

Object Storage for Research Data: The Layer Git Cannot Hold

Ronald G. Thomas

2026-09-11



A repository never edits a box; it accessions a new one, and the old number still means what it always meant.

Introduction

Leaving a sync daemon settles where code goes: into git repositories. It does not settle where research data goes. The usual answer, some mixture of a shared folder and a habit of copying, carries forward the property that made the sync daemon painful in the first place: the same mutable file existing in several places at once, with conflict copies, silent overwrites, and no way to say which version produced a given table.

i Note

Part 7 of the security series; start with [Security Foundations for a Multi-Machine Research Fleet](#) for the overview.

This post argues that the question is malformed. Asking how to synchronize research data presumes the data should be mutable and replicated. An object store declines that premise: data is written once under a key, never modified, and fetched on demand. Nothing has to agree with anything, because nothing changes. A laptop stops being a replica and becomes a cache.

The post is a design argument with a tested worked example, not a deployment report. No object store runs on the fleet's home server today. A section below states what the fleet does instead and what would justify the change.

Motivations

- Every sync mechanism compared in [the post on leaving Dropbox](#) is one-way, conflict-prone, quota-limited, or all three when applied to data.
- A reproducible compendium pins its code, environment, and package versions, then says nothing machine-checkable about which bytes of data it consumed.
- Students arriving in a lab need a rule for where data goes that does not depend on remembering to copy something.
- Restricted data under a data use agreement needs access that can be revoked centrally, which copies on laptops cannot offer.

Objectives

1. Explain what an object store is, separately from the file-sharing services it superficially resembles.
2. Give a manifest-and-fetch pattern small enough to adopt in an afternoon, tested end to end.
3. Show where the pattern fits in a zzcollab compendium, including the hash-linked provenance chain from a rendered report to the exact bytes, where that chain is weak, and whether it is a blockchain.
4. Give a tested setup for a self-hosted store that enforces immutability, and the operational conditions for running it on a home server.
5. Compare costs and say when the pattern is the wrong shape for the work.



What is an Object Store?

An object store is not a folder, and the most common way to misuse one is to mount it and treat it as though it were.

The model is a flat namespace of immutable blobs. You **put** an object under a key and **get** it back by that key; no operation edits it in place. The interface is HTTP rather than a filesystem: no partial writes, no locking, no rename, and no directories except as a convention about slashes in key names.

Two API families implement the model. Amazon S3 is the commercially dominant one, and its interface is the de facto standard that SeaweedFS, Garage, Ceph, Backblaze, and Wasabi all speak. OpenStack Swift is older and still common in academic and national-laboratory deployments, where a bucket is called a container. The architecture is identical; only the wire protocol and tooling differ, which matters when an institutional service turns out to speak Swift.

Three properties follow from immutability, and they carry the whole argument:

- **No conflict, because no concurrent modification.** A corrected extract is not an edit; it is a new object under a new key.
- **A key plus a content hash is a durable citation.** If an analysis records the hash of what it read, the claim ‘this table came from that data’ is checkable years later without trusting a filename.
- **Access is centrally governed.** Revoking one credential removes access everywhere at once. No folder-sharing arrangement can promise that, because copies have already left.

What it is not

It is not a backup system. A mistaken write under a new key is still a mistaken write, and lifecycle rules that expire old versions will expire the one you wanted. Backup stays a separate concern; see [the backup post](#).

It is not a filesystem. Tools that make it look like one (`s3fs` and its relatives) reintroduce every problem the model removes. Fetch explicitly.

It is not, on its own, a data management plan. It is a substrate that makes a good plan expressible.

When a Store Earns Its Keep

The fleet described in this series does not run an object store. Research data that is too large or too restricted for git lives in a plain directory on [the home server](#), and workstations fetch from it. Code is in git; working copies of projects are protected by Borg, cloud copies, and Time Machine.

That arrangement is adequate while one person produces and consumes each dataset. It has none of the properties argued for here: no versioning, no write-once enforcement, and no read-only identity that a collaborator could be given. The manifest pattern below does not need a store to be useful; it was tested against a plain directory exposed as an `rclone` remote, and it adds verification and provenance there too. What a store adds is enforcement on the server side.

The deployment is justified when any of the following becomes true:

- **A second analyst needs the canonical dataset.** At that point ‘everyone agrees not to overwrite it’ stops being a control.
- **A data use agreement requires revocable access** or audited custody that copies on laptops cannot provide.
- **A result must be defended later** by showing the exact input bytes, which needs both the manifest’s hash and a store that cannot destroy the version it names.
- **Datasets outgrow the plain directory’s informal discipline:** many versions, several projects reading the same extract, or derived data expensive enough to publish rather than regenerate.

Until then, adopting the manifest and the read-through helper costs an afternoon and moves nothing.

Prerequisites

- A project laid out as a `zzcollab` compendium, or any layout with a raw-data directory excluded from git.
- `rclone` (tested with 1.75), R with the `digest` and `tinytest` packages, `bash`, and `shasum`.
- For the store walkthrough: `SeaweedFS` (tested with 4.46 and 4.47) and the AWS CLI, on a machine you control.
- Credentials kept in `pass` and loaded on demand, as described in [the secrets post](#).

The Pattern

The pattern is four moves: data leaves the project directory, a manifest takes its place in git, a build step fetches exactly what the manifest names and checks it, and the analysis refuses to read anything the manifest does not vouch for.

Every listing in this section is read at render time from a small working example stored beside this post ([analysis/scripts/manifest-pattern/](#)). Two test scripts in the same folder ran it end to end, one against a local directory and one against an S3 endpoint with object lock; the results are tabulated at the end of the section. The layout follows a [zzcollab compendium](#) ([analysis/data/raw_data/](#), [analysis/scripts/](#), [R/](#), [inst/tinytest/](#)), for reasons the next section gives.

