Migrating Off Dropbox: Beyond Dotfiles

Ronald 'Ryy' G. Thomas

2026-05-07

Figure 1: A wooden tray on a plain workbench, with one large compartment on the left holding a single round glass weight and three narrower compartments on the right holding three different small objects, suggesting a single category being separated into three purpose-fit ones.

A portable dotfiles repository is only the first half of leaving Dropbox. The other half is the rest of the workflow.

> **Note**
>
> **Multi-Machine Setup Series** (recommended reading order): Part 1: Migrating off Dropbox | Part 2: Multi-Laptop Bootstrap | Part 3: Security Audit

Introduction

I did not really appreciate how much of my daily workflow depended on Dropbox until I tried to leave it. The dotfiles migration described in post 24 is the obvious half: pull the configuration

files out of a cloud-mounted directory, put them under git, write an installer that creates symlinks. Once that work is committed, every dotfile in `$HOME` becomes version-controlled and reproducible on a new laptop with two commands.

The non-obvious half is everything else. The dotfiles deploy cleanly on a fresh machine, then the new shell tries to `cd ~/prj/some-project` and fails because the project tree still lives at `~/Library/CloudStorage/Dropbox/prj/`. Backup scripts hardcode the same Dropbox path. AI-tool history files live under a Dropbox-synced symlink and accumulate conflict copies on every overlapping write between machines. The configuration is portable; the workflow is not.

This post is the second half of post 24. It frames the problem as three independent layers (dotfiles, project content, append-only history), surveys the sync mechanisms each layer can use, and walks through a migration sequence that does not break a running pipeline mid-move. The companion deliverable is a decision worksheet that takes a reader from current state to a concrete migration plan. The choice of whether to leave Dropbox at all is left to the reader; this post is about doing it deliberately rather than partially.

> **Status of this migration**
>
> This post documents an ongoing migration on the author's own setup. At time of writing, Layer 1 (dotfiles) is complete: the repository exists at `~/dotfiles/`, the installer is written and dry-run-tested, and the architecture below has been exercised end-to-end on this laptop. Layer 2 (project content) and Layer 3 (append-only history) are in progress. The framework is presented as it stands today; the 'After completing this migration' results section near the end is labelled as expected outcomes rather than claimed ones, and will be updated with measured numbers once the migration reaches a fresh laptop. The companion plan tracking this work is available in the dotfiles repository as `MULTI_LAPTOP_PLAN.md`.

Motivations

- A new laptop bootstrap should be a one-command operation. Today, after running the dotfiles installer, half the workflow still requires installing Dropbox, signing in, and waiting for several gigabytes to materialise before anything functional happens.
- Dropbox sync of append-only files (shell history, AI-tool history, editor undo files) routinely produces conflict copies on multi-machine setups. The 12 conflict copies of `history.jsonl` discovered in this account on 2026-05-02 are one symptom of an architectural mismatch.
- Project content directories under Dropbox commingle with cloud-only artefacts that should not be synced (model output, large derived data, machine-specific caches). Excluding them via `.dropboxignore` is fragile and error-prone.
- The set of services that compete for `$HOME` (Dropbox, iCloud Drive, Google Drive, OneDrive) keeps growing, and choosing one default for everything is increasingly suboptimal as each service specialises.
- The work to reach 'workflow independence' is largely the same work as 'workflow modernisation': separating concerns by layer, choosing sync mechanisms that match each layer's semantics, and documenting each move so it can be audited later.

Objectives

1. Inventory the workflow against three layers (dotfiles, project content, append-only history) and classify every sync-relevant artefact into exactly one layer.
2. Produce a decision worksheet that, for each layer, names the candidate sync mechanisms and the trade-offs that distinguish them.
3. Sequence the migration so that the backup pipeline never points at a stale source path and the launchd jobs stay valid throughout the move.
4. Establish a fresh-machine bootstrap that does not require Dropbox and verifies the entire workflow end to end.

This post documents the author's own migration. If errors are spotted or better approaches are known, the comment thread below is the right place to note them.

