json` prints the raw payload (`entryId`, `found`, `profile`, `values`).
`export [FOLDER]`	`--root`, `--csv`, `--json`	The folder page as a spreadsheet. `--root` exports the repository-default profile's root page instead (one of FOLDER or `--root` is required, and they conflict: the tree's root level has no entry id, so no path reaches it). `--csv` renders RFC 4180 CSV: header `name` plus each field's heading, one row per listed entry with the **raw stored** values (never the formatted display), a cell containing a comma, quote, CR or LF quoted with `"` and its inner quotes doubled, and `\n` row separators (trailing newline included). Text cells — headings, entry names and string values — starting with `=`, `+`, `-`, `@`, a tab or a CR get a leading apostrophe so a spreadsheet treats them as text, never a formula; numbers and booleans are typed data and stay as-is (numbers render in the web's canonical form, `1e16` as `10000000000000000`). Without `--csv` — or under `--json` — the raw JSON rows print instead. A folder with no bound profile refuses `--csv` with exit 2 (there are no headings to export); an unknown path or a hub/control failure exits 2 with a one-line reason.
`set <path> <HEADING=VALUE>...`	`--json`	Set field values on one entry under its resolved profile; the assignments are repeatable and all ride one metadata commit. Cells are pre-checked per item like the web editor's own parse: `number` / `currency` as numbers (`Net=1234.50`; hex/bin/oct spellings are refused), `bool` as `true` / `false`, `date` as `YYYY-MM-DD` with a real-calendar check, `enum` against the field's options, `url` as an absolute http(s) address, a field `pattern` as an anchored whole-value match, and `text` / `url` / `user` trimmed as typed. A blank value **clears** the field — the web editor's emptied-cell rule, `bool` included (an emptied bool cell clears it). An unknown or ambiguous heading, or an invalid cell of a locally checked type, exits 2 with the per-item reason on stderr while the remaining assignments still commit; a value the HUB rejects (the local checks are a pre-check — the hub stays the final validator) aborts the batch instead: nothing commits, and the hub's own message is reported. A conflict (a value that moved under you) likewise commits nothing and exits 2 (`aborted: conflict`; `aborted` appears for conflict/forbidden only). No bound profile is exit 2.
`unset <path> <HEADING>...`	`--json`	Clear the named headings on one entry — a cleared field is a deleted value, never a null — all in one metadata commit. An unknown or ambiguous heading exits 2 with the reason on stderr; the rest still commit.
`import <FOLDER>`	`--csv <FILE>`, `--json`	Apply a CSV to a folder page in one metadata commit — the inverse of `export --csv`. The header's leading `name` column plus the field headings (any order) map each row to a listed entry; the file follows RFC 4180 (quoted commas/quotes/newlines, CRLF or LF records, blank lines skipped, a leading BOM dropped), so the export's own output round-trips — including the export's formula guard: a leading apostrophe before a `=`, `+`, `-`, `@`, tab or CR is stripped again on import, so a guarded text cell (or a guarded entry name) returns as the original value. A **blank cell clears** that field for that row, like the grid's import (`bool` included). Rows naming no listed entry (or with an empty name cell), heading labels that match no field or match more than one, and invalid cells of the locally checked types (`number` / `currency`, `date`, `enum`, `url`, `pattern`, `bool`) are reported per item on stderr and exit 2 — the valid cells still commit. A value the HUB rejects aborts the whole batch instead: nothing commits and the hub's message is reported. A name repeated on several rows updates that entry once per cell (the last row wins where they disagree) — the summary counts that entry's cells once, where the web grid counts every row's; the stored result is identical. Every echoed cell in the messages carries the export's formula-injection guard. A missing or unreadable `--csv` file, or a folder with no bound profile, exits 2 before any request.

```
swarmfile field profile ./Repositories/invoices
swarmfile field profile --root
swarmfile field show ./Repositories/invoices/INV-0042.pdf
swarmfile field export ./Repositories/invoices --csv > invoices.csv
swarmfile field export ./Repositories/invoices            # raw JSON rows instead
swarmfile field export --root --csv > root-page.csv   # the repository default's root page
swarmfile field set ./Repositories/invoices/INV-0042.pdf Net=1234.50 Paid=true
swarmfile field set ./Repositories/invoices/INV-0042.pdf Note=   # a blank value clears the field
swarmfile field unset ./Repositories/invoices/INV-0042.pdf Net Paid
swarmfile field import ./Repositories/invoices --csv invoices.csv   # the export shape, back in
```

`trash`

Soft-deleted entries in this mount's repository — the headless equivalent of the Desktop App's Trash page.

Subcommand	Flags	Description
`list`	`--limit <N>` (default 50), `--offset <N>`	Your own soft-deleted entries, newest-deleted first.
`restore <entry-id>`	`--with-nearby`	Undelete back to its original location. Refused with `name_conflict` / `parent_deleted` (exit 2) if something else has since taken that name or the parent folder is itself gone — either way the response names the conflicting entry so you can resolve it by hand. A folder deleted from the desktop drive is deleted file by file, so restoring it brings back the folder and lists the items deleted inside it just before it (`nearbyDeleted` under `--json`); `--with-nearby` restores those too.
`purge <entry-id>`	`--preview`, `-y, --yes`	Permanently delete ("Delete Forever") — **not** recoverable, unlike a normal delete. `--preview` reports how many descendants a folder would take with it, without removing anything. Asks first; `--yes` skips the prompt (or refuses on a non-interactive stdin / under `--json`).

```
swarmfile trash list --limit 100
swarmfile trash restore 3f9a2b --with-nearby
swarmfile trash purge 3f9a2b --preview
swarmfile trash purge 3f9a2b --yes
```

`unlock-request`

Ask someone to release a file they have locked — distinct from `lock` / `unlock` above (edit locks *you* hold); this is the "please give it back" workflow for a lock someone **else** holds, the same feature the Desktop App's request-unlock modal drives.

Subcommand	Flags	Description
`create <path> [-m, --message]`	—	Ask the current holder to release `path`. If it just freed up on its own, the hub reports that directly (`noLock: true`) instead of creating a request nobody needs to answer.
`list [--status] [--entry-id] [--limit]`	`--status` (default `pending`)	Unlock requests you're involved in, as requester or holder.
`respond <entry-id> <request-id> --decision <granted|denied>`	—	As the current **holder**, grant or deny a pending request against a file you have locked. Exit 2 if it's already resolved or isn't yours.
`cancel <request-id>`	—	As the **requester**, withdraw a still-pending request. Exit 2 if it's already resolved.

```
swarmfile unlock-request create ./Repositories/tower-a.rvt -m "need to push a fix before EOD"
swarmfile unlock-request list --status pending
swarmfile unlock-request respond 3f9a2b req_88 --decision granted
```

`presence` (alias `who`)

Who's currently editing what in this repository.

`check-ignore <path>`

Report whether a path is excluded from sync by `.gitignore` / `.swarmfileignore`, and by which rule — the diagnostic counterpart to `ignore-clean`. Answers "why isn't this file syncing?" the way `git check-ignore -v` does.

```
swarmfile check-ignore ./Repositories/node_modules/pkg.js
```

`ignore-clean [path]`

A `git rm --cached` equivalent: remove already-hub-synced content that matches this machine's own `.gitignore` / `.swarmfileignore` sync-exclusion rules (see `check-ignore`) — for content that synced before the matching rule applied, or from a machine whose ignore rules differ.

The hub delete is a **soft delete** (recoverable from Trash for a while), and a matched directory's whole subtree is removed in one call rather than entry by entry.

Prints a size estimate and asks for confirmation first, unless `--yes`; without `--yes` a non-interactive stdin or `--json` refuses (exit 2), and `--dry-run` prints the estimate without confirming or deleting.

Flag	Description
`[path]`	Mount-relative path to narrow the scope to a folder or file. Omit for the whole repository.
`--dry-run`	Print the size estimate and exit without deleting anything — the preview of exactly what `--yes` would remove.
`-y, --yes`	Skip the confirmation prompt.
`-f, --force`	Cross-machine safety override. Alongside `--yes`, proceed even if a top-level candidate was last modified by **someone other than you** within the last 7 days. Without `--force`, `--yes` refuses in that case. Interactive mode is never gated by this — a human already sees the same who/when detail and answers the prompt regardless.

```
swarmfile ignore-clean
swarmfile ignore-clean ./Repositories/old-import --yes
```

With `--yes --json` it prints `{"deleted":[ids],"failed":[{"entryId","error"}],"paused":[{"entryId","jobId","retryAtSecs","message"}]}`. A `paused` entry is a large folder the hub has already moved to Trash, but whose cleanup is paused because the repository used up today's write budget; it finishes on its own (the human output prints it as `paused: …`, not as removed). Exit `2` when any delete failed, `75` when none failed but one is paused, otherwise `0`.

`completions <shell>`

Generate a shell completion script. Works without a running engine. Supports `bash`, `zsh`, `fish`, `elvish`, `powershell`.

```
swarmfile completions zsh > ~/.zsh/completions/_swarmfile
```

`doctor`

Connectivity diagnostics — asks the running engine to run its probe suite. `--latency` adds the **hub latency** check: it times `/health`, the org's repository list, and the repository's `branches` / `changesets` routes three times each and reports min/avg/max, so a slow push or download can be attributed to the edge, the organization, or the repository's service. For a machine where the engine isn't running, use the standalone `swarmfile-doctor` binary instead (it takes `--latency` too).

Flag	Description
`--latency`	Also measure per-route hub latency (health, the org repository list, the repository's branches/changesets). Adds a few round trips per route.

`log`

The commit log as a compact, colored feed (like `git log --oneline`), scoped to the branch this mount is on. Each line shows the commit's 12-char hash (the full hash is what `hash:` refs and CI pins use).

Argument / Flag	Description
`[REF]`	Start the log at a commit and walk back from it. Full ref vocabulary (`#N` / `HEAD~N`, a branch, a tag, a changeset id, or a 64-hex hash). A branch or tag lists **that** branch's history (`log feature` on a main mount shows feature's commits); a bare number, hash, or changeset id walks the mount's branch. `--branch` wins when given.
`--limit <N>`	Default `20`
`--branch <name>`	Log another branch's history (read-only) instead of the mount's.

`steps`

The harness-written agent timeline: commits whose message is `step N: ...` (newest first). The PostToolUse hook writes one after each mutating tool call, so this is the "what did the agent just do" view. Restore any step with `swarmfile restore --to <seq>`; the web History page has the same list behind **Steps only**.

Flag	Description
`--limit <N>`	Default `50`.
`--branch <name>`	Read another branch's steps (read-only).

```
swarmfile steps
swarmfile steps --limit 20 --branch agent/render-farm-a1b2c3
swarmfile restore --to 482
```

`tail`

Watch this repository's activity live — comments, watched-file changes, lock conflicts. Ctrl+C to stop.

This is the engine's **local** event bus for the mounted repository, not the org-wide Activity feed — for that, use `activity` (`list` / `export` / `tail`). The global `--mount` flag has no effect on `tail`: the stream covers every open mount's activity, not just the current one.

Headless engines

`api-key`

Repository-scoped API keys, for an engine that runs where nobody can complete a browser sign-in — a render-farm node, a CI runner, a build box. A machine holding one authenticates without an interactive OIDC session at all.

Subcommand	Description
`list`	Every key for this mount's org. Never shows key material.
`create <name> [--repository <id>] [--branch-scope <branch>] [--path-glob <glob>]... [--deny-glob <glob>]...`	Mint a key. `--repository` defaults to this mount's own repository; it takes the repository **id**, not its slug (an id the org doesn't have is refused with `repository_not_found`). `--branch-scope` binds the key to one non-default branch and keeps git, history and merge requests there; repeatable `--path-glob` / `--deny-glob` make it engine/API-only and require a branch scope. The branch must already exist and be non-default.
`revoke <id>`	Revoke by id (from `list`). Immediate — the key stops authenticating right away.
`rate-limit <id> [max] [--clear]`	Set (or, with `--clear`, clear) that one key's own request ceiling, in requests per minute. Capped at the plan's per-user maximum — use it to let a render-farm/agent key burst above the plan's flat default.

Minting and revoking require an **admin or owner** role in this mount's org. The same mint/list/revoke lives on the dashboard's API keys page and the tray's API keys panel.

On a self-hosted community server a key stands for a person, so `list`, `create` and `revoke` work with the key the engine runs as. An **owner** key (what `swarmfile-server admin mint-key` makes by default) manages every key in the organization; a **member** key manages only itself and the keys it created.

A key created here is a member key for one repository; a key scoped to one repository, an owner's included, manages only that repository's keys. There is no plan cap, and `rate-limit` is not available there. Scoped branch/path keys are hosted-only; the community server rejects the scope fields. Roles and the rest are described in Self-hosting: Keys and access.

Minting is capped by the plan's **headless-seat allowance** — Free 1, Starter 2 per seat (min 5), Pro 5 per seat (min 10), plus any purchased headless seats at $2/month each — and a mint past the cap is refused with `keys_cap`; revoke an unused key or add a headless seat under Settings → Billing. Revoking frees the slot immediately, and a downgrade never revokes keys you already have — it only blocks minting new ones past the new allowance.

> `create` prints the raw secret **once**. The hub never returns it again — copy it before you scroll. With `--json` the secret is in the response body's `key` field, which is what makes it scriptable; treat that output as the credential it is.

A key is walled off to exactly one repository. It cannot read or write anything else in the org — not even repositories the admin who minted it can see — so a compromised farm node is confined to the repository it was built for, and revoking a key stops that one machine without touching anybody's own session.

A **branch-scoped** key is narrower still: `--branch-scope` pins it to one non-default branch and keeps git, history and merge requests on that branch. Adding `--path-glob` / `--deny-glob` makes it engine/API-only: git, history, the commit walk and merge requests refuse it. The branch must already exist and be non-default.