Move 1: the manifest

The compendium tracks a small tab-separated file naming each dataset, where it lives, its content hash, and the terms under which it may be used. The example ships with only the header row; the tests publish into it. Populated, it looks like this (the codebook row is illustrative):

```
# analysis/data/manifest.tsv
name          remote  container  object      sha256      terms
trial_extract_v3 lab     trial     ext_v3.csv  77cad0...   synthetic
codebook_v3   lab     trial     cb_v3.csv   41cd88...   synthetic
```

The location is written as `remote`, `container`, and `object` rather than as one `s3://` URL on purpose. A protocol scheme inside the key bakes one vendor into a file meant to outlive the storage choice. The `remote` name resolves through the client's configuration, which is where protocol details belong.

Two columns carry the argument. The `sha256` is the dataset's identity; the key only says where to look, and a later section shows why that distinction matters. The `terms` column puts the license or data use restriction on the same line as the hash, so it travels into every provenance record the analysis writes.

The file is a few hundred bytes, diffs cleanly, and reviews in a pull request. It is append-only in practice: a corrected extract is published under a new name and key, which adds a row, and the analysis moves to it by changing the one line that names the dataset. Both halves appear in the same pull request.

Move 2: fetch what the manifest names, and nothing else

The fetch belongs in the build, not in anyone's memory. The Makefile is deliberately thin; each data target hands off to one script that reads the manifest.

What matters about the script is what each subcommand refuses to do.

- **fetch takes every key from the manifest**, so changing the manifest changes what is downloaded.

Listing 1 Makefile

```
SHELL := /bin/bash
.SHELLFLAGS := -eu -c

RAW_DATA_DIR ?= analysis/data/raw_data
export RAW_DATA_DIR

.PHONY: fetch-data verify-data publish-data analysis report test

fetch-data: ## Pull every manifest dataset not already cached, verified
    bash tools/data-manifest.sh fetch

verify-data: ## Check local copies against the manifest's sha256 column
    bash tools/data-manifest.sh verify

# make publish-data NAME=trial_extract_v4 FILE=ext_v4.csv \
#   CONTAINER=trial OBJECT=ext_v4.csv TERMS='DUA 2026-014'
publish-data: ## Upload a new dataset and append its manifest row
    bash tools/data-manifest.sh publish \
        "$$(NAME)" "$$(FILE)" "$$(CONTAINER)" "$$(OBJECT)" "$$(TERMS)"

analysis: fetch-data
    Rscript analysis/scripts/01_prepare_data.R

report: analysis ## Render the report and register its hash
    Rscript analysis/scripts/02_report.R

test:
    Rscript -e "tinytest::run_test_dir('inst/tinytest')"
```

-
- **A verified local copy is a cache hit.** If the file is present and its hash matches, nothing is transferred.
 - **Bytes that fail the hash are deleted before anything can read them**, and the target fails, naming the dataset.
 - **verify** derives its expectations from the same **sha256** column, so there is no second checksum file to drift.
 - **publish** hashes the bytes being uploaded, **at write time**, refuses a name already in the manifest or a key already in the store, reads the object back and hashes it again, and only then appends the row.

i Full listing: `tools/data-manifest.sh`

Listing 2 tools/data-manifest.sh

```
#!/bin/bash
# data-manifest.sh -- fetch, verify, and publish datasets named in
# analysis/data/manifest.tsv. The manifest is the single source of truth:
# the key says where to fetch from, the sha256 says what must arrive.
#
#   data-manifest.sh verify
#   data-manifest.sh fetch
#   data-manifest.sh publish NAME FILE CONTAINER OBJECT [TERMS]
set -euo pipefail
```

```
MANIFEST=${MANIFEST:-analysis/data/manifest.tsv}
RAW_DATA_DIR=${RAW_DATA_DIR:-analysis/data/raw_data}
REMOTE=${DATA_REMOTE:-lab}
```

```
sha256_of() { shasum -a 256 "$1" | cut -d' ' -f1; }
```

```
# Rows only: skip the header and comment lines.
rows() { grep -v '^#' "$MANIFEST" | tail -n +2; }
```

```
local_path() { printf '%s/%s/%s' "$RAW_DATA_DIR" "$1" "$2"; }
```

```
verify_one() {
    local name=$1 path=$2 want=$3 got
    if [ ! -f "$path" ]; then
        echo "MISSING $name ($path; run 'make fetch-data')"
        return 1
    fi
    got=$(sha256_of "$path")
    if [ "$got" != "$want" ]; then
        echo "FAILED $name expected ${want:0:12} got ${got:0:12}"
        return 1
    fi
    echo "OK $name ${want:0:12}"
}
```

```
cmd_verify() {
    local bad=0 name remote container object sha terms
    while IFS=$'\t' read -r name remote container object sha terms; do
        verify_one "$name" "${local_path "$container" "$object")" "$sha" \
            || bad=$((bad + 1))
    done < <(rows)
```

`rclone` is the client because the remote is named `lab` and nothing in the script says which protocol `lab` speaks. `rclone` has native backends for S3, Swift, and roughly forty others, so the same build step works against a self-hosted gateway, an institutional Swift service, a commercial bucket, or a plain directory. The two tests make the point: the same unmodified script ran against a local directory and against an S3 endpoint, and only the definition of `lab` differed. Credentials for either backend can be supplied through environment variables from `pass` rather than written to `rclone.conf`.

Move 3: read only through the manifest

Verification in the build protects the build. It does not protect a script run directly from the console, which is how most analysis scripts are run. So the analysis never opens a raw file by path; it asks for a dataset by name.