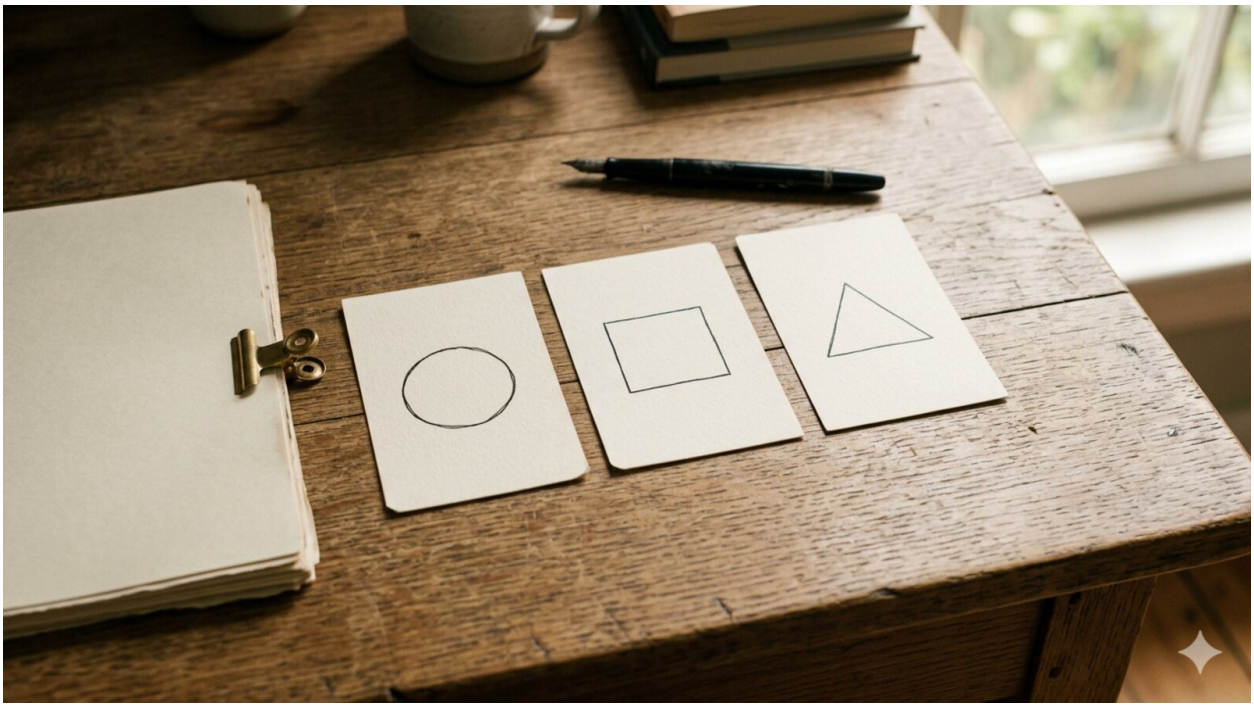

Figure 2: A wooden desk seen from above with three small index cards arranged in a horizontal row, each card showing a hand-drawn symbol, suggesting classification before action.

What is the Problem?

The problem is that 'sync everything via one cloud provider' was a reasonable default in 2014, when Dropbox was the only practical option for arbitrary file sync, but it now bundles three semantically distinct concerns into a single mechanism. The three concerns:

- **Configuration distribution** wants version history, atomic deploys, and the ability to roll back. Git solves this; cloud sync only approximates it.

- **Project content sync** wants block-level diffs for large files, conflict semantics that match the file format, and the option to exclude derived artefacts. Different sync mechanisms handle these very differently.
- **Append-only history** wants either single-machine writes or a sync mechanism that understands append semantics. Cloud sync writes the whole file on every change and produces a conflict copy on every overlap.

A useful analogy: it is the same reason a research compendium separates `data/`, `R/`, and `analysis/`. Each directory has different rules about what changes, how it changes, and who is allowed to change it. Lumping them all into one sync mechanism enforces no useful constraint and produces friction at the boundary.

A concrete example from this migration: the file `~/.claude/history.jsonl` is an append-only log of prompts entered into the Claude Code CLI. Five machines (athena, Joe, julia.local, MacBook-Air.local, and the author's primary laptop) write to it through a Dropbox-synced symlink. Most days, only one machine is active and the file syncs cleanly. The day two machines write within the same minute, Dropbox cannot merge the two append streams and creates a conflict copy. Across April 2026 alone, this produced four new conflict copies on this account, each holding several hundred KB of unique prompts; the running total since January is eleven. The file format is not the problem; the choice to sync an append log via a whole-file cloud provider is.