The scope is set at mint time — revoke and mint a new key to change it — and each allow/deny list holds at most 64 globs of at most 1,024 characters. Minting does not validate the branch; the hub refuses a dead or default branch on first use. The dashboard and tray use the same fields.

Using one. Set it as `SWARMFILE_API_KEY` on the machine, and the engine skips sign-in entirely: no browser, no refresh token, no impersonating whoever happened to mint it. If both `SWARMFILE_API_KEY` and `SWARMFILE_OIDC_REFRESH_TOKEN` are set the API key wins, and the engine says so in its log rather than choosing quietly — a leftover personal refresh token on a shared machine is the more likely mistake of the two.

```
swarmfile api-key create farm-node-12 --repository proj_9f2a
```

If the mount covers more than one repository, `--repository` is required: there is no single obvious answer, and guessing one would scope a credential to the wrong place.

See Deployment Topologies for how this fits with per-node config files and seed nodes.

`repository`

Repository lifecycle and settings in this mount's org — the headless/scripted equivalent of the Desktop App's repository settings.

Subcommand	Flags	Description
`create <name> [--e2e] [--public] [--commit-mode] [--repository-type] [--keep-full-history] [--description] [--git-compatible] [--acl-mode] [--hub-only] [--residency] [--protect-default-branch] [--initial-branch] [--option]`	`--e2e` ; `--public` ; `--commit-mode <auto-commit|staged>` ; `--repository-type <revit_worksharing>` ; `--keep-full-history <true|false>` ; `--description <text>` ; `--git-compatible` ; `--acl-mode <open|protected>` (Open by default; `protected` needs `acls`); `--hub-only <true|false>` ; `--residency <eu|us|fedramp|fedramp-high>` (pins this **repository's** region at creation — per-repository and independent of the org's, fixed thereafter; needs the plan's residency capability *and* provisioned storage for the region); `--protect-default-branch` ; `--initial-branch <name>` (the default branch's name — most repositories keep `main` ; fixed at creation); `--option <key=value>` (repeatable — any create option by manifest key, e.g. `template=blank	media
`list`	—	Every repository in this mount's org, with the currently-active one marked.
`get <repository-id>`	—	Full settings for one repository by id (see `list` for ids).
`rename [name] [--slug]`	—	Rename this mount's repository and/or its slug. The old slug is reserved as an alias, so old `swarmfile://` URLs keep resolving (a keyed remote should use ids).
`archive` / `unarchive`	—	Soft-archive this mount's repository, or reverse it.
`delete [--yes]`	`-y, --yes`	Permanently delete this mount's repository and all its files, for everyone in the org. A hub-side soft delete: the repository leaves active use immediately and its files, history, git-LFS objects, and stored data are permanently purged 30 days later. Recovery isn't self-serve — contact support before the purge window closes to discuss it (it isn't guaranteed; the 30-day window is fixed, and a shorter per-user trash setting does not shorten it). Prints a warning and asks for confirmation first, unless `--yes` ; refuses outright in `--json` mode without `--yes` rather than reading stdin. Consider `archive` instead if you just want to hide it.
`set-keep-full-history <true|false>`	—	Opt this repository's history out of retention GC (`true` , every past version stays revertable forever) or back into it (`false`). New repositories default per the organization's new-repository default (keep every saved version unless an owner changed it); `create --keep-full-history false` is the create-time opt-out, and this flips an existing repository either way.
`set-type <revit_worksharing|none>`	—	Mark this repository as Revit-worksharing (native OS-file-lock enforcement, hub-cross-machine), or clear it with `none` .
`set-commit-mode <mode>`	—	Lock this repository to a commit mode — a branch lock overrides this while set. `mode` is `auto-commit` , `staged` , or omitted/ `unset` to clear.
`show [repository-id]`	`--json`	Print this repository's **effective** settings and where each value came from — the CLI view of the Desktop App's repository settings. Defaults to the active repository; `--json` for scripting.
`set <key=value>…`	—	Set one or more settings in a single call. Supported keys: `commit-mode` , `keep-full-history` , `type` , `name` , `slug` , `description` , `archived` , `hub-only` (`true` / `false` / `inherit` ; `false` and `inherit` both mean "inherit the org default" — a repository can only add hub-only, never force peer-to-peer over an org hub-only policy), `allow-external-collaborators` (`true` / `false` / `inherit` ; the per-repository guest policy, where `inherit` follows the org's), `allow-public-rfi` and `allow-public-access-requests` (`true` / `false` ; the public repository's RFI and access-request toggles, on by default), `allow-public-discussions` (`true` / `false` ; the public repository's Discussions toggle, off by default — an explicit opt-in), and `allow-public-issues` (`true` / `false` ; the public repository's Issues toggle, off by default — the same explicit opt-in), `contribution-policy` (`off` / `proposals` / `open` ; who may send a cross-repository fork proposal), and `conflict-mode` (`soft` / `hard` ; how a save that lost a concurrent-edit race is reported — `hard` makes the saving call itself fail with a stale-version error for automated writers, and `default` clears back to soft). Refuses an unknown key rather than silently ignoring it.
`rotate-key`	—	Rotate **this mount's E2E repository key**: mint a new generation and re-wrap it to every current device plus the org recovery key. Owner/admin and E2E only. Content written before the rotation stays sealed under the old generation and is **not** re-encrypted, so a device that already unwrapped the old key can still read it.

`rename` / `archive` / `unarchive` / `delete` / `set` / `set-keep-full-history` / `set-type` / `set-commit-mode` / `rotate-key` are owner-or-org-owner-gated: exit 2 on `forbidden` or `not_found` , exit 1 for anything else. `rotate-key` also exits 2 on a 409 (`stale_generation` — another device rotated first — or a recipient set that kept changing).

```
swarmfile repository create "Tower Block A"
swarmfile repository create "Secret Repository" --e2e
swarmfile repository create "Public Dataset" --public
swarmfile repository create "Lean Repository" --keep-full-history false
swarmfile repository create "Model" --repository-type revit_worksharing --description "Central model"
swarmfile repository create "Repo" --git-compatible --acl-mode protected --protect-default-branch
swarmfile repository create "EU Dataset" --residency eu --option hub-only=true
swarmfile repository create "Repo" --git-compatible
swarmfile repository show
swarmfile repository set commit-mode=staged keep-full-history=false hub-only=inherit
swarmfile repository rename "Tower Block A — Phase 2"
swarmfile repository set-keep-full-history true
swarmfile repository rotate-key
```

`config`

YOUR personal default commit mode for this mount's repository — advisory only. Distinct from `repository set-commit-mode` / `branch set-commit-mode` above, which are admin **locks**: those, when set, always win over this personal preference. Also distinct from `changeset-idle` / `attr-cache` /etc. above, which are local, per-machine engine-behavior knobs, not a hub-side preference tied to your account.

Subcommand	Flags	Description
`set-commit-mode [mode]`	—	Set (or, omitted/`unset`, clear) your personal default. `mode` is `auto-commit` or `staged`. Refused (exit 2, `policy_enforced`) if an admin lock already resolves for this repository+branch — a preference is never silently set unable to take effect.
`get-commit-mode`	—	Shows your personal preference (`pref`, null when unset) and the **effective** mode with its source: the response's `effectiveMode` is what this mount's next save or `changelist open` will use, and `effectiveSource` is `branch` or `repository` for an admin lock (a lock always wins), `user_pref` when your own `set-commit-mode` default applies, or `unenforced` when nothing is set.

```
swarmfile config set-commit-mode staged
swarmfile config get-commit-mode
swarmfile config set-commit-mode unset
```

`runner`

Read-only run history for the headless runner-as-CI daemon — see Runner (Headless CI) for the full walkthrough. Jobs execute unsandboxed as the daemon's own local user (see Security). There is no `runner create` / `runner edit` here: rules live in `.swarmfile/runner.yml` on the watched branch, not on the hub, so there's nothing on the hub side for this command to create or edit.

Subcommand	Flags	Description
`runs`	`--branch <name>`	Run history — every branch by default, or one branch with `--branch`. Each row carries the matched rule's name, the triggering commit, status (`running` / `success` / `failure` / `timed_out`), exit code, and a truncated log.

```
swarmfile runner runs
swarmfile runner runs --branch main
```

`ci`

Hosted CI (`.swarmfile/ci.yml` workflows that run on Swarmfile's own containers — see the Hosted CI guide). Requires the engine to be signed in: reads, `settings` / `size list`, `run` and `cancel` need a signed-in member; enable/disable, secrets, variables, size and image **writes** are owner/admin on the hub, and hosted CI stays off for a repository until an owner enables it. In some organizations hosted CI is unavailable entirely, in which case these commands answer `404 ci_not_available`. The workflow dialect is documented in the Hosted CI guide.

Subcommand	Flags	Description
`list`	`--branch <name>`, `--limit <n>`	Run history, newest first — every branch by default. Rows carry the workflow name, run number, commit, status and a blocked reason when admission refused.
`status`	`<run-id>`	One run plus its jobs: status, size, exit code, billed seconds, metered charge and the declared dependency-cache hit/miss.
`cancel`	`<run-id>`	Ask the hub to cancel a queued/running run.
`rerun`	`<run-id>`	Re-run a finished run as a NEW run at the same branch, commit and changeset — every job starts fresh. The hub re-runs the trigger gates, so an E2E repository or an unprotected branch without the opt-in refuses exactly like the original trigger (with the same typed code).
`retry`	`<job-id>`	Retry ONE failed job in place (owner/admin). The job's attempt count bumps, its per-attempt fields reset, and the run reopens and re-finalizes when the job finishes. `success` / `running` jobs cannot be retried.
`run`	`--branch <name>`, `--input NAME=VALUE`	Manually dispatch the branch head's workflow (`workflow_dispatch`). `--input` is repeatable for workflows declaring `on.manual.inputs` ; the hub coerces values to the declared type.
`approve`	`<job-id>`, `--deny`	Approve (or `--deny`) a job waiting on an environment. A denial fails the job `environment_denied` and settles the run. Eligibility is re-checked at decision time: the caller must be an eligible reviewer, cannot be the run's starter when the environment forbids self-approval, and cannot decide before the environment's wait timer elapses (exit 1 with `ci_environment_wait_not_elapsed` and the retry delay). Waiting time is not billed.
`logs`	`<job-id>`, `--offset <bytes>`, `--follow`	One job's log chunk on stdout (pipe-friendly); `--offset` continues where the last chunk ended. `--follow` keeps reading from the offset cursor (~2 s apart) and stops once the job is terminal and the tail is drained; Ctrl-C stops cleanly.
`fork-status`	`<mr-number>`	F4 — a contribution (fork-originated) merge request's live CI approval state: the bound source head, the live approval (id + approver) and the hub's own `canApprove` / `reason` verdict.
`fork-approve`	`<mr-number>`, `--head <sha>`	F4 — bind an owner/admin approval to the MR's EXACT 64-hex source head for contribution CI. A later commit revokes it (a stale `--head` is `409 head_changed`); the hub refuses a non-maintainer and a non-fork MR. Fork runs stay off until the deployment enables `CI_FORK_RUNS` .
`fork-revoke`	`<mr-number> <approval-id>`	F4 — revoke one fork-CI approval.
`settings`	—	The repository's hosted-CI settings, including the monthly budget.
`budget`	`[<usd>|off]`	Show the repository's monthly budget, or set it in dollars (owner/admin). `off` clears it; the hub enforces the $100,000 ceiling and refuses admission with `ci_budget_exceeded` . The repository comes from the current mount.
`enable`	`--allow-unprotected`	Enable hosted CI for the repository. `--allow-unprotected` is the loud opt-in for branches without effective protection (workflows can reference secrets, so leave it off unless you mean it).
`disable`	—	Disable hosted CI for the repository.
`environment list`	`--repository`	Environment definitions: the org's by default, this repository's own with `--repository` . Rows show the scope, URL, reviewers, self-approval rule, wait timer and allowed branches.
`environment set <name>`	`--repository`, `--url <url>`, `--reviewer <user:ID|role:ROLE>` (repeatable), `--starter-cannot-self-approve`, `--wait-seconds <0-43200>`, `--allow-branch <pattern>` (repeatable)	Create or replace one definition (owner/admin). The whole definition is replaced, so an omitted reviewer or branch is cleared. `--reviewer` takes `user:<idp-user-id>` or `role:<owner|admin|member>` (rank-inclusive); reviewer users must be org members. `--allow-branch` patterns are gitignore-style (`main` , `release/*` , a leading `!` negation). A repository definition composes **add-only** with the org definition of the same name: its reviewers join the org's, the longer wait wins, and a branch must satisfy both policies — a workflow `environment:` never loses an org protection.
`environment delete <name>`	`--repository`	Delete one definition (owner/admin). Deleting an org definition also deletes its environment-scoped secrets and variables; a repository deletion falls back to the org definition if one exists.
`environment secret list`	—	Environment-scoped secret names and metadata for `--environment <name>` (values stay write-only). Add `--repository` to target this repository's environment store; the environment must be one of this repository's definitions. Environment secrets are defined on the org environment a job's `environment:` resolves; a repository-only environment has none of its own.
`environment secret set <name>`	`--from-env <VAR>`, `--scope <ci|session|both>`	Store an environment-scoped secret for the `--environment <name>` definition (hidden prompt, `--from-env` , or piped stdin), same write-only contract as `secret set` (org/environment secrets are CI-only: a non- `ci` `--scope` is refused). `--repository` targets this repository's environment store; a repository-only value resolves only in this repository and that environment and wins over the org environment value of the same name.
`environment secret delete <name>`	—	Remove an environment-scoped secret (the environment is named with `--environment <name>` ; add `--repository` for this repository's store).
`environment variable list`	—	Environment-scoped variables (plaintext) for `--environment <name>` . Add `--repository` to target this repository's environment store.
`environment variable set <name> <value>`	—	Create or update an environment-scoped variable for `--environment <name>` (add `--repository` for this repository's store).
`environment variable delete <name>`	—	Remove an environment-scoped variable (the environment is named with `--environment <name>` ; add `--repository` for this repository's store).
`secret list`	—	Secret names and metadata only — values are write-only and never printed.
`secret set`	`<name>`, `--from-env <VAR>`, `--scope <ci|session|both>`	Store a secret value: a hidden prompt when run in a terminal, `--from-env <VAR>` to read it from that environment variable, or piped stdin for scripts. The value never echoes or lands in shell history. `--scope` selects the data class