Listing 3 analysis/scripts/01_prepare_data.R

```
source(here::here('R', 'data_manifest.R'))

trial <- read.csv(use_dataset('trial_extract_v3'))
stopifnot(nrow(trial) > 0, all(c('arm', 'outcome') %in% names(trial)))

by_arm <- trial |>
  aggregate(outcome ~ arm, data = _, FUN = mean)

derived <- here::here('analysis', 'data', 'derived_data')
dir.create(derived, showWarnings = FALSE, recursive = TRUE)
write.csv(by_arm, file.path(derived, 'outcome_by_arm.csv'),
  row.names = FALSE)
write_provenance(file.path(derived, 'outcome_by_arm.provenance.tsv'))
```

`use_dataset()` looks the name up in the manifest, hashes the local copy, and returns the path only if the two agree, recording the read (name, key, hash, terms, time). `write_provenance()` writes that record beside the derived output, with the full commit hash of the code and a flag saying whether tracked files had uncommitted changes. From the test run, with the `remote`, `terms`, and `read_at` columns omitted and hashes shortened:

name	key	sha256	commit	tree_dirty
trial_extract_v3	trial/ext_v3.csv	77cad0...	4adfedc...	FALSE

i Full listing: `R/data_manifest.R`

The two layers are independent, and the tests exercised both. With a tampered local copy, `make analysis` repaired the cache: the fetch step saw the mismatch, downloaded the object again, and the analysis ran on the right bytes. Running the script directly, with no fetch in front of it, stopped at the read (hashes abbreviated):

```
Error: dataset 'trial_extract_v3' does not match the manifest
  expected 77cad029cb5b0245c9df258f6898d3f9...
  found    b485c3f0a918e6175abbd21a1515d43a...
refusing to read it; re-run 'make fetch-data'
```

Move 4: ignore the bytes

```
# .gitignore
analysis/data/raw_data/
```

The repository carries the manifest and never the data. A fresh clone plus `make fetch-data` reconstitutes the project, and deleting `raw_data/` to reclaim disk is safe.



What this buys

No machine is authoritative and nothing needs reconciling. Two people working from the same manifest work from bit-identical inputs, and their provenance files prove it. The question behind the whole reproducibility apparatus, which data produced Table 1, has an answer recoverable from the commit and checkable against the bytes.

How it was tested

Both test scripts copy the example to a temporary directory, so running them never modifies it. `tests/run-local.sh` uses an `reclone` alias remote pointing at a local directory. `tests/run-s3.sh` starts SeaweedFS 4.47 on the loopback interface with per-run credentials for a publisher and a read-only identity, and a bucket under object lock in **COMPLIANCE** mode with a one-day retention. Both ran on 2026-09-28 on macOS arm64.

Scenario	Expected	Observed
Publish a dataset	Row appended, hash from uploaded bytes	As expected
Fetch with no local data	Downloaded and verified	As expected
Fetch again	Cache hit, no transfer	As expected
Analysis via <code>make</code>	Reads via <code>use_dataset()</code> , writes provenance	As expected
Local copy tampered, <code>verify-data</code>	Fails, names dataset	As expected
Same, script run directly	Refuses to read, shows both hashes	As expected
Same, via <code>make analysis</code>	Re-fetches, heals the cache	As expected
Remote object replaced under same key	Fetch fails, bad file deleted	As expected
Publish an existing name or key	Refused	As expected
Read-only identity overwrites a key (S3)	AccessDenied	As expected
Publisher overwrites a locked key (S3)	New version accepted; fetch fails on hash	As expected; see ‘A lock is not a pin’
Delete the original locked version (S3)	AccessDenied ; original recoverable	As expected
Uncommitted edit to tracked code	Provenance records <code>tree_dirty = TRUE</code>	As expected
Render a report	Register row: output hash, commit, inputs	As expected
Look the report up by its bytes	Row found	As expected
Edit the report, look it up again	No row matches	As expected
Signed commit, throwaway key	<code>verify-commit</code> passes; unsigned fails	As expected
Unit tests (<code>inst/tinytest/</code>)	All pass	14 of 14

Reproducibility in a zzcollab Compendium

The pattern is not tied to a template, but it was written for one. `zzcollab`, the framework behind most analyses on this blog, organizes a compendium around five artifacts, the Five Pillars, and the pattern repairs the weakest of them.

The weak pillar

The five are the `Dockerfile`, `renv.lock`, `.Rprofile`, the source code, and the research data. Four are pinned by something a machine can check: the image is built from the `Dockerfile` and a team project pulls it by the digest in `.team-image-digest`; package versions are pinned in `renv.lock`; `.Rprofile` and code are identified exactly by a commit.

The fifth is pinned by convention. `zzcollab`'s replication-crisis vignette says the raw data directory 'is read-only by convention'. Nothing records which bytes a commit expects there, and nothing stops them from changing. A compendium can rebuild its three-year-old environment exactly and still produce a different Table 1, because the extract was replaced by a corrected one with the same filename, while every other pillar reports, truthfully, that nothing changed.

What `zzcollab` already does, and where it stops

The `zzcollab` Makefile ships two data-integrity targets. `make hash-data` writes the sha256 of every file under `analysis/data/raw_data/` to `data-manifest.sha256`, and `make verify-data` checks against it. Committed, that file makes silent mutation detectable. It stops at detection:

- **It records what is present, not where it came from.** Hashes are taken from whatever is on disk when the target runs.
- **A fresh clone cannot act on it.** It says what should be there, not how to get it.
- **It covers the directory, not the analysis.** It cannot say which files a script read, or which commit read them.

The manifest here is the same idea with location and terms added, and with those, verification becomes reconstitution. Its natural home is the existing target pair rather than a parallel system; see Opportunities.