Prerequisites

This post assumes:

- **Operating system:** macOS 13+ or a recent Linux distribution. The sync-mechanism table is applicable to both, but specific path conventions reference macOS.
- **Already in place:** a working dotfiles repository per post 24. The architecture below builds on that foundation; without it, half the moves cannot be made cleanly.
- **Background knowledge:** comfort with git remotes, basic launchd plist structure, rclone or rsync at the level of a single sync command. The post does not require expertise in any of these.
- **Hardware:** a single primary laptop is sufficient. A second machine (or VM) is useful for the migration's verification step but is not required.
- **Time required:** about a day's careful work to inventory and decide; about a half-day to execute the moves once the decisions are made.

On a single-machine setup where conflict copies have never appeared, the migration's payoff is smaller and may not warrant the undertaking. The framework still applies, but Dropbox is unlikely to be causing friction in that scenario.

The Three Layers

Treat the workflow as three layers. Each layer has its own sync semantics, its own candidate mechanisms, and its own failure modes.

Layer 1: Dotfiles distribution

Solved by post 24. The summary, for reference: configuration files live in a git repository at `~/dotfiles/` (outside any cloud-mounted path), an `install.sh` creates symlinks into `$HOME` per machine, secrets stay out of the repository entirely. New-machine bootstrap is `git clone … ~/dotfiles && cd ~/dotfiles && ./install.sh`.

This layer is the easiest to migrate because the files are small, change infrequently, and are nearly always edited from a single machine. Git handles the multi-machine merge cleanly when overlap does occur.

What post 24 explicitly does not address: project content (the next layer), backup-pipeline source paths (also next layer), and append-only history (the third layer). Those failures show up only after the dotfiles layer is migrated, which is why this post exists.

Layer 2: Project content sync

Project content is the directory tree that holds research repositories, data, drafts, and archives. On a Dropbox-bound workflow this is `~/Dropbox/prj/` and similar siblings (`docs/`, `sbx/`, `work/`, `shr/`).

Project content has different sync characteristics from dotfiles:

- It is much larger (tens of GB is typical).
- It contains both small text (R, Quarto) and large binary (data, figures, PDFs).
- It changes frequently, often on a single machine within a session.
- It includes derived artefacts that should not be synced at all (renv libraries, model output, caches).

The candidate sync mechanisms, with the dimensions that matter:

Mechanism	Strength	Weakness
Per-project git remotes (GitHub, GitLab, Codeberg)	Diff-aware history, atomic deploys, free for small repos	Bad with large binary files (Git LFS quotas), each project must be a repo
Syncthing (peer-to-peer, no cloud)	No middleman, no quotas, sync-to-LAN is fast, files are local-first	Two machines must be online together to sync; conflict semantics are simpler than Dropbox's but still file-level
rsync over SSH to a NAS or VPS	Bandwidth-efficient, deterministic, scriptable	One-way push (or hand-coded bidirectional); requires the destination to exist; no version history without extra work
iCloud Drive (native macOS)	Tight macOS integration, included with Apple ID	macOS-only; opaque sync timing; cannot exclude subdirectories cleanly
Google Drive (rclone or native)	Effectively unlimited storage on enterprise plans	Same whole-file model as Dropbox; conflicts behave the same way

Most readers benefit from a hybrid: per-project git remotes for code (the bulk of the value) and Syncthing or rsync for the data that does not belong in a public or even private repo. The hybrid wins because it lets each artefact go to the mechanism that matches its semantics.

The single most consequential observation in this migration: 'project content' is not a homogeneous category. Splitting it into 'code' and 'data' before choosing a mechanism makes every subsequent decision easier.

Reconstructable local caches: the Maildir case

A subset of what looks like Layer 2 data is in fact a local cache of a canonical source that already lives elsewhere. For these, the right move is not to choose a sync mechanism at all; it is to let each machine rebuild its copy from the authoritative source on demand.