Subcommand	Flags	Description
		for a repository secret (new secrets default to `ci`; an update keeps the secret's existing class unless `--scope` is given); `session` / `both` are rendered into agent sessions and redacted from their output — a value under 4 characters is not redacted.
`secret delete`	`<name>`	Remove a secret.
`variable list`	—	Names and values of the non-secret config.
`variable set`	`<name> <value>`	Create or update a variable.
`variable delete`	`<name>`	Remove a variable.
`size list`	—	Org-defined custom runner sizes.
`size set`	`<name>`, `--vcpu`, `--memory`, `--disk`	Create or update a custom size (quarter-vCPU steps; the platform enforces the ceilings).
`size delete`	`<name>`	Remove a custom size.
`image list`	—	The organization's (bring-your-own) runner images plus the curated catalog.
`image set`	`<name> <ref>`	Register or re-point a digest-pinned image under `swarmfile-runner/<org-id>/` (Pro and above to run); prepares it automatically.
`image prepare <name>`	`-f, --force`	Build/retry the runtime snapshot artifact; `pending` means run it again later. `-f` / `--force` re-verifies a row already marked ready (the artifact still exists and matches this digest). The CLI runs it automatically after `image set`; use it to retry a failed preparation.
`image import <name> <sourceRef>`	`--wait`	Import a digest-pinned external image from a public or private registry (private-source credentials come from `SWARMFILE_SOURCE_USERNAME` / `SWARMFILE_SOURCE_PASSWORD`, never argv); the source is copied, the runner layer appended, and the resolved reference registered. `--wait` polls to ready/failed.
`image import-status <id>`	`--wait`	Poll an import started by `image import`.
`image build <name>`	`--context <dir>`, `--dockerfile <path>`, `--target <stage>`, `--build-arg KEY=VALUE`, `--secret NAME`, `--cache`, `--source-commit <sha>`, `--allow-dirty`, `--no-prepare`, `--wait`	Build an image from a Dockerfile through the hosted builder (Pro and above; enabled where the environment sets `CI_IMAGE_BUILD=on` — staging and production — otherwise `ci_build_disabled`). The context (default `.`) is packed as a gzipped tar with `.git` excluded (64 MiB compressed cap) and pinned to the checkout's HEAD; a dirty tree is refused unless `--allow-dirty`, and a secret-shaped build-arg key is refused. `--secret NAME` (repeatable; operator-gated, off by default) exposes an organization CI secret as a BuildKit secret (`RUN --mount=type=secret,id=NAME`); its value is resolved server-side and never returned or logged. `--cache` is refused (`ci_build_cache_unavailable`) in the trusted-importer build shape; no `CI_BUILD_CACHE` switch is read. The image is scanned before pushing, then the runner layer is appended and the image registered through the import path; `--wait` follows it and prepares it unless `--no-prepare`.
`image build-status <id>`	`--wait`	Poll one build (status, digest, scan summary, error).
`image build-logs <id>`	—	Print a build's log tail (the row keeps at most 16 KiB).
`image build-cancel <id>`	—	Cancel a running build; the builder deletes the staging tag on any non-ready terminal state, and cancel is refused once the image import has started.
`image scan <name>`	`--wait`	Start a vulnerability scan of an imported image. The command prints the scan id and status; `--wait` polls until the scan is `passed`, `blocked` or `failed` (blocked and failed exit 1) for up to the hub's 35-minute scan budget. If it is still running then, the command exits 1 rather than reporting success. With `--json --wait` it emits the terminal scan-status body (findings, waivers, current/effective verdict), not the POST start row.
`image scan-status <name>`	—	The latest scan: status, critical/high/fixable/blocking counts, every finding (CVE, severity, package, installed and fixed versions), the scanner error when one was recorded, and waivers for that scan's digest split into active and expired (revoked and other-digest waivers are omitted). A running scan reports findings as pending and a failed scan reports them unavailable rather than "none".
`image waive <name> <cve>`	`--reason <text>`, `--days <n>`	Grant (or refresh) a waiver for one CVE in the latest scan. `--reason` is required (3-500 characters); `--days` accepts 1-30 and defaults to 7.
`image unwaive <name> <cve>`	—	Revoke an active CVE waiver.
`image delete`	`<name>`	Remove an org image; jobs selecting it fail `ci_image_unknown` until it is registered again.
`runner list`	—	The organization's live self-hosted runners: name, labels, project allowlist, last seen and managed/E2E caps.
`runner create <name>`	`--label <label>` (repeatable), `--repository <id>` (repeatable), `--allow-managed`, `--allow-e2e`	Enroll one runner (owner/admin). Prints the one-time `sf_shr_` enrollment token; only its hash is stored, so copy it into the installer now.
`runner delete <id>`	—	Revoke a runner; in-flight leases and the enrolled E2E device key are cut.
`runner token <name>`	`--label`, `--repository`, `--max-runners <n>`, `--ttl-hours <n>`, `--allow-managed`, `--allow-e2e`	Mint a mint-only `sf_srt_` registration token for an autoscaled fleet. The token is printed once; it cannot be shown again.
`runner tokens`	—	Registration-token metadata (not the values).
`runner revoke-token`	—	Revoke a registration token and every ephemeral runner it minted.

Subcommand	Flags	Description
`<id>`		
`runner queue`	—	This repository's per-label self-hosted demand: queued jobs waiting for a runner and running jobs that hold one (owner/admin). The reference autoscaler polls the same route across the org's repositories.
`provenance <file>`	`--pin-key <jwk>`	Verify a hub-signed provenance/attestation JSON offline; `--pin-key` pins the signer key instead of trusting the statement's own key. No engine or hub is needed.
`oidc`	`--audience <aud>`, `--aws-role <arn>`, `--gcp <provider>`, `--azure`, `--azure-client-id <id>`, `--azure-tenant-id <id>`, `--export-env <file>`	Mint a short-lived OIDC token for a cloud provider. Only available INSIDE a hosted or self-hosted CI job whose workflow declared `write: [oidc]`; the job token is the credential, so there is no engine sign-in. The audience must exactly match one the organization's owner/admin allowed (an empty allowlist disables OIDC). The token is short-lived (10 minutes) and the command refuses after 10 mints for one job. `--aws-role` writes the AWS web-identity env, `--gcp <provider-resource>` writes a Google external-account credential file, and `--azure` (with `--azure-client-id` / `--azure-tenant-id`) writes the Azure federated-credential env; pick one provider per run. Without `--export-env` the command writes the provider env to `.swarmfile-oidc-token` (mode 0600).
`metrics`	—	Queue/cache SLO inputs for this repository (owner/admin): queued/running counts, waiting reasons, oldest queued, 24h terminal counts, started-job queue-time avg/p95/max, and cache declared/hit/miss/rate.

```
swarmfile ci list --branch main
swarmfile ci status 018f3c2e-…
swarmfile ci logs 018f3c2e-… --offset 65536 | less
swarmfile ci logs 018f3c2e-… --follow          # live until the job ends
swarmfile ci rerun 018f3c2e-…                  # clone a finished run
swarmfile ci retry 018f3c2e-…                  # re-queue one failed job
swarmfile ci approve 018f3c2e-…                # approve a waiting environment job
swarmfile ci approve 018f3c2e-… --deny         # deny it instead
swarmfile ci fork-status 12                    # a contribution MR's CI approval
swarmfile ci fork-approve 12 --head <sha>      # approve that exact source head
swarmfile ci fork-revoke 12 <approval-id>      # revoke it
swarmfile ci run --branch main
swarmfile ci environment list                  # the org's definitions
swarmfile ci environment list --repository        # this repository's own
swarmfile ci environment set production --reviewer role:admin --wait-seconds 600 --allow-branch main
swarmfile ci environment set staging --repository --reviewer user:idp_9f2a --allow-branch 'release/*'
swarmfile ci environment secret set DEPLOY_KEY --environment production   # org env
swarmfile ci environment secret set DEPLOY_KEY --environment production --repository  # repository env
swarmfile ci variable set REGION eu
swarmfile ci secret set NPM_TOKEN            # hidden prompt for the value
swarmfile ci secret set NPM_TOKEN --from-env NPM_TOKEN   # read it from the environment
swarmfile ci variable set REGION eu
swarmfile ci size set big --vcpu 4 --memory 12 --disk 20
swarmfile ci image list
swarmfile ci image set our-rust "$BYO_IMAGE_REF"   # the digest-pinned ref from the Hosted CI guide
swarmfile ci image prepare our-rust
swarmfile ci image import our-rust docker.io/library/python@sha256:<64 hex> --wait
# Or build it yourself from a Dockerfile (Pro and above; a clean checkout):
swarmfile ci image build our-rust --context . --dockerfile docker/Dockerfile --wait
swarmfile ci image scan our-rust --wait
swarmfile ci image scan-status our-rust
swarmfile ci image waive our-rust CVE-2024-1234 --reason "not reachable from a workflow" --days 7
swarmfile ci image unwaive our-rust CVE-2024-1234
swarmfile ci runner list
swarmfile ci runner create linux-builder --label linux --label x64 --repository <project-id>
swarmfile ci runner token fleet --label linux --repository <project-id> --max-runners 20 --ttl-hours 24
swarmfile ci runner tokens
swarmfile ci runner revoke-token <token-id>
swarmfile ci runner delete <runner-id>
swarmfile ci runner queue              # this repository's per-label demand
swarmfile ci oidc --audience sts.amazonaws.com   # inside a `write: [oidc]` job
swarmfile ci oidc --audience sts.amazonaws.com --aws-role arn:aws:iam::123456789012:role/ci --export-env "$GITHUB_ENV"
swarmfile ci oidc --audience <gcp-pool-audience> --gcp projects/p/locations/global/workloadIdentityPools/pool/providers/gh --export-env
"$GITHUB_ENV"
swarmfile ci oidc --audience api://AzureADTokenExchange --azure --azure-client-id <app-id> --azure-tenant-id <tenant-id> --export-env
"$GITHUB_ENV"
swarmfile ci provenance attestation.json --pin-key signer.jwk
swarmfile ci metrics                   # queue/cache SLO inputs
```

`billing`

Prepaid compute credit for hosted-CI overage (owner-only). A pack is bought once through Stripe Checkout; unused credit expires after 12 months and never raises the spend cap.

Subcommand	Flags	Description
`packs`	`--buy`	The current mount's org's compute-credit balance, recent packs and their remaining credit. `--buy` starts a one-time Stripe Checkout and prints the URL.

```
swarmfile billing packs
swarmfile billing packs --buy
```

`office`

Peer-discovery office grouping — applies immediately, no restart.

Subcommand	Flags	Description
`status`	—	This mount's current `office_id` and whether `lan_from_office` is on.
`set`	`<office_id>`, `--lan-from-office`	Set the office-grouping label this mount announces for peer discovery. Pair with `--lan-from-office` on a network that blocks mDNS multicast — see `ec-placement` above.

> Seed nodes are the one exception to `--lan-from-office`: a seed node joins another peer's placement roster purely by matching `office_id`, with no flag and no shared network required at all — see Self-Hosted Seed Nodes for using this to give a cloud-hosted render farm deliberate redundancy among its own nodes.

`acl`

Folder-ACL repository mode and the ACL change log — the headless equivalent of the dashboard's ACL settings panel.

Subcommand	Flags	Description
`mode get`	—	Whether this mount's repository is `open` (default-readable) or `protected` (default-denied, entries need an explicit grant).
`mode set <open|protected>`	—	Switch repository ACL mode. Admin-or-owner-gated: exit 2 on `forbidden` / `not_found`. Switching **to** `protected` additionally needs the org's plan to include folder ACLs — exit 2 with `plan_restricted` (402) if not.
`audit [--entry-id] [--limit] [--offset]`	—	The ACL change log — every grant/revoke, repository- or entry-scoped.
`entry get <entry-id>`	—	The ACL roster on one entry (direct + inherited rows). `<entry-id>` accepts a mount path too (resolved first). Admin-gated: 403s for a member who is neither an org admin nor an entry admin (exit 2), since the roster reveals who else has access.
`entry set <entry-id>`	`--principal <id>`, `--permission <read|comment|comment_upload|write|admin>`, `--effect <allow|deny>` (default `allow`), `--inherit`	Add or update a grant. Creating one needs the org's plan to include folder ACLs — exit 2 with `plan_restricted` (402) on Starter.
`entry remove <entry-id> <principal-id>`	—	Revoke a principal's grant on an entry.
`principals [--query] [--limit] [--offset]`	—	Look up an ACL `principal id` — the value `entry set --principal` and `mr assign` take. `--query` searches display name/email case-insensitively; without it, lists the org's principals. One page is capped at 50 rows; `--offset` pages past it, for a search as well as the full list.

```
swarmfile acl mode get
swarmfile acl mode set protected
swarmfile acl audit --entry-id 3f9a2b --limit 50
swarmfile acl principals --query "Priya"
swarmfile acl entry get 3f9a2b
swarmfile acl entry set 3f9a2b --principal u_9f2a --permission write --inherit
swarmfile acl entry remove 3f9a2b u_9f2a
```

`acl entry` is the per-entry grant surface the repository-wide `mode` toggle doesn't cover; the same grants are editable from the dashboard's permissions panel, and (on Windows) projected as NTFS DACLs on the mount.

`e2e-key`

End-to-end encryption device keys.

Subcommand	Description
`status`	This device's own enrolled key — id and public key — read straight from the local cache file, no hub round trip. `enrolled: false` means this device hasn't finished enrolling yet (or the binary was built without the `e2e` feature at all, flagged with `e2eBuild: false`).
`list`	Every device key on this account — every machine, not just this one. For finding the id of a lost device to revoke.
`revoke <id>`	Revoke a device key by id (see `list`) — e.g. after a lost laptop.