A hash-linked provenance chain from a report to the bytes

A compendium cannot record a chain of custody in the forensic sense, which documents who held an item and when. It can record a sequence of records in which each carries the hash of the next, so a change anywhere is detectable. That is strong evidence the bytes were not altered and weak evidence of who held them or when. The links:

1. **The rendered output is registered by its hash.** A document cannot contain its own hash, so `record_render()` appends a row to a register beside it: the output's sha256, the full commit hash, the dirty-tree flag, and the hashes of the raw data it was computed from. `zzcollab`'s render stamp prints a source path, time, and `git describe` label on every page for human readers; the register is what a machine can check.
2. **The commit names the manifest.** A commit hash covers its whole tree.
3. **The manifest names the bytes.**
4. **Derived outputs name what was actually read,** through their provenance files; the report reads those, so its register row can name raw-data hashes rather than an intermediate CSV.
5. **The store keeps the bytes.** Object lock prevents the named version from being deleted during retention.

Working backward starts from the bytes in hand:

```
source('R/data_manifest.R')
verify_render('analysis/report/share/outcome_report.txt')
# -> the register row: output hash, full commit, tree_dirty, inputs
```

```
git clone <project> && cd <project>
git checkout <commit from that row>
make report # fetches the named data, verifies, re-renders
```

The test ran exactly this walk: a fresh clone at the registered commit fetched, verified, re-rendered, and produced a report whose sha256 matched the register row. An edited report fails the lookup, because the lookup is by content; an altered object fails `make fetch-data`, which names the dataset rather than rerunning on something else.

Links 3 and 4 look redundant and are not. The manifest records what a commit intended to read; the provenance file records what was read. They diverge when someone runs a script with uncommitted changes, and the divergence is visible because `tree_dirty` is recorded.

Where the chain is weak

- **The register is a file.** It binds a document's bytes to a commit, but it is trustworthy only as far as it is committed (in a later commit, since a commit cannot name itself) and the history protecting it is trustworthy.
- **Abbreviations are not identifiers.** Seven hexadecimal characters find a commit today and do not bind one; the example records forty.
- **A commit does not pin a dirty tree.** The flag makes this visible; the policy that follows is not to publish from a dirty tree.
- **Git hashes with SHA-1 by default.** Since 2.13 git uses a hardened SHA-1 that detects the SHAttered collision technique, and repositories can use SHA-256 (`git init --object-format=sha256`) if the hosting service supports it. See the [hash-function transition document](#).
- **Who and when are self-asserted.** Authors and dates come from the committer's configuration and clock, and history can be rewritten. Signing commits and tags binds them to a key, so `git verify-commit` can say who; signing does not establish when.
- **A hash can leak small data.** The hash of a small, predictable table can be reversed by enumeration. Do not publish hashes of low-variety restricted data.

Is this a blockchain?

Structurally it is a relative. Chaining hashes so records cannot be altered unnoticed goes back to Haber and Stornetta, '[How to time-stamp a digital document](#)' (Journal of Cryptology, 1991), widely credited as the first blockchain design. Git history is the same kind of object: each commit hash covers its tree and parents. The Software Heritage archive uses the structure to give research software [intrinsic, content-derived identifiers](#), which in my reading are the closest published analog to the manifest here.

What a blockchain adds is agreement among many mutually untrusting writers and, on a public chain, a record of time no single party controls. Neither is needed here. Wüst and Gervais, ‘[Do you Need a Blockchain?](#)’ (2018), conclude a blockchain is warranted only when several mutually untrusting parties must write shared state and no trusted third party is available. A lab publishing its own datasets has one writer and trusted parties within reach: its institution, a timestamp authority, a registry.

The research literature arguing otherwise is thinner than its visibility suggests. According to PubMed, an early proof of concept for blockchain-timestamped trial protocols, Irving and Holden (2016), [was retracted](#) by its authors in 2017 after concerns raised in post-publication review ([original](#)). Nugent and colleagues (2016) extended the idea to smart contracts ([DOI](#)), and Benchoufi and Ravaud (2017) argued the case in a commentary ([DOI](#)). A 2026 systematic review in JMIR Medical Informatics found ten qualifying studies of smart contracts in clinical trials, all prototypes or simulations, none tested in a real regulatory setting, with conflicts with the GDPR’s right to erasure among the common gaps ([Sheikh et al.](#)).

Something beyond this chain does add value for a dated claim that must convince a skeptical outsider, such as proof that an analysis plan existed before a result. Three tools address it without a private blockchain (none was tested for this post):

- **A timestamp authority** ([RFC 3161](#)) signs a statement that a hash existed at a given time.
- **Transparency logs** such as [Certificate Transparency](#) (RFC 6962) and Sigstore’s Rekor ([Newman, Meyers, and Torres-Arias, 2022](#)) give public verifiability with no consensus protocol.
- **OpenTimestamps** records an aggregate of many hashes in a Bitcoin transaction, proving a commit existed before a block without putting data on the chain; `git-ots` applies it to git.

For describing provenance rather than protecting it, [W3C PROV](#) is the standard vocabulary, and it complements the hashes rather than replacing them.

Several analysts, one dataset

zzcollab’s vignette proposes multi-analyst scaffolding, a `multiverse/` directory with one subdirectory per team, following the crowdsourced protocol of Silberzahn and colleagues (2018), whose first stage is a shared, documented dataset. The pattern makes that stage enforceable. Each team gets read-only credentials, so no team can alter the canonical dataset (the test confirms a read-only identity is refused an overwrite), and the claim that twenty-nine teams analyzed the same data becomes a check that twenty-nine provenance files contain the same hash.