The worked example: a Maildir-based mail setup with mutt, mbsync, and notmuch. The IMAP server holds the canonical mailbox. `mbsync` mirrors a subset of folders into a local Maildir at `~/.local/share/mail/` (or any other non-synced path; the XDG `~/.local/share/<app>/` location is a sensible default). `notmuch new` indexes the Maildir into a local Xapian database. `mutt` reads the Maildir directly. On a second laptop, the same configs deploy via Layer 1, the same `mbsync` runs from a launchd job, the local Maildir gets repopulated from IMAP, and `notmuch new` rebuilds the index. There is nothing to sync between the two laptops; the IMAP server is doing the synchronisation by being the canonical source.

Putting the Maildir in Dropbox is wrong on four counts. It duplicates the work IMAP already does. It triples the disk footprint (IMAP + Dropbox + local). Maildir's one-message-per-file format produces hundreds of thousands of small files that strain cloud-sync engines. And the notmuch Xapian database is a B-tree-on-disk format that is not safe to share between processes on different machines via file sync.

The same pattern applies to a broader class of local artefacts: Homebrew's package cache (`~/Library/Caches/Homebrew/`), language-server indexes, browser caches, ccache, AI-model caches that hash to the same content given the same prompts. None of these need to be synced between machines; each is a derivative of an authoritative source and is rebuilt cheaply on demand. Treat them as per-machine state, exclude them from any sync mechanism, and add the rebuild step to the new-machine bootstrap if it does not happen automatically.

The new-laptop bootstrap for the Maildir case, after `install.sh` has deployed the configs:

```
# Restore IMAP/SMTP credentials (manual; from pass / keychain / 1Password).
# Then:
mbsync -a                       # populate ~/.local/share/mail/
notmuch new                     # build the Xapian index
launchctl bootstrap "gui/$UID" \
  ~/Library/LaunchAgents/local.mbsync.plist
launchctl bootstrap "gui/$UID" \
  ~/Library/LaunchAgents/local.notmuch.new.plist
```

Time depends on mailbox size; a 3 GB mailbox over residential broadband is roughly 30-60 minutes. The two scheduled jobs keep the Maildir current thereafter.

The one cost of this pattern is that mail is not available offline until the first `mbsync -a` completes. For mutt users this is rarely a meaningful constraint; for users who need immediate offline access on a fresh laptop, an `rsync` of a recent Maildir snapshot from the previous laptop into `~/.local/share/mail/` skips the IMAP fetch step.

Worked example: relocating an existing Maildir to ~/.local/share/mail/

A common starting point for long-time mutt users is a Maildir at `~/Mail/` (the convention from before the XDG Base Directory specification existed). The relocation to `~/.local/share/mail/` is mechanical, and the cache-and-index can be moved in place rather than re-downloaded. The steps below assume mutt + mbsync + notmuch and a single Gmail account at `~/Mail/gmail/`; generalisation to other layouts is direct.

```
# 1. Stop any running mbsync. If a launchd job runs mbsync on a
#    schedule, bootout it temporarily.
pkill -f mbsync || true
launchctl bootout "gui/$UID/local.mbsync" 2>/dev/null || true

# 2. Cheap insurance: dump the notmuch tag database. Tags are user
#    state; the message index can be rebuilt, the tags cannot.
notmuch dump --output="$HOME/notmuch-dump-$(date +%Y%m%d).txt"

# 3. Move the Maildir AND .notmuch together. They must move
#    atomically to keep the Xapian database's relative paths valid.
mkdir -p ~/.local/share/mail
mv ~/Mail/gmail   ~/.local/share/mail/gmail
mv ~/Mail/.notmuch ~/.local/share/mail/.notmuch

# 4. Update the dotfiles repo (Layer 1, single source of truth)
#    BEFORE editing any live config under $HOME. Edit:
#       ~/dotfiles/editors/notmuch-config         (path = $HOME/.local/share/mail)
#       ~/dotfiles/config/mutt/muttrc             (folder, spoolfile, postponed, record)
#       ~/dotfiles/mbsyncrc                        (Path, Inbox under each MaildirStore)
#    Commit and push. (The dotfiles repo is OUTSIDE cloud-mounted
#    paths per Layer 1, so in-place editors are safe to use here.)

# 5. Reflect the dotfiles changes into $HOME (idempotent).
cd ~/dotfiles && ./install.sh

# 6. Verify.
mbsync -a                 # Resumes incrementally; should NOT re-download.
notmuch search '*' | head  # Confirms the index resolves message paths.
mutt -e 'exit'            # Opens and exits cleanly with the new paths.

# 7. Re-enable the scheduled mbsync once verification passes.
launchctl bootstrap "gui/$UID" \
    ~/Library/LaunchAgents/local.mbsync.plist
```