There's no `enroll` here — a device enrolls its own key automatically at startup.

```
swarmfile e2e-key status
swarmfile e2e-key list
swarmfile e2e-key revoke key_9f2a
```

`device-transfer`

Move a device's end-to-end repository keys onto a second device without going through the org's recovery-key quorum — a one-time, ten-minute code links the new device to an already-enrolled one and moves the wrapped repository keys across, entirely client-side. The private key never touches the hub.

Subcommand	Description
`start`	Run on the device that already has the repository's key. Prints a one-time code, waits for a new device to link, then uploads this mount's currently-mounted repository key to it automatically. Requires an enrolled E2E key and a mounted E2E repository on this engine.
`link <code> [--label]`	Run on the new device with the code from `start`. Waits for the other device to upload, then finalizes — re-wrapping every recovered repository key to this device's own already-enrolled key. `--label` names the device (shown later in `e2e-key list` and **Settings → My Devices**).

Both commands poll every couple of seconds until the handshake completes or the code expires; past 10 minutes either side reports a clear "expired, run `start` again" rather than hanging.

```
# On the device that already has the repository:
swarmfile device-transfer start
# Transfer code: 7F3K-9QRT
# On the new device, within 10 minutes, run:
#     swarmfile device-transfer link 7F3K-9QRT

# On the new device:
swarmfile device-transfer link 7F3K-9QRT --label "Jordan's laptop"
```

`start` transfers only the mount's currently-configured repository — run it again, once per mounted repository, on a device that holds more than one. There's no `--enroll` step on the new device: every engine enrolls its own long-term device key automatically at startup, and `link` / `finalize` re-wrap to that key directly.

If nobody's engine is online to complete an ordinary access grant, an org member can also fulfill it manually from the web dashboard rather than running this at all — see End-to-End Encryption Setup.

`admin nodes`

The hub's P2P mesh node allow-list — owner-only. `node_id` is the 64-hex-char Ed25519 public key an engine self-generates (visible in the engine's own startup log, or in a not-yet-authorized node's connection-refused message). The same list, with an authorize form and a revoke button, is also available from **Settings → Nodes** in the web dashboard.

Subcommand	Flags	Description
`list`	—	Every node currently on the org's allow-list.
`authorize <node-id> <user-id> [--label]`	—	Add a node, authenticating as `user-id`.
`revoke <node-id>`	—	Remove a node — it can no longer join the mesh.

```
swarmfile admin nodes list
swarmfile admin nodes authorize a1b2c3...64hex user_88 --label "render-farm-3"
```

`admin retention`

The organization's trash/history retention window and its **size-adaptive floor**, plus the history-retention default for NEW repositories — owner-only. When a repository approaches its storage limit, the hub automatically shrinks the retention window under pressure; the floor is the tightest it may ever shrink to. Both are `null` (the global defaults) until set.

`keepFullHistoryDefault` decides what newly created repositories start with: `true` (the product default) keeps every saved version, `false` starts them on named commits only — a creator's explicit choice at creation still wins. The same controls are in **Settings → Organization** in the web dashboard.

Subcommand	Flags	Description
`get`	—	The current window, floor, and new-repository retention default.
`set`	`--days <1-3650>`, `--floor <1-3650>`, `--clear-days`, `--clear-floor`, `--keep-full-history-default <true|false>`	Set and/or clear the day values, and/or set the new-repository default. Omitted fields are left unchanged. The hub rejects a floor above the window (it would be inert).

```
swarmfile admin retention get
swarmfile admin retention set --days 120 --floor 30
swarmfile admin retention set --clear-floor
swarmfile admin retention set --keep-full-history-default false
```

`admin rate-limits`

The organization's **request budgets**: the per-org and per-user request ceilings, in requests per minute — owner/admin-configurable. The same limits appear in the tray's **Settings → Developer** panel and the web dashboard's **Settings → Organization** page. `show` reports the current values alongside each plan default and maximum; `set` changes them. The Free plan's limits are fixed, Starter/Pro cap at their tier maxima, and Enterprise's org ceiling can be removed entirely. A PAT shares its owner's per-user window; an API key gets its own window at the plan's default per-user value; content streaming isn't counted here.

Subcommand	Flags	Description
`show`	—	The current per-org and per-user ceilings, their plan defaults and maxima, and whether each is configurable.
`usage`	—	This minute's usage against the ceilings, for the whole org and for you (the caller), including the separate media-read windows.
`set`	`--org <n>`, `--user <n>`, `--clear-org`, `--clear-user`, `--unlimited-org`, `--for <dur>`	Set and/or clear either ceiling. Omitted fields are left unchanged. `--clear-*` resets a field to the plan default; `--unlimited-org` removes the org ceiling entirely (Enterprise only); `--for 6h` (units `s` / `m` / `h` / `d`, max 30 days) makes a raise auto-revert. The hub rejects a value above the plan's max and refuses any change on the Free plan.

`rate-limits` is also available at the top level (`swarmfile rate-limits …`) — the same subcommands, without the `admin` prefix.

```
swarmfile admin rate-limits show
swarmfile admin rate-limits usage
swarmfile admin rate-limits set --org 4000 --user 450
swarmfile admin rate-limits set --org 8000 --for 6h
swarmfile rate-limits set --clear-org
swarmfile rate-limits set --unlimited-org
```

`notifications`

This account's notification inbox — comments, mentions, unlock requests, merge requests, RFIs, quarantine, failed uploads, and more. `list` / `unread-count` / `read` / `read-all` hit the **hub's full notification feed** (the same inbox the dashboard's bell shows), not just this machine's own upload failures.

Subcommand	Flags	Description
`list`	`--kind`, `--unread`, `--since <time>`, `--limit`	Notification history, newest first. `--since` accepts Unix seconds, epoch milliseconds, a duration (`2h`, `7d`), or an RFC 3339 timestamp. `--kind` accepts a hub notification kind (the same open vocabulary — comments, mentions, unlocks, RFIs, MRs, quarantine, uploads, storage, …); an unknown value is rejected with the accepted list and exit 2, so a typo can't silently return the whole inbox.
`unread-count`	—	Cheap count for a shell prompt or a CI/cron check.
`read <id>`	—	Mark one notification read.
`read-all [--kind]`	—	Mark every notification read, optionally narrowed to one kind. An unknown `--kind` is rejected (exit 2) rather than marking the whole inbox read.
`preferences get`	—	Current email/webhook delivery preferences — the webhook status (`url` / `enabled` / `configured`) rides along in this same output.
`preferences set`	`--email-enabled [<true|false>]` (bare = on), `--quiet-start` / `--quiet-end` (`HH:MM`), `--timezone`, `--kind <KIND=on|off|default>` (repeatable)	Update one or more fields — omitted flags leave the existing value untouched. `--quiet-start` / `--quiet-end` must both be set together or both left unset. Clearing quiet hours or the timezone is dashboard-only: the API takes an explicit null for it, but the CLI has no null form (an empty value is a validation refusal), so there is no CLI clear. `--kind` sets a per-kind email override (`off` mutes that event, `on` forces it, `default` clears that kind's whole override — email and webhook — back to the kind's defaults: the global switches plus that kind's own default, where one applies); the CLI merges into your current overrides, so other kinds and their webhook flags are preserved. An unknown kind or malformed spec exits 2. Exit 2 on a `400` (mismatched pair or malformed time).
`preferences webhook set <url>`	`--enabled [<true|false>]` (bare = on)	Set (or change) YOUR OWN personal webhook — a third notification channel next to the in-app bell and email, one URL per user. The `<url>` argument is required on every call (no partial "just flip `--enabled`" shorthand — pass the same URL again to toggle it alone). Mints a fresh signing secret whenever the URL changes, or is set for the first time, and prints it ONCE. Distinct from the org-wide `swarmfile webhook` command group below: this one is fire-and-forget, with no delivery log or auto-disable.
`preferences webhook clear`	—	Remove your personal webhook.
`preferences webhook rotate-secret`	—	Mint a fresh signing secret without changing the URL. Hard cutover — the old secret stops verifying immediately. Prints the new secret ONCE.
`preferences webhook test`	—	Fire one synthetic event at your configured webhook right now, synchronously. Nothing is logged — this channel has no delivery log — the result prints directly.

```
swarmfile notifications list --unread --limit 20
swarmfile notifications list --kind comment_mention
swarmfile notifications read-all
swarmfile notifications preferences set --quiet-start 22:00 --quiet-end 07:00 --timezone America/New_York
swarmfile notifications preferences set --kind file_changed_by_other=off
swarmfile notifications preferences set --kind comment_added=default   # clears email + webhook for that kind
```

`share`

Share links — a read/comment/respond link to one file, handed to someone outside the org. This covers only the member-authed management side: creating your own links, listing/revoking them, and (owner-only) the access log and email block-list. The anonymous recipient's own side of a link is the web-hosted `/s/:token` page, not a CLI surface.

Subcommand	Flags	Description
`create <entry-id>`	`--expires-at <unix-secs>`, `--max-accesses <N>`, `--password <string>`, `--allow-comments`, `--allow-responses`	Create a link for one file. `<entry-id>` accepts a mount path too (the path is resolved first). Exactly one of `--allow-comments` / `--allow-responses` is meaningful — the hub picks the more permissive one if both are set. `--allow-responses` is the RFI share-link path (needs a directory entry, not a file) — see RFI attachments. Exit 2 on a `400` (missing/invalid `entry-id`, or wrong entry type for the requested permission) or a `404` (entry gone).
`list`	`--entry-id`	The caller's own share links, optionally narrowed to one file.
`revoke <id>`	—	Revoke a link. Exit 2 on `404` (not found, or not yours).
`access-log <id>`	`--limit`, `--offset`	Who has actually opened this link (the mandatory-email-gating log, not just a hit count). Owner (creator) only — exit 2 on `404`.
`block <id> <email>`	—	Block one email from this link going forward. Owner only. Takes effect on that email's next gated request — an already-issued 30-day session cookie isn't separately revoked.
`unblock <id> <email>`	—	Unblock a blocked email address. Owner only.

```
swarmfile share create e-1a2b3c --allow-comments --expires-at $(( $(date +%s) + 7*86400 ))   # one week from now
swarmfile share list --entry-id e-1a2b3c
swarmfile share access-log shr_9f2a --limit 20
swarmfile share block shr_9f2a spammer@example.com
swarmfile share revoke shr_9f2a
```

`activity`

The org/repository activity feed — commits, branch switches, comments, CI runs, membership/ACL changes, who did what and when. An org owner sees everything; everyone else only what they have read access to (owner-only sources like quarantine and org-policy events repository to empty for them).

Subcommand	Flags	Description
`list`	`--since` / `--until <time>`, `--limit`, `--kind <csv>`, `--repository-id`, `--entry-id`, `--actor-id`, `--cursor`	The feed, newest-filtered-window first. Defaults to the last 7 days when `--since` / `--until` are omitted. `--since` / `--until` accept Unix seconds, epoch milliseconds, a duration (`2h`, `7d`), or an RFC 3339 timestamp. `--kind` takes a comma-separated list of event kinds. `--cursor` resumes from a previous response's pagination cursor.
`export`	`--since` / `--until`, `--kind`, `--repository-id`, `--out <path>`	Unbounded CSV export (capped at 100k rows hub-side) — **org-owner-only**, and needs the Pro+ `audit_log` entitlement (exit 2 on the `403` either gate returns). Prints CSV to stdout by default; `--out` writes it to a file instead. Not paginated — narrow `--since` / `--until` if you need a complete window (a response is capped at 100k rows).
`tail`	`--interval <secs>` (default 3), `--since <time>`, `--kind`, `--repository-id`, `--actor-id`	Follow the feed: print new events as they land, until Ctrl+C. Starts from now unless `--since` is given (same time shapes as `list`).

`tail` follows by polling the same feed `list` reads — polling is the supported way to follow the feed. Expect a few seconds of latency, bounded by `--interval`. With `--json` it prints one event per line (JSON-lines), so a consumer can process the stream incrementally instead of re-parsing a document that repeats each poll.

```
swarmfile activity list --kind mr_merged,commit_created --limit 50
swarmfile activity list --since 30d --repository-id p_9f2a
swarmfile activity tail --kind commit_created
swarmfile activity export --since 1767225600 --until 1798675200 --out audit-2026.csv
```

`webhook`

Manages this **org's** outbound webhook subscriptions — signed HTTP callbacks fired on file/branch/merge/ACL/membership events, into Slack, Zapier, a SIEM, or your own endpoint. Deliveries are dispatched when the event is recorded, typically within seconds; the hourly tick is a backstop. Everything the web dashboard's webhook settings page does is available here too; this is the org-wide subscription list, distinct from the single personal channel under `notifications preferences webhook` above.

Subcommand	Flags	Description
`list`	—	Every webhook subscription for this mount's org. Never shows signing secrets.
`create <url>`	`--event <kind>` (repeatable), `--description`	Subscribe a URL to org activity events. Prints the raw signing secret ONCE — copy it now, the hub never returns it again. `--event` may be repeated to subscribe to specific kinds only; omit it to receive every kind.
`delete <id>`	—	Remove a subscription. Immediate.
`reenable <id>`	—	Clear a subscription's auto-disable — the hub disables one automatically after too many consecutive delivery failures.
`rotate-secret <id>`	—	Mint a fresh signing secret. Hard cutover — the old secret stops verifying immediately. Prints the new secret ONCE.
`update <id>`	`--url`, `--description`, `--clear-description`	Update a subscription's URL and/or description. An omitted flag keeps its current value. Doesn't touch which event kinds it receives or its custom headers.
`deliveries <id>`	`--limit`, `--cursor`	A subscription's delivery log, newest first, cursor-paginated (use the previous response's `nextCursor`).
`test <id>`	—	Fire one synthetic event at a subscription's URL right now, synchronously. Recorded in the delivery log, but never counts toward auto-disable health tracking.
`retry-delivery <id> <delivery-id>`	—	Manually re-send one past delivery. A delivery with no stored payload can't be retried.

```
swarmfile webhook create https://hooks.example.com/swarmfile --event mr_merged --event commit_created
swarmfile webhook list
swarmfile webhook deliveries wh_9f2a --limit 20
swarmfile webhook rotate-secret wh_9f2a
```

Scripting against the CLI

Pass the global `--json` (before or after the subcommand — both work) and it prints the engine's raw response, which is the intended path for automation — commit from CI, hydrate before a render, check `conflicts` in a pre-flight script (`conflicts resolve <path> --auto` clears text conflicts headlessly, exiting 2 when a human is needed). A usage error under `--json` emits `{"code":"usage_error",…}` rather than clap's prose, so the same parser handles both. Pair it with an API key and no part of the loop needs a human.

