Backup and Restore Course

Flavio Copes

Updated August 3, 2026

Contents

Design a recovery plan, create encrypted versioned backups with restic, protect databases and keys, and prove every backup through restoration.

What you'll learn: Automate encrypted backups with clear retention and offsite copies, then restore files and an application in a timed recovery rehearsal.

Plan recovery

Inventory data, define recovery targets, apply 3-2-1, and model loss and compromise.

Inventory what must survive

List unique data, reproducible software, configuration, credentials, metadata, owners, and locations.

A backup plan starts with what cannot be recreated. Not with tools, not with schedules. If you don't know what must survive, you'll back up the wrong things and discover it during a disaster.

The key distinction is **unique data** versus **reproducible data**. Your operating system is reproducible: you can reinstall it. Your application code is reproducible if it's pushed to Git. But an unpushed repository, a production database, user-uploaded files, and encryption keys exist in exactly one place. Lose them and they're gone.

Build the inventory

Create a plain inventory without secret values. One entry per thing that matters:

```
item: customer database
owner: application team
location: production PostgreSQL
changes: continuously
backup method: logical dump plus provider snapshot
restore dependency: database version and encryption key
```

Every field earns its place. **Owner** tells you who to call when the restore fails. **Changes** tells you how often to back it up. **Restore dependency** is the one people forget: a database dump is useless without a compatible database version, and an encrypted backup is useless without its key.

Now walk through every device, server, cloud service, and personal folder that contains unique data. Your laptop's `~/Documents`. The VPS running your side project. The SaaS tool that holds your invoices. For each one, ask: if this disappeared right now, could I recreate it?

```
item: family photos
owner: me
location: laptop ~/Pictures plus phone
changes: weekly
backup method: none yet
restore dependency: nothing
```

An honest `backup method: none yet` entry is the whole point of this exercise. It turns a vague worry into a task.

What to watch out for

Mark unknown ownership as a problem. Data that belongs to nobody gets backed up by nobody.

Keep credentials out of the inventory itself. This document will be copied, printed, and shared during an incident, so it must be safe to circulate. Record *where* protected recovery material is held instead: ”restic password: in the password manager emergency kit”. That pointer is useful during recovery. The actual secret in a plain text file is a liability.

Define RPO and RTO

Choose acceptable data loss and recovery time for each workload before choosing tools or schedules.

Two numbers drive every backup decision, and you should write them down before touching any tool.

The **Recovery Point Objective (RPO)** is how much recent data you can afford to lose. If you back up nightly and the disk dies at 11 PM, you lose a day of work. Your actual RPO is 24 hours, whether you chose that or not.

The **Recovery Time Objective (RTO)** is how long recovery may take. From "it's broken" to "it works again". That includes finding credentials, downloading data, and verifying the result — not just the restore command.

Set targets per dataset

Different data deserves different numbers. Write targets for two different datasets:

```
family photos: RPO 1 day, RTO 3 days
customer orders: RPO 5 minutes, RTO 1 hour
```

Losing a day of photos is sad but survivable, and nobody needs them back within the hour. Losing five minutes of paid orders costs real money and trust, so that dataset needs frequent backups and a fast, rehearsed restore.

Each number maps to a concrete mechanism. Connect RPO to backup frequency and replication: an RPO of 5 minutes means continuous archiving or streaming replication, because no nightly dump can meet it. Connect RTO to restore speed, staffing, instructions, and replacement infrastructure: a 1 hour RTO means a written runbook and somewhere ready to restore to.

Do the bandwidth math

RTO promises die on transfer speed. Check yours:

```
# 500 GB over a 100 Mbit/s connection:
# 500 * 8 / 0.1 / 3600 = ~11 hours, before verification
```

If your data lives in remote object storage and your office uplink gives you 100 Mbit/s, a "restore within 2 hours" target is fiction. Either keep a local copy too, or change the target to match reality.

One warning: replication alone can copy deletions and corruption immediately. A replica gives you a great RPO against hardware failure and nothing against `DROP TABLE`. It does not automatically meet historical recovery needs — only versioned backups do that.