Two points worth flagging:

- The notmuch Xapian database stores message paths relative to `database.path`. Moving the Maildir and `.notmuch` together preserves those relative paths, so `notmuch new` is not required. If the two directories are moved separately for any reason, the index becomes invalid and `notmuch new` is needed to rebuild it.
- Step 4 (update the dotfiles repo first) is the step the obvious version of this recipe omits. Editing the live `~/.config/mutt/muttrc` directly before updating the repo copy makes the two diverge. A second laptop bootstrapped from the repo today would deploy a muttrc that points at a Maildir that does not exist. Always edit the repo file, then `install.sh`.

The pre-XDG location (`~/Mail/`) can be deleted after a few days of confirmed working state; keep it around briefly as a rollback target.

Layer 3: Append-only history files

This layer is small in bytes but disproportionate in friction. The relevant files:

- `~/.zsh_history` (shell command history)
- `~/.viminfo` (vim's last-used files, registers, marks)
- `~/.claude/history.jsonl` (Claude Code prompt history)
- Editor undo files (`~/.local/state/vim/undo/`)
- AI-tool histories from any other tools (`~/.gemini/`, etc.)

Each is append-only at the application level and whole-file at the sync layer. That mismatch is what produces conflict copies. There are three architecturally honest choices:

1. **Per-machine state.** Do not sync these files at all. Accept that arrow-key history on one laptop does not include commands typed on another. Simplest; works for most users; what most non-Dropbox dotfile setups assume.
2. **Single-machine writes.** Designate one machine as the canonical owner; the others write to local files that are never synced. Practical only if the user genuinely uses one machine 95% of the time.
3. **Append-aware sync.** A few sync mechanisms (atuin for shell history, custom JSONL mergers for AI logs) understand append semantics and can merge two divergent streams. More moving parts, but eliminates the conflict-copy class entirely.

The right choice depends on the user. Most readers will be happiest with option 1 plus, optionally, atuin for shell history because it is genuinely useful across machines and well-engineered for the multi-machine case.

The Decision Worksheet

The companion deliverable for this post is a Quarto document at `docs/migration-decision-worksheet.qmd`. It walks the reader through:

1. **Inventory.** List every sync-relevant artefact in `$HOME` and `~/Dropbox/`. Classify each into one of the three layers.

2. **Mechanism selection.** For each layer, pick a sync mechanism (or 'no sync, per-machine') from the table above.
3. **Migration order.** Sequence the moves so the backup pipeline never points at a stale path. The general rule is: dotfiles first (already done if post 24 was followed), then a single project as a pilot, then the rest of project content, then history files.
4. **Verification.** Bootstrap a clean user account or VM and confirm the full workflow comes up without Dropbox.

The worksheet is generic; it does not assume any particular project structure or sync mechanism. Completing it for a given setup converts the migration into a series of small, audited moves rather than a single large rewrite.

Migration Sequence

The order of operations matters because the backup pipeline must remain valid at every intermediate step. The sequence below comes from running this migration on a real workflow.

1. **Confirm dotfiles foundation.** Verify post 24's repository works. Run `install.sh --dry-run` on a clean account or VM. Resolve any failures before proceeding.
2. **Introduce a path-abstraction env var.** Add `PRJ_ROOT` to `~/.zshenv` with a fallback that resolves to `~/Dropbox/prj` for now. Replace literal `~/Dropbox/prj` references in scripts and configs with `$PRJ_ROOT`. Nothing visible changes, but the next move is now a single-line edit.
3. **Pilot one project.** Pick a single small repo. Move it from `~/Dropbox/prj/<project>/` to `~/prj/<project>/` (real path, outside cloud). Push to a private git remote if it is not already. Confirm the workflow on it.
4. **Migrate the rest of project content in batches.** A batch a day prevents the backup pipeline from going stale. For each batch: move, repoint any in-script paths, push to remotes, verify.
5. **Repoint backup sources.** Once `$PRJ_ROOT` resolves to the new path, edit `backup-gdrive` and `backup-icloud` (or their equivalents) to source from `$PRJ_ROOT` instead of the literal Dropbox path. The launchd plists themselves do not change; only the scripts they call.
6. **Resolve append-only history.** Pick option 1, 2, or 3 from the previous section. Apply per-file. Document the choice in the dotfiles repo's README for future reference.
7. **Verify on a clean account.** Boot a fresh user, clone dotfiles, run installer, set up project remotes, confirm the workflow. The first surprise reveals the assumption that was hiding.
8. **Optional: uninstall Dropbox.** Most users do not need to actually remove Dropbox. Once the workflow is independent, Dropbox can stay installed as a backup or for sharing files with colleagues, without being load-bearing for the workflow itself.