To react to changes as they happen, `activity tail` polls the activity feed every `--interval` seconds and prints new events as they land. If your integration shouldn't run a poller at all, `swarmfile webhook create` (org owner) or `notifications preferences webhook set` (personal) both set up a signed HTTP callback to a URL you control instead.

They differ in delivery: org webhooks are dispatched when the event is recorded — typically within seconds, with the hourly tick as a backstop — and carry retries and a delivery log; the personal channel is fire-and-forget, enqueued as each notification happens, with no delivery log, retry, or auto-disable.

swarmfile-doctor

`swarmfile-doctor` diagnoses Swarmfile connectivity: DNS, hub, cloud storage, the peer-to-peer network, mDNS, and OIDC, plus `install-layout` (detects an install an update can't reach) and a `plan` check.

When an engine is running, the `plan` check reads your organization's plan standing and **fails when Swarmfile isn't signed in**, because a machine that can't read your plan generally can't reach your files either; without an engine it reports `pass` with "skipped (requires the engine)". On seed/NAS nodes it additionally verifies your plan includes the seeding entitlement. Signed in as a guest reviewer, it passes and says so: guests aren't members, so there's no plan to read.

It is ONE tool that works with or without a running engine: on its own it runs the engine-free probes (DNS, the hub's `/health`, the block path, OIDC, the server's profile descriptor, and its signed host identity when served); when a `swarmfile-engine` is running it discovers it (its cache-dir `control.json`) and fetches the engine/OS-only checks (the mount, the peer-to-peer swarm and local-network peer discovery, transfer stats, the firewall, the engine socket) over the local control socket, against whatever hub the engine is configured for (the hosted cloud or a community server).

Use it on a machine where the engine isn't up, won't start, or you're not sure which is true.

It ships on `PATH` with the .pkg/.deb/MSI installs and is also published as a standalone download for machines where installing the app isn't an option: Developer ID-signed and notarized on macOS, Authenticode-signed on Windows (SmartScreen reputation may still warn), and unsigned on Linux; see Getting Started and Security.

Figure omitted in print (animation): swarmfile-doctor reads a hub end to end — DNS, TLS, health, the key, the block path, the declared profile — then walks three breakages deployments actually ship: a revoked key, a service that isn't running, and a typo'd repository id. Every answer is a real request against a real misconfiguration. - https://swarmfile.com/demo/oss-doctor-debug.cast

Flags

Flag	Description
`--json`	Machine-readable output
`-q, --quiet`	Suppress non-error output. Implies `--json` for stdout.
`--report <path>`	Also write the JSON report to this file. No report file is written without it. With `--repair`, the report holds one `repair` check with the outcome.
`--hub-url <url>`	Override the hub URL to check. Without it, `$SWARMFILE_HUB_URL` is used, and with neither, a running engine supplies its own hub.
`--org-id <org>`	Organization id (with `--repository-id`, enables the block-path probe).
`--repository-id <id>`	Repository id (with `--org-id`, enables the block-path probe).
`--token <key>`	Bearer token (`sf_key_…` / `sf_pat_…`). Defaults to `$SWARMFILE_API_KEY`.
`--timeout <secs>`	Per-request timeout (default 10).
`--no-iroh`	Skip the peer-to-peer connectivity probe
`--no-mdns`	Skip the mDNS discovery probe
`--latency`	Also time the hub routes a push/download walks (`/health`, the org's repository list, the repository's `branches` and `changesets`) three times each and report min/avg/max. The **hub latency** check warns above 500 ms on any route; the repository-service routes are expected to be slower than `/health`.
`--iroh-peer <HOST:PORT>`	Accepted for compatibility and unused: the running engine's own peer-to-peer check runs instead
`--cache-dir <path>`	Engine cache dir. Default `~/.cache/swarmfile` (macOS/Linux), `%LOCALAPPDATA%\swarmfile` (Windows).
`-v, --verbose`	Verbose output
`--repair`	Reinstall the Swarmfile desktop app from its update channel, after verifying the download. See `--repair`.
`--yes`	Skip the confirmation prompt for `--repair` (required when there is no terminal to ask on)
`--allow-downgrade`	With `--repair`, install the update channel's version even when it is older than the installed one (macOS and Linux).

Host identity

The `host identity` check fetches `/.well-known/swarmfile.json` from the checked hub and verifies the statement's shape, its canonical `origin` (the configured public origin, not the request `Host`), its key/signature format and its freshness.

A `404` / `503` is a **pass**: the statement is optional and a community or self-hosted deployment without a public origin is healthy.

A malformed body, a different origin, or a stale/future `issuedAt` is a failure. Shape is all this check proves; use `swarmfile-verify-history --check-host-identity` for the signature and active-key binding (see Verifiable History).

Hub latency

`--latency` (and the Desktop App's **Run doctor**, which includes it) answers the "why is my push slow?" question with numbers instead of a guess. It times four routes three times each:

- `/health` — the edge the request lands on;
- `GET /orgs/<org>/repositories` — the organization's own control-plane service;
- `GET .../repositories/<id>/branches` — the repository service (the push preflight);

- `GET .../repositories/<id>/changesets?limit=1` — the repository service's commit log.

The check's detail lists each route as `avg ms (min, max)` with the HTTP status when it is not 2xx (a repository key may be refused the org list, for example), and the check **warns** when any route's worst sample exceeds 500 ms or a route cannot be reached. A healthy `health` with a slow `branches` / `changesets` points at that repository's service (or its region), not the network; a slow `health` too points at the path to the edge. For route-level detail use the hosting platform's request log for the repository and the same time window.

Exit codes

The process exit code is the machine-readable verdict, so a script or provisioning step can gate on it without parsing the report:

Code	Meaning
`0`	Every check passed.
`1`	At least one check **failed**.
`2`	At least one **warn**, and no fails.

With `--repair` the codes mean:

- `0` the desktop app was reinstalled (or upgraded; the result line says which)
- `1` a step failed (reading the update channel, the download, the verification, or the installer itself)
- `3` refused or canceled with nothing changed (no desktop install, an unsupported platform or update channel, an older version without `--allow-downgrade`, no confirmation, not running as root, the Windows administrator prompt declined, a Windows session that is neither elevated nor signed in to a desktop, more than one installed Windows product named exactly "Swarmfile", or a self-hosted install, which updates only from its bundle)

`--repair`

`--repair` reinstalls the Swarmfile desktop app on this machine from the update channel it was installed from. It is the command-line form of **Diagnostics → Reinstall...** in the Desktop App: use it when the app is too broken to open, or from a remote shell or a management tool. Your files, settings and upload queue stay in place.

1. **It finds the desktop install** from the configuration file the installer wrote, which also names the install's update channel.

   The `swarmfile-doctor` that ships inside an install repairs that install; a separately downloaded copy repairs the standard install (`/Applications/Swarmfile.app`, `%ProgramFiles%\Swarmfile`, or the `swarmfile` package on Linux). With no desktop install it refuses (exit `3`) and points you at swarmfile.com/downloads.

   It also refuses (exit `3`) on the self-hosted edition: that install is pinned to the bundle it came from, and reinstalling from an update channel would replace it with the hosted app.

2. **It checks the update channel.** Only Swarmfile's own update channels are accepted, and only the one that matches the install, so a reinstall can never move an install to a different Swarmfile environment.

3. **It picks the full installer** for this machine from the update channel: the `.pkg` on macOS, the `.msi` for the machine's architecture on Windows, the amd64 `.deb` on Debian, Ubuntu and their derivatives (where the `swarmfile` package is installed with `dpkg`).

   Other Linux distributions and non-amd64 Linux aren't supported by `--repair`; reinstall the package with your package manager.

   If the `swarmfile` package is installed but its configuration file (`/etc/swarmfile/config.json`) is missing, `--repair` can't tell which update channel the install came from: it refuses and says so, and installing the `.deb` from Downloads with `sudo apt-get install ./swarmfile_<version>_amd64.deb` restores the file.

4. **It won't move you backwards.** If the channel publishes an older version than the one installed, it refuses; `--allow-downgrade` installs it anyway on macOS and Linux. Windows never installs an older version over a newer one: uninstall first.

5. **It shows the plan and asks.** It prints the installed version, the version it will install, the download, how the download is verified and the installer command (with the exact path of the file it will run), then asks for confirmation. `--yes` skips the question; with no terminal to ask on (a script or a management tool), `--repair` needs `--yes`.

6. **It verifies before anything runs.** The download is over HTTPS and stays on the channel's own servers. The `.msi` and `.deb` must match the release signature published in the update channel, checked with that channel's own release key (the same check automatic updates apply), and the signature must be for that exact file. The `.pkg` must match its published SHA-256, carry Swarmfile's own Developer ID Installer signature, and pass Gatekeeper as notarized. A download that fails any check is never run.

7. **It runs the installer** and reports the result: `installer -pkg … -target /` on macOS; on Windows `msiexec /i … REINSTALL=ALL REINSTALLMODE=vamus` when the channel has the installed version (a full in-place reinstall, with the verified `.msi` saved under the file name Windows recorded for the install, such as `Swarmfile.msi` for a `Setup.exe` install, which an in-place reinstall requires) or `msiexec /i …` for a newer one, with `/norestart REBOOT=ReallySuppress` (it never restarts the machine; if a restart is needed to finish, it says so).

   It shows a progress bar (`/qb`) in a signed-in desktop session and runs fully silently (`/qn`) without one, such as SYSTEM from a management tool or an elevated remote shell.

   The verified `.msi` is saved in a new folder only its owner, SYSTEM and administrators can open (in the install folder for an elevated run, otherwise your temp folder), and the installer's verbose log (`msiexec.log`) goes in the same folder; after a failure the folder stays with the log and the result names its path.

   If Windows has more than one installed product named exactly "Swarmfile" (an old install left beside the current one), an in-place reinstall can't tell which to use: it refuses (exit `3`) and asks you to uninstall the extra one in Settings > Apps. `dpkg -i …` on Linux. If `dpkg` reports missing dependencies, `sudo apt-get -f install` finishes the install.

Administrator rights

The installer needs administrator rights. On macOS and Linux, run it as root (`sudo swarmfile-doctor --repair`); otherwise it shows the plan and stops.

Because it then decides what to install as root, it refuses when the installed app's configuration isn't owned by root or is writable by other users. A `.pkg` or `.deb` install is root-owned; an app copied from the `.dmg` belongs to the user who copied it, so on that machine download the `.pkg` from Downloads and run it instead (or make the app root-owned first, `sudo chown -R root:wheel /Applications/Swarmfile.app`). On Linux the refusal names the fix for the configuration file and its folder (`sudo chown root:root /etc/swarmfile /etc/swarmfile/config.json`, then `chmod 755` the folder and `chmod 644` the file).

On Windows, run it from an administrator prompt or approve the Windows prompt (an administrator on a machine with UAC turned off counts as elevated); a run that is neither elevated nor in a signed-in desktop session (a scheduled task or a remote shell as a standard user) refuses with exit `3`, because no one could approve the prompt.

See Operations for where a reinstall fits into a broader recovery workflow.

`--repair` prints its plan and result as text: `--json` and `--quiet` don't apply to it. Use `--report <path>` for a machine-readable result.

Examples

Plain diagnostic run:

```
swarmfile-doctor
```

JSON output suitable for attaching to a support ticket:

```
swarmfile-doctor --json --report doctor-report.json
```

Reinstall the desktop app without a prompt (macOS or Linux):

```
sudo swarmfile-doctor --repair --yes
```

swarmfile-migrate

`swarmfile-migrate` bulk-imports data you already have into a Swarmfile repository: a local directory, a flat list of file paths, or an S3-compatible bucket. It's a standalone binary — no running engine required — meant for the first-time import of an existing archive, not for day-to-day file operations.

Every desktop installer carries it, and the Linux `.deb`, macOS `.pkg` and Windows installers put it on `PATH` (`/usr/bin` on Linux, `/usr/local/bin` on macOS, or the Windows install folder); a macOS `.dmg`-only install needs one `.pkg` (or **Diagnostics → Reinstall**) run for that entry — so if the desktop app is installed there's nothing extra to download.

On a machine without the Desktop App, the open-source standalone build (swarmfile-migrate, `curl -fsSL https://get.swarmfile.com/migrate | sh`) does the same job with its own flags (`--token`, `--state-path`, `--ec`; no `--dry-run` or `--report-json`); Migrating Existing Data compares the two. This page is the flag reference for the desktop build. For a practical walkthrough with a worked example, see Migrating Existing Data.

> Run with `--dry-run` first. It walks the source and reports what would be migrated without uploading anything, which is the safe way to check your exclude patterns and source/destination settings before committing to a real run.

Source selection

Pick exactly one source: a local directory, a flat file list, or an S3-compatible bucket.