Apply the 3-2-1 pattern

Place multiple copies across distinct storage and one offsite boundary without counting synchronized replicas incorrectly.

The **3-2-1 rule** is the classic baseline: keep three copies of important data, on two kinds of storage or failure domains, with one copy offsite.

Each number defends against a different disaster. Three copies means one failure never leaves you with a single point of truth. Two storage types means a bad disk batch or a filesystem bug can't take everything. One offsite copy means a fire, flood, or burglary at one location doesn't end the story.

Map one dataset

Take a real dataset and write down where its copies live:

```
copy 1: live laptop data
copy 2: versioned backup on external disk
copy 3: encrypted versioned backup in remote object storage
```

Note that the live data counts as copy one. You only need two backups to satisfy the rule, but they must be genuinely independent.

Now name the failure each copy survives: accidental deletion, disk loss, device theft, account compromise, or site disaster. This is the verification step for a backup *plan*. If two copies fail for the same reason, they count as one:

```
accidental deletion  -> external disk (has history)
laptop disk death    -> external disk, object storage
house fire           -> object storage only
cloud account hacked -> external disk only
```

Every threat should have at least one surviving copy. If a row comes up empty, you've found the gap before the disaster did.

The counting mistake

The most common error is counting a synchronized mirror as a backup. A mounted mirror is vulnerable to the same deletion, malware, and operator mistakes as the live data. Delete a file, and Dropbox, RAID, or an always-connected `rsync` target deletes it too — instantly and faithfully. Same for ransomware: it encrypts the mirror right along with the source.

What makes a copy a real backup is **version history** and some **independence** from the live system: an offline disk, separate credentials, or snapshots the source machine can't overwrite. Independence matters more than count alone.

Threat-model the backup

Plan for deletion, corruption, ransomware, stolen credentials, lost keys, provider failure, and operator mistakes.

Backups are valuable targets. Modern ransomware crews go for them first, because a victim with working backups doesn't pay. So the backup system itself needs a threat model, the same way the live system does.

The core question: **who can delete or read my backups, and from where?**

Start with the worst realistic case. An attacker with the same credentials may delete both live data and every recovery point. If your server holds a credential that can write *and* delete backups, then anyone who owns the server owns your backups too.

Build a threat table

Create a threat table, one row per scenario:

```
threat: compromised server
path: backup credentials readable by root
control: append-only or separate deletion credentials
```

```
recovery test: restore from protected snapshot
```

The **path** column forces honesty: exactly how would this threat reach the backups? The **control** column names the defense. For a compromised server, the standard controls are append-only storage (the server can add snapshots but never remove them) or split credentials, where the delete permission lives on a separate administrator account the server never sees.

Now add the quieter threats. Each gets its own row:

```
threat: accidental deletion       control: version history, retention
threat: silent corruption          control: integrity checks, checksums
threat: lost backup password       control: escrowed key, second keyholder
threat: expired cloud account      control: billing alerts, second provider
threat: unavailable restore tool   control: documented versions, offline copy
threat: legal retention needs      control: written policy, immutable storage
```

The unglamorous rows matter most. Far more backups die from an expired credit card or a forgotten password than from attackers.

No single control covers everything

Encryption protects confidentiality but not deletion — an attacker doesn't need to read your backups to destroy them. Immutability protects history but not a lost decryption key: your snapshots can sit safe and unreadable forever. Every control in the table answers one threat and ignores the others. That's why you need the whole table, not one favorite defense.

Check your understanding: plan recovery

Check the commands, decisions, safety boundaries, and failure cases from plan recovery.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

File backup tools

Create archives, copy trees, preserve metadata, verify checksums, and distinguish synchronization from backup.

Create and extract an archive

Package a directory with tar, inspect it before extraction, and restore into a separate destination.

An **archive** combines files and metadata into one stream. That single file is easy to copy, checksum, and store, which is why `tar` has been the backbone of Unix backups for decades. Compression changes size, not whether the archive is an independent backup — a compressed archive on the same disk as the original protects you from nothing.