The sequence is robust to interruption. Stopping after step 3 leaves a working pilot with the backup pipeline intact. Stopping after step 5 leaves the workflow independent; the history-file decision can be deferred.

Figure 3: Two open-top boxes side by side on a workbench, the left box full of mixed objects and the right box divided into three sorted compartments, with a single object mid-transfer between them.

Things to Watch Out For

Seven gotchas to anticipate during the migration. Each lists the symptom and the fix.

1. **The backup pipeline goes stale silently.** Symptom: `backup-gdrive` runs every night, exits 0, but the syncthing folder it backs up has not changed in a week because the source path no longer exists. Fix: `rclone sync` exits 0 even when the source is empty, so add an explicit `[[ -d $PRJ_ROOT ]]` guard at the top of every backup script and fail loud on missing source.
2. **Cloud-mounted git working trees corrupt their indexes.** Symptom: `git status` reports phantom changes; `git fsck` finds dangling refs; the working tree is inconsistent. Fix: never put `~/dotfiles/` inside `~/Dropbox/`, `~/Library/CloudStorage/...`, or `~/Library/Mobile Documents/`. Git's frequent writes to `.git/index` race with the cloud provider's sync engine. The repo MUST be on a non-synced filesystem path.
3. **Symlinks to Dropbox break asymmetrically across machines.** Symptom: `~/.config/git -> ~/Library/CloudStorage/Dropbox/dotfiles/config/git` on machine A, but on machine B the same symlink is dangling because Dropbox is at a different path or not installed. Fix: symlinks created by `install.sh` should target `$DOTFILES` (an env var resolved at install time), not literal Dropbox paths.
4. **Append-only history conflict copies do not announce themselves.** Symptom: a file named `history (machine's conflicted copy YYYY-MM-DD).jsonl` appears next to the canonical

file, with no notification, no error, no merge prompt. Fix: a weekly grep for 'conflicted copy' across the Dropbox tree, or an explicit migration of the affected files out of cloud sync.

5. **launchd plist substitutions overwrite themselves.** Symptom: an edit to `~/dotfiles/launchd/local.backup.research.plist` does not change the running job, because `install.sh` sed-substitutes `__USER__` to `$USER` when writing to `~/Library/LaunchAgents/`, and the active plist is the substituted copy. Fix: do not edit the active plist directly; edit the source in `~/dotfiles/launchd/`, then re-run `install.sh` plus `launchctl bootout` and `launchctl bootstrap`.
6. **Project content includes runtime artefacts that should not move.** Symptom: a 200 MB `renv/library/` directory comes along with a project move and now lives at `~/prj/<project>/renv/library/`, where renv expects to manage it. Fix: confirm `.gitignore` excludes the right paths before pushing; add `**/renv/library/` to the dotfiles repo's `.gitignore` if one is accidentally tracked.
7. **In-place editors race with cloud sync and truncate files to zero bytes.** Symptom: `sed -i.bak file`, `awk -i inplace`, or `perl -i` against a path under `~/Library/CloudStorage/...` returns 0 but leaves the file empty. The `.bak` saved by `-i.bak` is also placed in cloud storage and is subject to the same race; on the incident that produced this gotcha, both the file and its backup were lost locally and required Dropbox version-history restore. Fix: never run `-i`-style editors on cloud-mounted paths. Either use the Edit tool / a real editor, or copy to `/tmp`, edit, and `mv` back. (If the dotfiles repo is on a non-synced path per gotcha 2, this hazard is confined to scripts that touch `~/Library/CloudStorage/...` from elsewhere.)

Uninstall / Rollback

The most common 'undo' is not 'reinstall Dropbox' but 'put project content back under Dropbox while keeping dotfiles independent'. The architecture supports this directly because the layers are independent.