Derived data, containers, and CI

- **Derived data is regenerated, not published.** Raw data goes to the store and the manifest; derived data stays in `analysis/data/derived_data/`, identified by its provenance file. The exception is a derived dataset that is expensive to recompute or depends on a licensed source the build cannot reach: publish it as though it were raw, with its provenance file beside it.

- **Fetch on the host, read in the container.** `rcclone` and the store credentials stay on the host, where `pass` supplies them; the container reads the verified cache through the project mount, and credentials never enter the shared image.
- **The verifier is a dependency.** `use_dataset()` hashes with the `digest` package, which must be in `renv.lock`; paths resolve with `here::here()`, so the cache is found on the host and in the container alike.
- **CI checks structure, not data.** A store on a home network is invisible to a hosted runner, and for restricted data it should be. CI can run the unit tests, which use a synthetic file, and validate manifest rows; the full rebuild is verified on machines entitled to hold the data, which means someone has to actually do it.

Where this stops

- **A hash identifies bytes, not meaning.** A codebook revision that reinterprets a column leaves every hash intact; publish codebooks as datasets in their own right.
- **Provenance is per script, not per table.** Finer lineage is what pipeline tools such as `targets` add.
- **The manifest is hand-rolled on purpose**, so the mechanism is visible. DVC and `git-annex` automate the same idea with more to learn.

Running Your Own: SeaweedFS, Step by Step

A self-hosted store buys the one property consumer services cannot: the server refuses the overwrite, rather than everyone agreeing not to attempt it. The walkthrough below was run on a laptop against a loopback endpoint; the next section covers what changes on a home server.

Choosing the software

The recommendation changed twice during testing, and the path is the part a reader is most likely to repeat.

MinIO is the name most people arrive with. Its open-source upstream is archived; Homebrew marks `minio` and `minio-mc` deprecated with removal scheduled during 2027. The binaries run but receive no security patches.

Garage is what most published comparisons recommend for a single machine: maintained, S3-compatible, designed for hardware outside a datacenter, and modest with memory. It does not enforce immutability. On 2.4.1, `PutBucketVersioning` returned `501 NotImplemented`, and a 1 KB upload over a 5 MB object under the same key silently replaced it. It is also not simpler to operate: nine steps to an authenticated endpoint (configuration file with generated secrets, node id, cluster layout with zone and capacity, layout apply, then bucket, key, and permission commands) against three for SeaweedFS. Where Garage plausibly wins is memory, since `weed server` runs master, volume, filer, and S3 gateway in one process. That was not measured; on a 4 GB machine it could decide the question, on the 16 GB home server it is less likely to.

SeaweedFS enforces write-once. The rest of this section is the setup verified on 4.46.

Step 1: install

```
brew install seaweedfs
```

On Debian-derived Linux, take the release binary from the project's GitHub rather than the distribution package, which tends to lag. (The Linux install was not tested.)

Step 2: start the server

```
mkdir -p ~/seaweed/data
weed server \
  -dir=$HOME/seaweed/data \
  -s3 -s3.port=8333 \
  -ip=127.0.0.1 \
  -volume.max=50 \
  -master.volumeSizeLimitMB=256
```

-s3 turns on the S3 gateway. Bind to 127.0.0.1 while testing so nothing is reachable before authentication is configured.

-volume.max is not optional. SeaweedFS preallocates volumes per bucket and the default runs out once a second bucket exists. The symptom is an opaque `InternalError` on upload, with `failed to find writable volumes for collection` visible only in the server log.

Allow about fifteen seconds. The S3 gateway comes up well after the process starts, and an early client reports what looks like a configuration fault.

Step 3: create a bucket with object lock

Lock must be enabled **at bucket creation**; it cannot be retrofitted.

```
export AWS_ACCESS_KEY_ID=... AWS_SECRET_ACCESS_KEY=...
EP="--endpoint-url http://127.0.0.1:8333"

aws $EP s3api create-bucket --bucket lab-data \
  --object-lock-enabled-for-bucket

aws $EP s3api put-object-lock-configuration --bucket lab-data \
  --object-lock-configuration \
  '{"ObjectLockEnabled":"Enabled",
   "Rule":{"DefaultRetention":{"Mode":"COMPLIANCE","Days":365}}}'
```

Use `COMPLIANCE`. Under `GOVERNANCE` a privileged caller can override retention, which returns you to immutability by permission. Choose the window deliberately: every version is stored that long and the period cannot be shortened.

Step 4: point rclone at it

```
rclone config create lab s3 \  
  provider=Other \  
  access_key_id=... secret_access_key=... \  
  endpoint=http://127.0.0.1:8333 \  
  region=us-east-1
```

make `fetch-data` now works unchanged against `lab:lab-data`, which is the payoff of keeping protocol out of the manifest. In practice, leave the key fields empty in the file and supply them from `pass` through rclone's environment variables at invocation.

Step 5: verify the lock actually holds

Accepting a lock configuration and enforcing it are different things, and the difference is what disqualified Garage.

```
aws $EP s3api put-object --bucket lab-data \  
  --key trial/ext_v1.parquet --body ext_v1.parquet  
  
VID=$(aws $EP s3api list-object-versions --bucket lab-data \  
  --prefix trial/ext_v1.parquet \  
  --query 'Versions[0].VersionId' --output text)  
  
# Both of these must fail.  
aws $EP s3api delete-object --bucket lab-data \  
  --key trial/ext_v1.parquet --version-id "$VID"  
  
aws $EP s3api put-object-retention --bucket lab-data \  
  --key trial/ext_v1.parquet --version-id "$VID" \  
  --retention \  
  '{"Mode":"COMPLIANCE","RetainUntilDate":"2020-01-01T00:00:00Z"}'
```

On 4.46 the first returns `AccessDenied` and the second `InvalidRequest`, and the object survives with its size and version id unchanged.