Flag	Description
`--source`	Local directory source
`--source-list`	Flat file-list source
`--source-root`	Root to resolve relative paths in `--source-list` against
`--source-s3-bucket`	S3-compatible source bucket
`--source-s3-prefix`	Prefix within the S3 bucket to migrate
`--source-s3-endpoint`	S3-compatible endpoint URL
`--source-s3-region`	Default `us-east-1`

> S3 support ships in every released installer — `--source-s3-*` works out of the box, with no extra download. The AWS SDK it needs is large and lives in the `swarmfile-migrate` tool rather than the Swarmfile engine; a from-source build can opt out with `--no-default-features`, and a binary built that way fails immediately (naming the missing feature) instead of partway through a migration.

Destination

Flag	Description
`--root-path`	Destination path inside the repository. Default `/`
`--hub-url`	Your hub URL, e.g. `https://hub.swarmfile.com` (or your self-hosted hub's URL). Defaults to `$SWARMFILE_HUB_URL` if set, otherwise the hosted hub (`https://hub.swarmfile.com`) — so a machine whose environment already names its hub does not need this flag
`--org-id`	Target organization
`--repository-id`	Target repository. Defaults to `$SWARMFILE_REPOSITORY_ID` if set; with neither, entries are created **repository-less** — invisible to any mount that scopes its feed to a repository (which every engine does by default), so set one for a normal import
`--oidc-issuer`	OIDC issuer for authenticated hub access
`--oidc-client-id`	OIDC client ID
`--oidc-refresh-token`	OIDC refresh token

Authentication

The simplest and primary way to authenticate is a **repository-scoped API key** in the environment: set `SWARMFILE_API_KEY=sf_key_…` and `swarmfile-migrate` picks it up automatically — the same headless credential every other Swarmfile binary honors — with no auth flags at all. This is the recommended path for a CI runner or a render-farm import box.

The `--oidc-issuer` / `--oidc-client-id` / `--oidc-refresh-token` flags are the alternative: pass **all three together** to authenticate with an OIDC refresh token instead. If you pass no API key and not all three OIDC flags, hub calls go out unauthenticated and will fail against any hub that enforces auth.

Exclusions

Flag	Description
`--exclude`	Glob pattern to exclude. Repeatable. Matching a directory prunes its whole subtree.
`--exclude-from`	File containing exclude patterns, one per line

Reliability

Flag	Description
`--concurrency`	File-level concurrency. Default `8`
`--block-concurrency`	Max concurrent block uploads, shared across all files. Default `4`
`--retry`	Max retry attempts per file — both in-run (a transient failure retries immediately, with backoff, before the run moves on to the next file) and across separate re-invocations of this tool against the same `--state-db`. Default `3`; `0` disables retries either way
`--verify`	Verify each migrated file against the source with a CID/hash comparison. On by default (`true`); pass `--verify=false` to skip — verifying means reading the source a second time, so skipping it roughly halves I/O for a very large migration where you trust the transport
`--state-db`	Local state DB path. Tracks what's already uploaded and enables resuming a re-run. Default `<source>/.swarmfile-migration/migrate.db`; for a `--source-list` source, it sits alongside the list file instead
`--report-json`	Write a JSON report of the run
`--start-at`	Resume marker to start from

Safety

Flag	Description
`--dry-run`	Report what would be migrated without uploading anything
`--force`	Re-migrate files already registered/uploaded in a prior run, instead of skipping them

Encryption & erasure coding

Key selection

There's no `--encrypt` flag — migrated data is encrypted the same way any other write to the repository is, driven by environment variables rather than a migration-specific setting: `SWARMFILE_ENCRYPTION_KEY` / `SWARMFILE_ENCRYPTION_REPOSITORY_ID`, or simply `--repository-id` combined with OIDC credentials, resolves the key.

Encryption is on by default; if no key source is configured for the invocation, the tool warns and migrates the data as plaintext rather than refusing to run — unless encryption was explicitly requested, in which case it fails instead.

Repository tiers

With `--repository-id`, the destination repository's encryption tier decides: a managed repository's files are encrypted with its key, a plaintext repository's files are stored unencrypted (as that tier stores everything), and an end-to-end encrypted repository is refused, because migrate has no way to encrypt for it.

Erasure coding

Erasure coding is controlled by `SWARMFILE_EC_ENABLED` (`on` / `off` / `auto`): default `auto` (also what an unset or unrecognized value falls back to) **never erasure-codes during migration**, because `auto` is evaluated from live peer/network topology and a bulk import has none — set `SWARMFILE_EC_ENABLED=on` (or `true`) to force erasure coding on the migrated blocks.

Examples

Step 1 — preview with `--dry-run`. Always start here: it walks the source and reports what would be migrated without uploading anything, so you can confirm your exclude patterns and source/destination settings before committing to a real run:

```
swarmfile-migrate --source /Volumes/nas/RepositoryArchive \
  --org-id acme-films --repository-id feature-01 \
  --exclude "*.cache" --exclude "**/tmp/**" \
  --dry-run
```

Step 2 — the real run. Once the dry-run looks right, drop `--dry-run` (post-upload verification is on by default, so no flag needed). Here `SWARMFILE_API_KEY` supplies the credential and `$SWARMFILE_HUB_URL` the hub, so neither needs a flag:

```
export SWARMFILE_API_KEY=sf_key_…
swarmfile-migrate --source /Volumes/nas/RepositoryArchive \
  --org-id acme-films --repository-id feature-01 \
  --exclude "*.cache" --exclude "**/tmp/**" \
  --state-db ./migrate-state.db
```

S3-compatible bucket (preview it with `--dry-run` the same way first):

```
swarmfile-migrate --source-s3-bucket repository-archive \
  --source-s3-prefix renders/2025 \
  --source-s3-endpoint https://s3.us-west-000.backblazeb2.com \
  --hub-url https://hub.swarmfile.com \
  --org-id acme-films --repository-id feature-01 \
  --dry-run
```

swarmfile-search

`swarmfile-search` searches file, folder, repository and people names in the local Swarmfile metadata cache, with fallback to the hub. It's a standalone binary for searching from a terminal or a script, rather than the dashboard omnibox. Every desktop installer carries it, and the Linux `.deb`, macOS `.pkg` and Windows installers put it on `PATH` (`/usr/bin` on Linux, `/usr/local/bin` on macOS, or the Windows install folder); a macOS `.dmg`-only install needs one `.pkg` (or **Diagnostics → Reinstall**) run for that entry.

The local cache is entries-only: it indexes file and folder names in a local full-text index, and it's what a plain query checks first. Repository and people (principal) name matches only come from the hub fallback — the local cache doesn't mirror those, so a query for a repository or person name needs a hub round-trip (or `--remote`) to find anything.

On the hub the two halves also live in different places: entries are indexed per repository, while repository and people names are org-wide. A query scoped with `--repository` (or `$SWARMFILE_REPOSITORY_ID`) therefore asks both routes and merges them, so you still get repository/people matches alongside the file hits; an unscoped hub query searches repository and people names only — **file hits from the hub need `--repository`**, or a warm local cache.

Either way, it's a metadata search — it finds things by name, not by file contents.

For the conceptual overview of how search resolves results and where else you can search from, see Search.

Usage

```
swarmfile-search <query>
```

Flags

Flag	Description
`--repository`	Scope the search to a single repository by ID. Defaults to `$SWARMFILE_REPOSITORY_ID` when set
`--limit`	Maximum results. Default `20`
`--remote`	Force the search against the hub. Conflicts with `--local-only`
`--local-only`	Force the search against the local cache only. Conflicts with `--remote`
`--json`	Print raw JSON instead of a human-readable list. Emits the hub's `{files, folders, repositories, principals}` shape plus a top-level `source` field

Examples

Plain query across the local cache, falling back to the hub if needed:

```
swarmfile-search "concept_v3"
```

Scoped to a repository, capped at 5 results, local cache only:

```
swarmfile-search "elevation" --repository proj_xyz789 --local-only --limit 5
```

JSON output for scripting:

```
swarmfile-search "plan.dwg" --json
```

Alongside the hub-shaped `{files, folders, repositories, principals}` object, `--json` adds a top-level `source` field so a script can tell where the results came from without changing the other keys — `local`, `local (no fallback)`, `remote`, or `remote (local fallback)`.

swarmfile-seed

`swarmfile-seed` seeds a single local file into the Swarmfile hub: it reads the file, chunks it, uploads the resulting blocks, and registers a file entry (creating any missing parent directories along the way) at a virtual path you specify. It's a standalone binary with no subcommands — one invocation seeds one file.

This page documents the build the desktop installers ship. Every desktop installer carries it, and the Linux `.deb`, macOS `.pkg` and Windows installers put it on `PATH` (`/usr/bin` on Linux, `/usr/local/bin` on macOS, or the Windows install folder); a macOS `.dmg`-only install needs one `.pkg` (or **Diagnostics → Reinstall**) run for that entry.

> **Two builds share the name.** The open-source standalone `swarmfile-seed` (`curl -fsSL https://get.swarmfile.com/seed | sh`, source at swarmfile-seed) is for machines without the Desktop App. It takes every target as a flag: `--hub-url`, `--org-id` and `--repository-id` (all required), `--token` (defaults to `$SWARMFILE_API_KEY`), `--branch`, plus the same `--file`, `--path`, `--cdc` and `--auto-profile`. Seeding a File Into a Repository walks through it. The flags and environment variables below apply only to the desktop-bundled build; `swarmfile-seed --help` shows which one is on your `PATH`.

> `swarmfile-seed` is a one-shot ingestion tool, not a persistent process. It uploads the file you point it at and exits. For running a machine as an ongoing, persistent participant in the peer-to-peer swarm, see `swarmfile seed enable` on the full client, covered in Self-Hosted Seed Nodes.

Flags

Flag	Description
`--file`	Path to the local file to upload
`--path`	Virtual path in the filesystem (e.g. `/Repositories/plan.dwg`)
`--hub-url`	Hub URL. Defaults to `$SWARMFILE_HUB_URL`, else the hub the install's `config.json` names, else the build's fallback (`https://hub.swarmfile.com` in a **release** build, `http://localhost:8787` in a **debug** build). Override for a self-hosted / Enterprise hub
`--cdc`	Use content-defined chunking (FastCDC) instead of fixed-size
`--auto-profile`	Auto-select chunking strategy based on file extension (AEC profiles). Conflicts with `--cdc`

`--file` and `--path` are both required. `--path` must not be empty or contain empty segments (no leading/trailing/double slashes past trimming) — it's validated before any I/O happens.

Chunking

By default, `swarmfile-seed` chunks the file with a fixed chunk size. `--cdc` switches to content-defined chunking (FastCDC), which tends to deduplicate better across files with small internal shifts. `--auto-profile` instead picks a chunking strategy per the file's extension using the same AEC (architecture/engineering/construction) profiles the rest of the ingest pipeline uses, and is mutually exclusive with `--cdc`. If neither flag is passed, fixed-size chunking is used.

What it does not do

Unlike `swarmfile-migrate`, `swarmfile-seed` uploads exactly one file per invocation — there's no directory-walk or file-list mode. It also does not erasure-code the upload — a deliberate scope limit for this single-file admin/seeding tool.

This build has no `--repository-id` or `--org-id` flag (the standalone build does); repository selection is environment-driven.

The entry has to land in a repository — the repository-scoped metadata route is the only one that can accept a create — so `swarmfile-seed` resolves it from `SWARMFILE_REPOSITORY_ID` (falling back to `SWARMFILE_ENCRYPTION_REPOSITORY_ID` when that is the only one set, for the managed-key path) and fails with an actionable error before uploading anything if neither is set. If both are set they must name the same repository.

Any encryption-key lookup keyed to a repository is likewise driven entirely by environment variables rather than a CLI flag. The key sources are:

- `SWARMFILE_ENCRYPTION_KEY` — a static hex-encoded key you supply directly.
- `SWARMFILE_ENCRYPTION_REPOSITORY_ID` — resolve a managed-tier repository's hub-held key. This path additionally needs OIDC credentials in the environment: `SWARMFILE_OIDC_ISSUER`, `SWARMFILE_OIDC_CLIENT_ID`, and `SWARMFILE_OIDC_REFRESH_TOKEN` (all three).

Encryption is **on by default**. If no key source is configured for the invocation, the tool warns and seeds the file as plaintext rather than refusing to run — unless encryption was explicitly requested (any of the above set), in which case it fails instead. To silence the plaintext-mirror warning on a node that is deliberately seeding without encryption, set `SWARMFILE_ENCRYPTION_ENABLED=false`.

Example

```
SWARMFILE_REPOSITORY_ID=<repository-id> swarmfile-seed --file ./plan.dwg \
  --path "/Repositories/Site A/plan.dwg" \
  --hub-url https://hub.swarmfile.com --auto-profile
```

swarmfile-mirror

`swarmfile-mirror` copies a repository (or a subtree) from one Swarmfile hub to another over the public HTTP API — for example, moving a repository off the hosted hub onto your own self-hosted `swarmfile-server`, or keeping a second copy. It is the counterpart to `swarmfile-migrate`: migrate brings a local tree or an S3 bucket *in*; mirror copies a *repository* to *another repository*.

A mirror is normally a copy, not a signed replica: files are re-chunked/re-sealed for the destination, so it has no origin commit or checkpoint.

With `--provenance-out` on a plaintext whole-repository mirror, the copy preserves the source's block CIDs, manifests and entry lineage, and the origin's signed manifest is written beside it, so `swarmfile-verify-history --pin-key <key> --provenance <file>` can prove the destination's confirmed head commit or tree is the origin's (the manifest's key ring is untrusted input, hence the pin).

A managed/E2E **mirror** copy re-seals under the destination's key, so its tree differs and the flag refuses; the community server's optional `admin copy-repository` is the same-server exact-history alternative.

`swarmfile-verify-history --mirror-*` likewise matches only a replica that preserves the origin's confirmed head commit. See Verifiable History.

It ships as a standalone open-source binary (install with `curl -fsSL https://get.swarmfile.com/mirror | sh`; source and releases at swarmfile-mirror); no engine or mount is needed. See the Mirroring a Repository Between Hubs guide for the workflow.