To roll project content back into Dropbox:

1. Move `~/prj/<project>/` back to `~/Dropbox/prj/<project>/`.
2. Update `~/.zshenv` so `PRJ_ROOT` resolves to the Dropbox path.
3. The dotfiles repo, backup scripts, and launchd plists do not need any change because they reference `$PRJ_ROOT`.

This is also the recommended path for a reader who completed the dotfiles work but is not yet ready to migrate project content. Set up the env var, leave the path pointing at Dropbox, defer the project move until later.

To roll back the entire migration: the dotfiles installer's `--dry-run` should never have shipped destructively, so the rollback is just `rm -rf ~/dotfiles` plus restoring `$HOME` from a Time Machine backup. Most readers will not need this.

Figure 4: A closed wooden chest with three small brass clasps on a stone surface, an open notebook and a capped fountain pen beside it, suggesting a deliberate and completed reorganisation.

What Did We Learn?

Lessons Learnt

Conceptual:

- The 'sync everything via one cloud provider' default conflates three independent concerns. Separating them makes each easier to reason about.
- Migration sequencing matters more than mechanism choice. The wrong mechanism is correctable; a half-migrated workflow is hard to leave for a year.
- Per-machine state is a feature, not a bug, for many file types. The right answer for shell history is often 'do not sync', not 'sync better'.

Technical:

- A single environment variable (`$PRJ_ROOT`) provides enough indirection to migrate path references in two passes: first introduce the variable while it still resolves to Dropbox, then change what it resolves to.
- launchd plists with `__USER__` placeholders deploy cleanly across machines via `install.sh` sed-substitution; bootstrapping the substituted copy avoids machine-specific files in the dotfiles repo.
- Atomic git operations on a non-synced filesystem are reliable; the same operations on a cloud-synced filesystem race with the sync engine and produce subtle corruption.

Gotchas:

- 'rclone sync exits 0' does not mean the sync succeeded; it can mean the source had nothing to send because the path is wrong.
- A symlink created on one machine resolves correctly on that machine but may dangle on another. Always test the new-machine case.
- Conflict copies of append-only files accumulate silently. Periodic auditing is the only way to notice.

Limitations

- The migration described here covers a single-user workflow. Team setups (where the same Dropbox folder is shared with collaborators) need additional considerations not addressed in this post.
- Project content moved from Dropbox to per-project git remotes is now subject to GitHub or GitLab quotas, including LFS limits for binary files. Plan accordingly.
- The decision worksheet is a generic framework; it does not generate scripts for any particular sync mechanism. Each reader still has to write the rclone or rsync commands for their setup.
- Append-aware history sync (option 3) requires either an existing tool (atuin works for shell history) or hand-rolled mergers for proprietary log formats. The post does not provide those mergers.
- Migrating off Dropbox does not, by itself, improve the security posture of credentials. Files like `~/.aws/credentials` should be addressed separately; cloud sync was not their only exposure.
- The post assumes macOS or Linux. Windows-specific path conventions and sync semantics are out of scope.

Opportunities for Improvement

1. Extend the decision worksheet into a small CLI tool that audits a current setup and generates a draft migration plan.
2. Build an atuin (or atuin-equivalent) integration template for the dotfiles repo, so option 3 from layer 3 becomes a one-line opt-in.
3. Add a `verify.sh` to the dotfiles repo that runs the end-to-end check (dotfiles installed, `$PRJ_ROOT` valid, backup scripts pointing at it, launchd jobs loaded) on demand and reports pass/fail.
4. Document a canonical 'two-laptop' setup with Syncthing replacing the Dropbox role for shared project content.
5. Generalise the worksheet's mechanism table into a maintained list (a small repo or a wiki page) that adds new tools as the ecosystem evolves (Filen, S3-backed, etc.).

Wrapping Up

A portable dotfiles repository is the first half of a Dropbox-free workflow; this post is the framework for the second half. The key idea: stop treating sync as a single dimension and start treating it

as three layers (configuration, project content, append-only history) with different semantics and different right answers.

Most of the work is in the inventory and decision step, not in the moves themselves. Once each artefact is classified and each layer has a chosen mechanism, the actual migration is a sequence of small file moves and a few text edits. The pieces that take time are testing the new-machine bootstrap, deciding what to do about history files, and resisting the temptation to migrate everything in one weekend.