Use a version id, or you will misread the result

Deleting **without** a version id does something different and legitimate. On 4.46, `delete-object` with no version id returns `"DeleteMarker": true` and succeeds, and `list-objects-v2` then shows nothing. But `list-object-versions` still shows the protected version, and reading it by version id returns bytes whose sha256 matches the original. This is standard S3 behavior, and almost certainly what lies behind tracker issue [#8350](#), ‘Object Lock COMPLIANCE mode does not enforce WORM (Delete still succeeds)’, closed as ‘not planned’. Object lock protects versions; listings are a separate question.

A lock is not a pin

Object lock prevents a version from being destroyed. It does not freeze what a key resolves to. In a versioned bucket, an upload to an existing key creates a new current version and leaves the old one protected, and a fetch by key returns whichever is current.

Tested on SeaweedFS 4.47 with the manifest pattern running against it: the publisher identity uploaded a 15-byte file over the 2,571-byte extract under the same key, and the locked bucket accepted it. A fetch by key returned the 15 bytes and the manifest hash refused them. The original version was still retrievable by version id, still matched the manifest, and still could not be deleted.

Three mechanisms each do one job:

- **Object lock** stops a version from being destroyed.
- **A read-only identity** stops most clients from writing at all; the read-only test identity was refused the same overwrite.
- **The manifest hash** is the only one that notices a key now pointing at different bytes.

The publisher identity remains the gap, since it can always create a new version under an old key. Keep it on one machine, use it only through the publish step, which refuses existing keys, and let the hash rather than the key decide what counts as the right file.

Verify it again after a restart

A store that forgets its locks on reboot is worse than one that never claimed them, because you would trust it. After stopping and restarting the server, the bucket lock configuration, the protected version, its `RetainUntilDate`, the refusal to delete by version id, and the bytes were all intact. Run that check once on your own hardware before the store holds anything you care about.

Step 6: authentication

Started plainly, the S3 gateway answers unauthenticated requests with 200; started with `-s3.config`, it answers 403. A minimal configuration with one publisher and one read-only client:

```
{
  "identities": [
    {
      "name": "publisher",
      "credentials": [{"accessKey": "...", "secretKey": "..."}],
      "actions": ["Admin", "Read", "Write", "List", "Tagging"]
    },
    {
      "name": "laptop-ro",
      "credentials": [{"accessKey": "...", "secretKey": "..."}],
      "actions": ["Read", "List"]
    }
  ]
}
```

```
]
}
```

Pass it with `-s3.config=/path/to/s3.json`. Tested on 4.46 as `laptop-ro`: listing and reading succeeded; creating an object, overwriting a key, planting a delete marker, and creating a bucket were each refused with `AccessDenied`. The delete-marker refusal matters: a read-only client cannot hide an object, which closes the gap the version-id discussion opens. The lock stops destruction; the identity stops mischief.

Operating It on the Home Server

The walkthrough yields an authenticated, lock-enforcing store on a loopback interface. Moving it onto **the home server** is not tested, but the services already running there (Borg repositories, a Samba Time Machine share, a default-deny firewall) set the conditions it would have to meet. Each item below is a practice established for those services and transferred by analogy.

- **Run it under the service manager and make it wait for the network.** The `-ip=127.0.0.1` from Step 2 must change, and any service bound to a specific interface inherits a failure the Samba share on that server exhibited: started before its adapter had an address, it listened only on loopback and never retried, while every status check said it was running. A systemd unit for the store should declare `Wants=network-online.target` and `After=network-online.target`, and a post-reboot check should confirm what the port actually listens on (`ss -tln`), not merely that the process exists. The home-server post covers this in detail.
- **Address it by its Tailscale address.** The fleet's scripts reach the server by its Tailscale address, because its LAN address can change at a reboot; when it did, clients that named the old address failed quietly for several nights (**the backup post** tells that story). An `endpoint=` in `rclone.conf` is exactly such a hardcoded address. The same choice covers access from campus or travel without any port forwarding.
- **Firewall first.** The server denies incoming connections by default, trusts the Tailscale interface, and admits specific ports from the LAN only. The S3 port should join that list deliberately, before the store leaves loopback.
- **Credentials in pass, one identity per role.** The publisher identity lives on one workstation; every other machine gets a read-only identity. This mirrors the restricted backup keys that limit each Borg client to its own repository.
- **A retention policy you have thought about.** Compliance mode plus a careless default fills a disk with undeletable versions of a file someone regenerates nightly.
- **Durability comes from elsewhere.** One disk with excellent locks is still one disk, and object lock protects against deletion, not a dead SSD. The store's data directory belongs in the server's own Borg backup and in an offsite replica (see the costs section).
- **Monitor it.** Every failure in the fleet's backup jobs so far was found by reading logs and exit statuses by hand, sometimes days late, while the jobs looked healthy. A store holding the canonical copy of a dataset deserves the same daily check proposed in the backup post: up, reachable from a client, and still refusing a version delete.

What self-hosting still does not provide is uptime guarantees (a repurposed laptop sleeps, loses power, and gets moved) or anyone to call. Upgrades, disk health, key rotation, and restore tests are the operator's job.

What It All Costs

Self-hosting is free at the margin, so the comparison is what you would otherwise pay, and on which axis.

Hot-tier commercial storage runs roughly \$18 per TB per month for Azure Blob, \$20 for Google Cloud Storage Standard, and \$23 for Amazon S3 Standard. The differentiator is egress: S3 charges about \$0.09 per GB, so pulling one terabyte adds roughly \$90. A pattern built on fetching on demand stresses exactly that axis.