Figure omitted in print (animation): Two community servers, and the standalone swarmfile-mirror copies a repository between them over the HTTP API: run it again and the state file resumes the copy, re-uploading nothing, and the destination then passes the same doctor checks as the source. - https://swarmfile.com/demo/oss-two-hubs.cast

Flags

Flag	Description
`--source-hub-url <url>`	Source hub base URL.
`--source-org-id <org>`	Source organization id or slug.
`--source-repository-id <id>`	Source repository id.
`--source-token <token>`	Source token (`sf_pat_…` / org key). Defaults to `$SWARMFILE_SOURCE_API_KEY` , then `$SWARMFILE_API_KEY` .
`--dest-hub-url <url>`	Destination hub base URL.
`--dest-org-id <org>`	Destination organization id or slug.
`--dest-repository-id <id>`	Destination repository id.
`--dest-token <token>`	Destination token. Defaults to `$SWARMFILE_DEST_API_KEY` , then `$SWARMFILE_API_KEY` .
`--path <path>`	Subtree to mirror (default: the whole repository).
`--branch <name>`	Branch to read on the source (default `main`).
`--cdc`	Content-defined chunking (FastCDC) instead of fixed-size.
`--auto-profile`	Auto-select the chunking strategy from each file's type (AEC profiles). Conflicts with `--cdc` .
`--exclude <glob>`	Skip entries by name (repeatable, `*` / `?`).
`--dry-run`	List what would be copied; touch nothing. With `--provenance-out` / `--preserve-tree` it still fetches the confirmed head tree and verifies the live source listing, but writes no manifest.
`--json`	Print a JSON summary (files, bytes, dirs, symlinks, resumed, skipped).
`--state <file>`	Resume file: records each mirrored source path → (cid, size) atomically; a re-run skips already-mirrored files. Refuses a state file from a different run.
`--concurrency <n>`	Files copied in flight (default 4; `1` = sequential). Ignored with `--state` and in `--dry-run` .
`--preserve-tree`	Plaintext whole-repository copy that keeps the source's block CIDs and entry lineage (no manifest). Use when the origin has no signed history but the exact tree matters. Same constraints as `--provenance-out` : empty destination (or matching resume state), no `--path` / `--exclude` , plaintext both sides; managed/E2E re-seals and is refused. `--dry-run` performs the tree/listing verification without writing.
`--provenance-out <file>`	Write the origin's portable provenance manifest after a successful **plaintext whole-repository** mirror. Implies a tree-preserving copy: the source's block CIDs, manifest/segments and entry **lineage** are kept, so the destination's tree (and usually its confirmed commit) matches. Requires an empty destination branch (or a resume with the same `--state`), a source with a confirmed head and zero unconfirmed commits, no `--exclude` / `--path` , and a credential that can read the whole tree; sync-ignored entries are included (`showIgnored=1`). It fetches the confirmed head's canonical tree and refuses if the source or destination live listing differs (lineage/content/kind/target), and a resume state is pinned to the source head. Refused for a managed/E2E source or destination, which re-seals and cannot match. `--concurrency` / `--cdc` / `--auto-profile` do not affect this mode (source chunks are copied verbatim). With `--dry-run` it performs only the source tree/listing verification and writes no manifest. Verify with `swarmfile-verify-history --pin-key <key> --provenance <file>` .

Exit codes

Code	Meaning
`0`	Success.
`1`	A failure (including a refused collision or an absolute symlink target).
`2`	A usage error.

Examples

```
export SWARMFILE_SOURCE_API_KEY=sf_pat_…    # source hub token
export SWARMFILE_DEST_API_KEY=sf_key_…      # destination hub token

swarmfile-mirror \
  --source-hub-url https://hub.swarmfile.com \
  --source-org-id my-org --source-repository-id <id> \
  --dest-hub-url https://your-server \
  --dest-org-id my-org --dest-repository-id <id> \
  --path /Repositories --state /tmp/mirror.json
```

swarmfile-brlock

`swarmfile-brlock` drives the engine's byte-range lock API directly. It talks to a **running engine** over its control socket, so start the engine (or the Desktop App) first.

It's a developer/integrator tool — a test client for the same lock primitive that a native worksharing app plugin (the kind of Revit-class CAD/BIM integration that needs to lock only the elements one person is editing, not an entire file) would implement against.

It is not something a typical end user runs, and **it isn't part of any installer or public download**. If you're building a plugin against the byte-range lock API, contact us for the client; this page documents its interface.

For the conceptual introduction to byte-range locking and how it differs from a whole-file entry lock, see Working with Files.

Global flags

Flag	Description
`--cache-dir`	Engine cache dir to connect against. Defaults to `$SWARMFILE_CACHE_DIR`, else `~/.cache/swarmfile` (macOS/Linux) or `%LOCALAPPDATA%\swarmfile` (Windows).

`acquire`

Acquire an exclusive byte-range lock.

Flag	Description
`--entry-id`	Entry to lock a byte range on
`--offset`	Start offset, in bytes (`u64`). Default `0`
`--length`	Length of the range, in bytes (`u64`). Default `0`, which means "to end of file," not a zero-length range
`--handle-ref`	Handle the lock is tied to

`release-handle`

Release every lock tied to a handle ref.

Flag	Description
`--handle-ref`	Handle whose locks should all be released

Example: acquire and release a range

A plugin integration would typically acquire a range around the elements being edited, hold it for the duration of the edit, and release everything tied to its handle on close — this simulates that sequence from the command line:

```
swarmfile-brlock acquire --entry-id ent_9f2a --offset 4096 --length 512 --handle-ref plugin-session-1

# ...edit is in progress, range is locked...

swarmfile-brlock release-handle --handle-ref plugin-session-1
```

Releasing by handle ref rather than by individual range means a plugin doesn't need to track every range it acquired — closing out a session releases everything that session took.

Exit codes

Meaningful for scripting a CI/integration test against a real lock denial, which is the tool's main use case:

Exit code	Meaning
`0`	Success.
`2`	The lock was denied (a conflicting range is already held).
`1`	Any other error.

swarmfile-runner

`swarmfile-runner` runs the engine in **runner mode**: it sets `SWARMFILE_RUNNER_MODE=true` (when the variable isn't already set), then re-execs the sibling `swarmfile-engine` binary with the same arguments and standard I/O. It has no subcommands or flags of its own — every engine flag passes straight through — and it runs the real engine boot path (config, single-instance lock, lifecycle) exactly as `SWARMFILE_RUNNER_MODE=true swarmfile-engine` would.

It exists so the headless CI role reads as its own tool rather than "the engine with an env var". For what a runner does and how to wire it into a CI system, see Runner (Headless CI).

Because it re-execs the engine, `swarmfile-runner --help` is the engine's help. The pass-through flags:

Flag	What it does
`--build-info`	Print the enabled cargo features and target architecture, then exit.
`--status`	Engine, hub, peers, and pending work at a glance (works while another engine holds the lock).
`--capabilities`	The compiled feature set of this binary.
`--print-env-vars`	The resolved `SWARMFILE_*` values this build consumed, secrets redacted.
`--pause` / `--resume`	Pause or resume sync on a running engine without unmounting.
`--quit`	Ask the running engine to drain and stop.
`--pin` / `--unpin` / `--list-pins` / `--verify-pins`	Offline cache pins: add, release, list, and verify.
`--prepare-offline [SECS]` / `--return-from-offline`	The engine-level Pack & Go pair — see Working Offline.

See Engine Config File → Engine command-line flags for the full descriptions.

Environment

Variable	Description
`SWARMFILE_RUNNER_MODE`	Forced to `true` when unset. An explicit value is preserved, so `SWARMFILE_RUNNER_MODE=false swarmfile-runner` still runs in normal mode
`SWARMFILE_API_KEY`	Repository API key (`sf_key_…`) the runner authenticates as. The hosted hub refuses to start with a PAT unless `SWARMFILE_RUNNER_ALLOW_PAT=1` (jobs run, but their reports are refused and the Runs tab stays empty)
`SWARMFILE_ORG_ID`	Org that owns the repository. Optional with a repository API key (the runner resolves it from the repository API key) or when the hub URL already ends in `/orgs/<id>` ; otherwise required, including with a PAT or an older hub
`SWARMFILE_REPOSITORY_ID`	Repository the runner serves. Optional with a repository API key (resolved with the org)
`SWARMFILE_BRANCH`	Branch the runner tracks (defaults to the repository's default branch)

Runner mode deliberately serves no control socket. A driverless CI host needs no FUSE — `materialize` and runner mode never call the mount path.

Acquiring it

- **Linux** — the `.deb` installs it at `/usr/bin/swarmfile-runner` , next to `swarmfile-engine` .
- **macOS** — it ships inside `Swarmfile.app/Contents/MacOS/swarmfile-runner` , next to the bundled engine, with a `/usr/local/bin/swarmfile-runner` symlink on a prod install.
- **Windows** — the MSI installs it next to `swarmfile-engine.exe` in the install folder, and the app-only zip carries it.
- **Keep it next to the engine.** It must sit in the same directory as `swarmfile-engine` , because it re-execs the sibling resolved next to its own executable; the installers put it there.
- **Equivalent without the alias** — on any engine binary, `SWARMFILE_RUNNER_MODE=true swarmfile-engine` behaves identically.

Example

```
SWARMFILE_API_KEY=sf_key_... SWARMFILE_ORG_ID=org_abc123 \
  SWARMFILE_REPOSITORY_ID=proj_xyz789 swarmfile-runner
```

Because it re-execs the engine, the process's exit code and output are the engine's. If the sibling `swarmfile-engine` is missing, the alias prints an error and exits `127` .

swarmfile-verify-history

`swarmfile-verify-history` is a standalone verifier for a repository's signed commit history. It walks the commit DAG from a branch's current `HEAD` back toward genesis, refetching each commit object from the hub, and recomputes every hash from the commit's own facts to prove the chain hasn't been tampered with: each parent link resolves and each stored hash matches what its facts actually hash to.

Every desktop installer carries it, and the Linux `.deb`, macOS `.pkg` and Windows installers put it on `PATH` (`/usr/bin` on Linux, `/usr/local/bin` on macOS, or the Windows install folder); a macOS `.dmg`-only install needs one `.pkg` (or **Diagnostics → Reinstall**) run for that entry.

The verifier is open source (dual MIT/Apache-2.0): source, releases and its test vectors are at swarmfile-verify-history, so checking the history doesn't require trusting our binary.

It also verifies the hub's **checkpoint signatures**. Every checkpoint the hub has published for the branch is fetched and its signature checked against a public-key ring — see the trust caveat below.

It does its own minimal HTTP GETs, with no dependency on a running engine or mount, so it can run on a CI box or an audit host.

Figure omitted in print (animation): swarmfile-verify-history recomputes a branch's commits from the canonical encoding. Then one commit's recorded message is edited directly in the server's database, and the next run names the commit it broke and exits non-zero. The chain is recomputed, not taken on the hub's word. - https://swarmfile.com/demo/oss-tamper-verify.cast

Usage

```
swarmfile-verify-history \
  --hub-url https://hub.swarmfile.com/orgs/org_.../repositories/proj_... \
  --org-id org_... \
  --repository-id proj_... \
  --branch main \
  --header 'Authorization: Bearer <token>'
```

`--hub-url` is the **repository-scoped** base: the verifier appends `/history/…` and `/metadata/…` paths to it, while `--org-id` / `--repository-id` identify the org and repository to verify. When the URL is in the `/orgs/<org>/repositories/<repository>` form, its two segments must be the same ids (the verifier validates them; use ids, not slugs, in the URL).

Add `--json` for a machine-readable report.

Flags

Flag	Description
`--hub-url <URL>`	Repository-scoped hub base URL (required) — the repository this run verifies, e.g. `https://hub.swarmfile.com/orgs/org_.../repositories/proj_...`.
`--repository-id <ID>`	Repository to verify (required).
`--org-id <ID>`	Org id (required).
`--branch <NAME>`	Branch to verify. Default `main`.
`--header "Key: Value"`	Extra raw header, repeatable — pass whatever auth header this deployment requires (for example, `Authorization: Bearer …`). No auth scheme is hardcoded.
`--header-file <FILE>`	Read extra raw headers from a file, one `Key: Value` per line (repeatable). Prefer this over `--header` for tokens so secrets do not appear in the process list.
`--checkpoint-hash <HASH>`	Stop walking when this commit is reached (inclusive), instead of continuing to genesis. Use when older history has been pruned server-side.
`--pin-key <PUBKEY|FILE>`	Pin the hub's checkpoint-signing Ed25519 public key out-of-band (64 hex, base64, or a file path). Repeatable for key rotation.
`--check-host-identity`	Fetch and verify the hub's signed `/.well-known/swarmfile.json` (its wire shape and synthetic fixture ship with the verifier's public-repo template and publish with its next release): require its `origin` to equal the `--hub-url` origin, check the self-signature, and require its key to be the active checkpoint key (or one of your `--pin-key`s). HTTPS is required unless the hub is loopback; redirects are refused and the statement must be at most one hour old. A hub that does not serve the document fails this check. Self-hosted servers serve it (set `SWARMFILE_SERVER_PUBLIC_URL`); the hosted hub serves it at the hub origin (dispatch needs the same history-signing secret and `HUB_PUBLIC_ORIGIN`).
`--write-provenance <FILE>`	After a fully successful origin verification, write a portable manifest: the canonical origin, the head's commit facts and tree CID, every signed checkpoint, the key ring, and the origin's signed `/.well-known/swarmfile.json` when it serves one. It refuses to write unless every checkpoint verifies, the confirmed head has one, and the head tree agrees with `headTreeCid`. `--provenance` consumes it.
`--pin-manifest <BLAKE3>`	Pin the manifest FILE's BLAKE3, obtained out-of-band. `--provenance` refuses a different file, so every field—branch, ids, checkpoints, keys—is authenticated, not just the head the checkpoint signs.
`--verify-blocks [N]`	Sample up to N files (default 3) from the mirror, fetch their manifest block and—for a plaintext manifest—one chunk block, and re-hash each against its content id. Requires a matching head commit in `--mirror-*`, or a matching commit/tree in `--provenance`; a sealed/unparseable managed manifest is reported as not content-verifiable (its leaf cid is the plaintext hash). This proves the sampled bytes are stored, not that every block is served; a sealed managed manifest gets its manifest block hashed but no chunk.
`--origin <URL>`	With `--provenance`, bind the manifest's origin label to the host whose key you pinned (e.g. `https://hub.example.com`). The manifest's origin is not signed by itself; this makes a relabelled manifest fail.
`--provenance <FILE>`	Verify the repository named by `--hub-url` as a **mirror** against a manifest, without contacting the origin. **Requires at least one `--pin-key`**: the manifest is a file, so its key ring is untrusted input and a pin is the only offline trust anchor. The head is re-hashed, its checkpoint and (when present) the host-identity statement are signature-checked against the pins, and the mirror passes on a matching confirmed head **commit**, or a matching head **tree** (a plaintext destination written by `swarmfile-mirror` in provenance mode preserves the source tree/lineage, so it matches; an ordinary plaintext copy has a different commit DAG but the same content tree; a managed/E2E **mirror** copy re-seals, so its tree differs (a same-server `swarmfile-server admin copy-repository` copies the key/data without re-sealing)). `originAttested` reports whether the origin statement was carried and verified. Cannot combine with `--check-host-identity` or `--mirror-*`.
`--mirror-hub-url`, `--mirror-repository-id`, `--mirror-org-id`, `--mirror-branch`, `--mirror-header`	Compare a mirror's **confirmed head commit** to the origin's verified head commit. `mirrorHeadMatches` is true only when the mirror's confirmed head commit equals the origin's verified head commit; a mismatch (or a mirror with no confirmed head) is a failure. The commit hash binds the tree CID, so a matching head cannot substitute a different tree. The mirror's own signatures/claims are not trusted, and the verifier does **not** fetch the mirror's tree objects or blocks — this is head-commit identity, not a byte-serving proof. Read it together with the checkpoint signature/pin result; the checker does not require the head itself to be checkpointed.
`--json`	Emit JSON instead of the human-readable report.