Expected outcomes

The architecture aims to deliver three measurable end states. The first is verifiable now (Layer 1 is complete on the author's primary laptop); the other two will be confirmed and updated with measured numbers when Layer 2 and Layer 3 land.

- **New-laptop setup** is `git clone … ~/dotfiles && ./install.sh && (per-project git clones)`. The plan budget is roughly 30 minutes from clean machine to working environment; the actual figure will be added after a fresh-VM bootstrap is run.
- **Conflict copies of `history.jsonl` and similar files** should stop recurring once the affected files are moved out of Dropbox sync. The post will be updated with the post-migration count once Layer 3 is applied.
- **Backup pipelines** will source from `$PRJ_ROOT`, which can point anywhere; once Layer 2 lands, Dropbox is no longer load-bearing for the workflow.

In conclusion, four points merit emphasis. First, the dotfiles layer is the wedge, not the goal; once it is in place, the rest of the migration is mechanical. Second, 'project content' is two layers in disguise (code, which wants git, and data, which wants something else); the two should be separated before a mechanism is chosen. Third, some of what looks like 'data' is actually a local cache of an authoritative source elsewhere (Maildir from IMAP, Homebrew cache, language-server indexes); for these the right answer is not to sync at all, but to rebuild on each machine from the canonical source. Fourth, append-only files are the only category that benefits from sophisticated sync; for everything else, simpler is better.

See Also

- Post 24, Creating a GitHub Dotfiles Repository — the prerequisite. This post is its sequel.
- Post 65, Multi-Laptop Bootstrap: Dotfiles Repository — the next post in this series; implements the dotfiles layer with `install.sh`, `Makefile`, and a sensitive-files inventory.
- Post 66, Setting Up pass, the Unix Password Manager — covers GPG key generation, pass initialisation, and migrating credentials identified in this post's Layer 1 inventory.
- Post 67, Multi-Laptop Security: Hardening the Bootstrap — a security audit of the full infrastructure established across posts 64-66.
- Post 20, Setting Up a Comprehensive Research Backup System on macOS — three-tier backup architecture; most of the relevant launchd-plist patterns originate there.
- Post 22, Launching AWS EC2 Instances with Bash Scripts and the AWS CLI — the worked example of `__USER__` substitution and CLI-driven deployment.

- `MULTI_LAPTOP_PLAN.md` (in the dotfiles repository) — the live action plan tracking this migration, with done-marks and effort estimates for each step. `MIGRATION_NOTES.md` in the same repository captures what was copied and what was deliberately excluded during the Layer 1 move.
- Syncthing documentation — peer-to-peer sync, the most-recommended Dropbox replacement for project content.
- Atuin — the cleanest answer to 'I want my shell history across machines'; an example of append-aware sync done right.
- The XDG Base Directory Specification — the underlying convention that determines where each XDG-aware tool actually puts its files; relevant when choosing between `~/.cache`, `~/.local/share`, and `~/.local/state`.

Reproducibility

Tested on:

Component	Version
OS	macOS 15.4 (Sequoia)
Shell	zsh 5.9
git	2.45
rclone	1.68
Syncthing	1.27
flock (homebrew)	2.40
Date verified	2026-05-07

The decision worksheet is a generic Quarto document and does not depend on any particular tool versions; the table above captures the environment in which the framework was developed and validated.

Let's Connect

I would enjoy hearing from you if:

- You have made the same migration with different sync-mechanism choices and want to compare notes.
- You see a gotcha I missed, especially in the append-only-history layer.
- You use Linux or Windows and want to extend the worksheet to your environment.
- You just want to say hello and connect.

Rendered on 2026-05-08 at 06:10 PDT. Source: ~/Dropbox/prj/qblog/posts/64-migrating-off-dropbox/migrating-off-dropbox/analysis/report/index.qmd

Related posts in this cluster

This post is part of the *Security, Backup, and Sync* series. Recommended reading order:

1. Post 31: Research Backup Architecture
2. **Post 32: Migrating Off Dropbox: Beyond Dotfiles** (this post)
3. Post 33: Setting Up pass: a Unix Password Manager
4. Post 34: Secrets Management for the Workflow Construct
5. Post 35: Security Foundations for a Multi-Laptop Research Cluster