Backblaze B2 (about \$6.95 per TB per month) and Wasabi (about \$7.99) include free egress within generous limits, and Cloudflare R2 (\$15) charges none. All three are S3-compatible, and B2 and Wasabi support Object Lock. So **if enforced immutability is the goal, the cheapest route is not self-hosting**; it is a \$7 per TB bucket at a provider that implements Object Lock. Self-hosting buys LAN speed, no quotas, no per-request accounting, and control of a machine already owned.

Consumer sync products (Google Drive, OneDrive, Dropbox) are mutable by design, with no content-addressed retrieval or version semantics a manifest can rest on, and API quotas a build step will feel. They are fine as a durable copy, which is how the fleet uses them for project directories, and wrong as the store the pattern addresses.

Institutional storage, where offered, is worth pricing first. It often carries custody and retention properties a data management plan needs, and sometimes favorable treatment on grant-funded work. Ask whether the endpoint speaks S3 or Swift before designing around it.

The arrangement I would build

A SeaweedFS instance on the home server as the working store, with `rclone sync` to an offsite replica at a provider that also supports Object Lock, B2 or Wasabi being the obvious candidates at around \$7 per TB. Fast fetches over the Tailscale network, enforced write-once, no marginal cost for daily work, and a replica that covers the single-disk hole. Whether lock semantics survive the sync is an open question (see Limitations); the destination bucket most likely needs its own default retention.

Because `rclone` speaks all of these and the manifest names none of them, the replica is a decision that can be deferred and revisited. The storage recommendation moved three times while this post was tested, and the build step never changed.



Things to Watch Out For

1. **Assuming S3.** Symptom: an `s3://` path and access-key stanza copied from a tutorial fail against a campus service that speaks Swift, and the authentication error is really a protocol mismatch. Fix: read the provider's own documentation, and keep protocol out of the manifest.
2. **Mounting the store.** Symptom: someone installs `s3fs`, and the lab is back to a shared folder with worse latency. Fix: make the fetch step the only sanctioned access path.
3. **A manifest with no hash.** Symptom: 'the file is present' is the only check the build can make. Fix: record the hash when the object is written, as the publish step does, not when it is first read.
4. **Mutable keys.** Symptom: a corrected file is uploaded under the old key and every fetch by key returns the new bytes. Versioning is not the fix; it keeps the old version but makes the new one current. Fix: put the version in the key, refuse existing keys at publish time, and let the manifest hash decide.
5. **Credentials in `rc1one.conf`.** Symptom: an access key sits in plain text in the home directory, the exposure `the secrets post` exists to prevent. Fix: inject from `pass` at invocation.
6. **A hardcoded LAN endpoint.** Symptom: fetches fail after the server reboots with a new LAN address. Fix: use the Tailscale address in every client's configuration.
7. **Egress and quota surprises.** Symptom: a cross-validation loop re-fetches a 40 GB object every iteration. Fix: cache under an ignored path and make the fetch idempotent.
8. **Assuming the license travels with the bytes.** Symptom: data under redistribution limits lands where a future collaborator gets blanket access. Fix: record terms in the manifest so they appear in every provenance file.
9. **The exploratory phase.** Symptom: a weekly revised spreadsheet is minted as a versioned object each time. Fix: keep a scratch area with loose rules and promote to the store when a dataset is stable enough to cite.

What Did We Learn?

Lessons Learned

- **Test the property, not the documentation.** Two of three candidate stores failed on inspection: one archived, one silently replacing locked data. The only reliable test was trying to delete a protected version and watching the server refuse.
- **A lock is not a pin.** Object lock kept the original extract safe, but only the manifest hash noticed when the key began pointing at different bytes.
- **Detection is not reconstitution.** A committed checksum file says data changed; a manifest with location and terms lets a fresh clone rebuild the project with the right bytes or refuse.
- **A label is not a binding.** A version string in a footer identifies nothing; a register keyed by the output's hash does.
- **Operating a store is mostly operating a server.** The conditions in 'Operating It on the Home Server' came from services that have nothing to do with object storage, and they would apply unchanged.

Limitations

- The pattern assumes datasets large relative to git's comfort and stable enough to version. Small, rapidly churning tables are better committed directly.
- Object storage is poor at partial reads of row-oriented formats; Parquet makes column projection workable, a 40 GB CSV does not.
- Someone has to administer the store and rotate credentials, which is real overhead for a single-investigator lab.
- Nothing is deployed. The SeaweedFS walkthrough ran on macOS arm64 on loopback; service supervision, Tailscale access, Linux, and the memory footprint were not tested, and the home-server operating advice is transferred from other services.
- The manifest pattern was tested with one synthetic 200-row CSV: no transfer timings, multi-gigabyte objects, many datasets, or Swift backend.
- The publisher identity can always create a new version under an existing key. The publish step refuses and the hash catches a bypass, but nothing on the server prevents it.
- The render register was exercised with a plain-text report. A PDF engine that embeds a creation time will not re-render byte-identically, so for PDFs the walk backward verifies inputs and code rather than reproducing the file.
- Commit signing was tested with a throwaway key; no trusted timestamp was tested. The chain establishes integrity and, with signing, who, but not when.
- Whether object lock survives `rclone sync` to a replica is untested and doubtful; retention metadata is not obviously among what `rclone` copies. Treat the replica's immutability as a hypothesis.
- Commercial prices are September 2026 list rates from secondary sources; treat them as orders of magnitude.