What it proves — and what it doesn't

- **Structural integrity** is real: every hash recomputes and every parent link resolves across the commits the hub served. That shows the chain is internally consistent; independent assurance that the hub is attesting to the same chain it originally recorded comes from the signed checkpoints below.
- **Signatures without `--pin-key`** are checked against the key ring fetched from the *same hub being audited* (trust-on-first-use). That proves the checkpoints were signed by whatever key that hub currently publishes — not that the key is the hub's real one. The report says so on every run.
- **With `--pin-key`** signatures are verified only against keys you supply out-of-band; the run fails if none is in the hub's ring, and a checkpoint signed by any other key fails. This is the mode that gives independent assurance.
- Where checkpoint signing isn't enabled, the key route answers `404`: unpinned that reads as "verifiable history isn't enabled here" (informational, exit OK); with `--pin-key` it is a failure.
- **`--mirror-*`** proves head-commit identity, not mirror trust or byte serving: the origin's commit chain/checkpoints are verified exactly as above, and the mirror contributes only its confirmed head commit. A match means the mirror's head is the same commit/tree as the origin's verified head; the verifier does not fetch the mirror's tree/blocks or require the head to be checkpointed. An ordinary `swarmfile-mirror` copy does not preserve the commit DAG — use a replica that does, or a plaintext provenance-mode mirror copy (which matches by commit or tree).
- **`--provenance`** proves the manifest's head is signed by a pinned key and (when present) that the origin's identity statement is genuine and pinned; `originAttested: false` means the origin served no statement and the origin NAME is only a label. A tree/commit match still does not prove the mirror serves the tree object or any block — fetch a file to check bytes.
- **`--check-host-identity`** adds a key↔origin binding: the statement's self-signature is checked over `swarmfile-host-v2` (origin, key id and issue time), its `origin` must equal the `--hub-url` origin, and its key must be the ring's active key (or a pinned key). The statement is signed from the server's configured public

origin, never the request `Host`, so a mirror cannot make the origin vouch for another hostname. TLS to that origin is the trust anchor; the flag fails closed on a server that does not serve the document, including a branch with no confirmed commits.

Exit code is non-zero on any verification failure, so it is safe to gate CI on.

swarmfile-engine lfs-transfer

`lfs-transfer` is the Desktop App's built-in **git-LFS custom transfer agent**. `git-lfs` invokes it; it is not a command you run by hand. Point `lfs.url` at your Swarmfile repository and set git-lfs to use this agent as the transfer, and pushes ride it chunked and resumable (up to the 256 GiB object cap), while pulls use the plain resumable GET path.

It ships as a subcommand of the engine: `swarmfile-engine lfs-transfer` (the engine binary is part of every desktop install). There is no separate `swarmfile-lfs-transfer` binary to install.

Don't confuse it with **`swarmfile-lfs`**, the standalone open-source agent (and `reflect` tool) for stock `git-lfs`. It ships on `PATH` with every desktop install, and machines without the app install it with `cargo install swarmfile-lfs-transfer` (the crates.io package is named `swarmfile-lfs-transfer`, but the binary it installs is `swarmfile-lfs`) or `curl -fsSL https://get.swarmfile.com/lfs | sh`. Its source and releases are at swarmfile-lfs. For most setups `swarmfile-lfs install` is the simpler path; the git-LFS guide compares the two agents.

How git-lfs calls it

git-lfs spawns a configured custom transfer agent with an operation and remote, and talks to it over stdio (the LFS transfer protocol). It takes no flags of its own. Configure it per repository by pointing git-lfs at the engine binary:

```
git config lfs.url "https://hub.swarmfile.com/orgs/<org>/lfs/<repository>"
git config lfs.standalonetransferagent swarmfile-chunked
git config lfs.customtransfer.swarmfile-chunked.path swarmfile-engine
git config lfs.customtransfer.swarmfile-chunked.args lfs-transfer
```

`swarmfile-engine lfs-transfer --help` prints the same setup and exits without starting the protocol.

See git-LFS for the full setup, the `.lfsconfig` auto-PR flow, and how LFS objects are billed and reflected onto the mount.

Notes

- LFS bytes are metered and **storage-billed like any other content** — there is no separate LFS SKU.
- On a managed (encrypted) repository, uploads must go through the transfer agent — this one or the standalone `swarmfile-lfs` — so objects are encrypted at rest. The stock direct paths are refused with the typed `lfs_agent_required` error. On an unencrypted repository the direct paths remain available (plaintext by definition). E2E repositories **refuse** LFS outright. Objects a managed repository stored before the agent requirement are converted by an operator-run migration before you can say every object is encrypted (see the git-LFS guide's older-objects note).
- Because it is spawned by git-lfs, there is nothing to run interactively; if you need a human-facing status, use the mount or `swarmfile status`.

git-remote-swarmfile

`git-remote-swarmfile` is the small **git remote helper** behind `git clone`, `git fetch` and `git push` on a `swarmfile://` URL. Git invokes it by name whenever it sees a `swarmfile://` remote, so you never run it yourself. The desktop build re-execs the engine binary in `git-remote` mode; either build needs no running engine, no drive and no mount.

Every desktop installer puts it on `PATH` beside `swarmfile` (on macOS, `/usr/local/bin`; on Linux, `/usr/bin`). On a machine without the app, install the open-source standalone helper with `curl -fsSL https://get.swarmfile.com/git-v | sh` (Windows: `irm https://get.swarmfile.com/git-v.ps1 | iex`); source and releases are at swarmfile-git-remote.

The standalone build has no built-in LFS transfer agent; pair it with `swarmfile-lfs` for objects over about 5 GiB.

With the desktop app installed, if `git` says it can't find a helper for `swarmfile`, use `swarmfile git clone`, which finds the helper next to the `swarmfile` binary even when it isn't on `PATH`; with the standalone build, put its install directory on `PATH`. Git 2.11 or later is required.

Figure omitted in print (animation): An owner enables a repository's git surface with one command; an ordinary git clone speaks swarmfile:// through the helper; a commit pushes fast-forward; and swarmfile-verify-history walks the chain it just created. - https://swarmfile.com/demo/oss-git.cast

How git uses it

```
git clone swarmfile://<org>/<repository>                    # the configured hub
git clone swarmfile://<hub-host>/<org>/<repository>         # a specific hub
git clone swarmfile://<hub-host>/fed/<peer>/<org>/<repository>  # a federated public read
git remote add origin swarmfile://<hub-host>/<org>/<repository>
git fetch origin
git push origin main
```

The host-qualified form names the authority, so a copied link means the same repository on every machine that can reach that host; the relative form resolves against the receiving machine's configured hub (`SWARMFILE_HUB_URL`, or the Desktop App sign-in in the desktop build).

An authority is `host[:port]`; **the org segment is required** — `swarmfile://<hub-host>/<repository-id>` is read as a relative URL whose org is the host, not as a repository on that host.

The grammar carries no scheme: the helper uses HTTPS by default, and HTTP only for loopback or an authority listed in `SWARMFILE_GIT_INSECURE_HOSTS`.

A renamed repository or org keeps the old slug as a reserved alias, so an old clone URL keeps resolving for a session, an anonymous public clone, or a repository key on a **public** repository (the directory carries aliases); a keyed **private** remote should use the ids.

The four-segment `fed` form reads a **public plaintext repository through a federating server**: `swarmfile://<hub-host>/fed/<peer>/<org>/<repository>`. `<peer>` is a name pinned on `<hub-host>`, not an org or a URL; the repository lives on that peer and must be exported to `<hub-host>`. The session is anonymous and read-only by construction — an ambient or stored credential is **not** sent, git-LFS **downloads** are carried through the federator (uploads and locking are not), and a push is refused with a typed read-only reason. The federating server carries the origin's bytes and signed history unchanged (no re-signing, no byte cache).

`<repository>` may be a bare repository name or a repository id. A bare name is qualified with the configured organization by `swarmfile git url` / `swarmfile git clone`. Git access is turned on per repository — from the web repository menu, the Desktop App's **Repository settings → Git access**, or `swarmfile git enable` — and a clone of a repository that hasn't enabled it is refused with `no_git_view`; the org's owners are nudged once, and `swarmfile git status` shows where a repository stands.

What it serves

Shallow clones with `--depth`, fetches, fast-forward pushes (merges land on auto-archived `merged/<sha>` branches), lightweight tag moves and deletes, and chunked resumable pushes above 5,000 paths.

Force push is refused by default for an older clone — a clone created through the helper sets `swarmfile.force-forward=true`, translating plain `--force` / `--force-with-lease` into a forward commit (the same end state `swarmfile git force-forward` lands); an explicit `git config swarmfile.force-forward false` keeps the refusal.

- Submodule gitlinks are carried as mode `160000` (the drive hides the submodule path).
- End-to-end-encrypted repositories are supported when `SWARMFILE_USER_KEY_PRIVATE_B64` and `SWARMFILE_USER_KEY_ID` name this device's enrolled user key (the helper unwraps the repository key locally; the view is all-inline).
- Direct HTTPS remotes on the hub are refused (an operator can run the read-only `swarmfile-git-gateway` for stock public clones).
- Annotated tags keep their exact object, and an octopus merge is accepted when every parent after the second is already on Swarmfile.
- `--shallow-since` / `--shallow-exclude` aren't supported.

The full list, and the version-control model around it, is in Clone and push with git and Git and Swarmfile.

Credentials

The **standalone build** reads its credential only from `SWARMFILE_API_KEY` (a repository API key or a personal access token), plus `SWARMFILE_HUB_URL` and `SWARMFILE_ORG_ID` when the URL names no host or org. With `SWARMFILE_API_KEY` unset it makes an anonymous public clone (below); the URL must then carry both the org and the repository slug.

The **desktop build** looks for, in order:

1. **`SWARMFILE_API_KEY`** in the environment. A repository API key wins over everything else, including a Desktop App session, and the helper says so when both are present.

2. **A Desktop App session** stored by the engine.
3. **A repository API key in git's credential store**, stored with `git credential approve` for the Swarmfile host.

An explicit `SWARMFILE_API_KEY` is global. For a host-qualified URL that names a different authority than `SWARMFILE_HUB_URL`, it is sent only when `SWARMFILE_ALLOW_CROSS_HUB_KEY=1`; otherwise the helper falls back to a git credential stored for that host, or the anonymous path.

With `SWARMFILE_HUB_URL` unset, the key is still **not** offered to an explicit host unless `SWARMFILE_ALLOW_CROSS_HUB_KEY=1` — the URL alone does not authorize a global key (the engine helper enforces the same rule).

An old public repository slug resolves through the hub's one-request `/public/<org>/_resolve/<repository>` route; a keyed push over an old **private** slug still needs the id or a signed-in session. The stored Desktop session and the key a self-hosted install saved for its server are never offered to another authority.

With none of these the helper falls through to an **anonymous public clone** when the repository allows it — a public, git-enabled repository with anonymous cloning on (the default; see Clone and push with git).

That clone is read-only, shallow unless `--depth` / `--unshallow` ask otherwise, and pulls large files through a ticket-bearing LFS URL. A repository that doesn't allow it fails with the sign-in guidance instead.

The same anonymous fallthrough covers a credential that IS present but the hub refuses for the repository (a session without access to it, an expired or revoked key, a key scoped to another repository) — and a public repository the credential can't even see yet, before its first published release: the helper prints one line naming the refusal or the failed lookup, then continues as the same anonymous, read-only clone. A repository that isn't publicly cloneable keeps the refusal.

When something goes wrong

A failed helper always prints its reason above git's `fatal: remote helper 'swarmfile' aborted session` line, and writes the same text to `git-remote-last-error.log` in the Swarmfile cache directory (`~/.cache/swarmfile` on macOS and Linux, `%LOCALAPPDATA%\Swarmfile` on Windows). The message names who can fix it; include that log in a bug report.