Opportunities for Improvement

1. Fold the pattern into `zzcollab`: `hash-data` and `verify-data` become `publish-data`, `fetch-data`, and a manifest-reading `verify-data`, with the manifest header, `.gitignore` stanza, and `R/data_manifest.R` in the scaffold.
2. Move `use_dataset()`, `write_provenance()`, `record_render()`, and `verify_render()` into `zzcollab` with their tests, and have the render stamp call `record_render()`, so each staged report's row carries its own sha256, the full commit, and input hashes.
3. Record the object version id in the manifest and fetch by it, so a new version under an old key is never downloaded at all.
4. Sign release commits and tags, and add an RFC 3161 timestamp or OpenTimestamps proof where a date must convince an outsider.
5. Deploy and measure on the home server when one of the triggers in 'When a Store Earns Its Keep' is met: systemd supervision, access over Tailscale, memory footprint, transfer times, and the replica leg including lock behavior.
6. Compare DVC and git-annex against the hand-rolled manifest.
7. A companion piece on institutional storage, the option many data management plans actually require.

Wrapping Up

The sync question for research data is better dissolved than answered. Code belongs in git. Mutable copies of data on several machines are the source of the problem, and an append-only store addressed by content hash removes them while adding the provenance a compendium otherwise lacks. The cost is an explicit fetch step and an administrator; the return is that every analysis can say exactly what it read.

For a `zzcollab` compendium the gain is specific: the data pillar becomes checkable like the other four, and a hash-linked chain runs from a rendered report through its commit to the exact bytes. That chain shares ancestry with blockchains without needing to be one; it proves the bytes were not altered, and who and when are jobs for signatures and timestamping services.

Takeaways:

- Adopt the manifest and the read-through helper now; they work against a plain directory and need no store.
- Deploy a store when a second analyst, a revocable-access requirement, or a result that must be defended appears.
- Choose software by trying to delete a locked version, not by reading comparisons.
- Let the hash, not the key, decide what counts as the right file.
- Run the store under the same discipline as the fleet's other services: network-aware startup, Tailscale addressing, firewall first, credentials in `pass`, and a daily check.

See Also

Security series:

1. [Security Foundations for a Multi-Machine Research Fleet](#)
2. [Setting Up pass: a Unix Password Manager](#)
3. [Secrets Management for the Workflow Construct](#)
4. [Migrating Off Dropbox: Beyond Dotfiles](#)
5. [Research Backup Architecture: Borg, Cloud Copies, and Time Machine](#)
6. [A Home Server for a Five-Machine Research Fleet](#)
7. [Object Storage for Research Data: The Layer Git Cannot Hold \(this post\)](#)
8. [Running the Quarterly Security Audit for the First Time: What Actually Broke](#)

External references:

- [Reproducible Blog Posts with ZZCOLLAB](#): the Five Pillars.
- [A pocket terminal with tttyd and Tailscale](#): the same overlay-network argument for a different service.
- [SeaweedFS](#) and [Garage](#).

Reproducibility

Store behavior. SeaweedFS object lock, persistence across a restart, scoped-identity enforcement, the delete-marker semantics behind issue #8350, the Garage versioning result, and the MinIO deprecation were verified on 2026-09-11 on macOS arm64, with integrity checked by sha256 on retrieved bytes. Versions: SeaweedFS 4.46, Garage 2.4.1, rclone 1.75.1, aws-cli against a local endpoint.

The manifest pattern. The example in `analysis/scripts/manifest-pattern/` was run end to end on 2026-09-28 by `tests/run-local.sh` and `tests/run-s3.sh` (SeaweedFS 4.47, object lock in COMPLIANCE mode, publisher and read-only identities), with results in ‘How it was tested’; the 14 `tinytest` assertions in `inst/tinytest/` pass. Listings in The Pattern are read from those files at render time. The render register, dirty-tree flag, byte-identical walk backward, and commit signing (throwaway key in a temporary keyring) were tested the same day. The dataset is synthetic, generated with a fixed seed. `zzcollab` details were read from its source and templates, not executed for this post.

The literature. Sources in ‘Is this a blockchain?’ were checked for existence and for the claims attributed to them through publisher or standards pages and, for biomedical papers, PubMed records. Full texts were not read.

What was not done. Both SeaweedFS instances were temporary and have been torn down. No service supervision, Tailscale access, Linux run, replica leg, or memory measurement. The home-server operating practices come from Samba, Borg, and firewall configuration on that machine, not from a store.

AI involvement in this post: drafting was AI-assisted, and the tool survey and testing were conducted with assistance. The architecture argument, the decision to keep protocol details out of the manifest, the decision to treat a self-hosted store as a first-class option, and the editorial judgment are the author’s.

Session information:

```
R version 4.6.1 (2026-06-24)
Platform: aarch64-apple-darwin25.4.0
Running under: macOS Tahoe 26.6.2

Matrix products: default
BLAS:   /opt/homebrew/Cellar/openblas/0.3.34/lib/libopenblas-r0.3.34.dylib
LAPACK: /opt/homebrew/Cellar/r/4.6.1/lib/R/lib/libRlapack.dylib;  LAPACK version 3.12.1

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/Los_Angeles
tzcode source: internal

attached base packages:
[1] stats      graphics  grDevices  utils      datasets  methods    base

loaded via a namespace (and not attached):
[1] compiler_4.6.1  fastmap_1.2.0   cli_3.6.6      tools_4.6.1
[5] htmltools_0.5.9 parallel_4.6.1  ote1_0.2.0     yaml_2.3.12
[9] rmarkdown_2.31  knitr_1.51      jsonlite_2.0.0 xfun_0.58
[13] digest_0.6.39   rlang_1.3.0     evaluate_1.0.5
```

Let's Connect

Questions, suggestions, or corrections are welcome.

- **GitHub:** [rgt47](#)
- **Twitter/X:** [@rgt47](#)
- **LinkedIn:** [Ronald Glenn Thomas](#)
- **Email:** [Contact form](#)

I would like to hear from you if:

- You find an error, or a better approach to anything in this series.
- You run a small research fleet and have found where this design breaks.
- You want to compare notes on backups, secrets, or reproducible research infrastructure.