Create, inspect, extract

Create and restore a practice archive:

```
tar -czf notes.tar.gz notes/
tar -tzf notes.tar.gz
```

```
mkdir restored
tar -xzf notes.tar.gz -C restored
```

The flags read naturally once you decode them: `-c` create, `-z` gzip compression, `-f` the archive filename, `-t` list contents, `-x` extract, `-C` change to a directory before extracting.

The `-t` listing is your inspection step. It prints every path the archive contains:

```
notes/
notes/report.txt
notes/meetings/2026-07-14.md
```

Always look at this before extracting anything. You want to see relative paths under a single top-level directory, so extraction can't scatter files around or overwrite something outside your target.

Extracting with `-C restored` keeps the restore away from the live `notes/` directory. That's a habit worth keeping: restore to a separate destination, verify, then decide what to move.

Verify the round trip

Compare source and restored files:

```
diff -r notes/ restored/notes/
# no output means identical content
```

Silence is the success signal here. Repeat the exercise with names containing spaces, symbolic links, and nested directories — these are exactly the cases where a naive backup script breaks, and `tar` handles them all.

The failure mode

Inspect archives from untrusted sources before extraction. Paths and links can target unexpected locations: an archive built maliciously can contain entries like `../../home/flavio/.bashrc` or a symlink pointing outside the extraction directory. Modern GNU tar refuses absolute paths by default and warns about suspicious members, but the `-t` check costs you two seconds and removes the guesswork. Never extract straight into `/` or your home directory from an archive you didn't create.

Copy a tree with rsync

Create an incremental file copy, preview changes, preserve metadata, and avoid accidental deletion.

rsync efficiently updates a destination tree. On the first run it copies everything. On every run after that, it transfers only what changed, which makes it ideal for pushing a working directory to an external disk or a remote server.

One thing to be clear about up front: by itself, one synchronized destination is not version history. rsync gives you the current state, mirrored. Yesterday's version of a file is gone from the mirror the moment you sync today's.

Preview, then copy

Preview before copying:

```
rsync --archive --dry-run --itemize-changes notes/ /Volumes/Backup/notes/
rsync --archive --itemize-changes notes/ /Volumes/Backup/notes/
```

`--archive` (or `-a`) preserves permissions, timestamps, symlinks, and ownership where possible — you almost always want it. `--dry-run` shows what *would* happen without touching anything. `--itemize-changes` prints one line per affected file:

```
>f+++++++++ report.txt
>f.st...... budget.md
```

`>f+++++++++` means a new file is being created. `>f.st......` means an existing file is being updated because its size and modification time differ.

Now modify, add, and remove a source file. Run the dry-run again and explain every proposed change before running the real command. When a second dry-run prints nothing, source and destination are in sync — that silence is your verification.

Trailing slashes and --delete

Be extremely careful with source trailing slashes and `--delete`.

The trailing slash changes meaning: `notes/` means "the contents of notes", while `notes` means "the directory itself". Get it wrong and you end up with `/Volumes/Backup/notes/notes/`, and your next sync compares the wrong trees.

`--delete` removes destination files that no longer exist in the source. It's what keeps a mirror honest, but it turns rsync into something that destroys data. A reversed source and destination can destroy the wanted copy: sync an empty new laptop *to* your backup disk with `--delete`, and the backup is now empty too. Always dry-run any command that includes `--delete`, every time.

Preserve required metadata

Decide whether ownership, permissions, timestamps, links, extended attributes, and application metadata must survive.

File bytes may be insufficient for recovery. A restored web server with the right content but the wrong ownership won't serve pages. A restored script without its executable bit won't run. Executable bits, ownership, links, access controls, and extended attributes can affect service behavior just as much as the content does.

For personal documents, bytes are usually enough. For a system restore, metadata is part of the data.

Capture a baseline

Before you can verify metadata survived, record what it looked like. Capture a metadata baseline:

```
stat notes/report.txt
getfacl notes/report.txt 2>/dev/null || true
ls -l@ notes/report.txt 2>/dev/null || true
```

`stat` shows ownership, mode, and the three timestamps. `getfacl` shows POSIX ACLs on Linux. `ls -l@` shows extended attributes on macOS. The `|| true` keeps the commands from failing on platforms where they don't exist — this baseline script should run anywhere.

Save that output next to your restore documentation. After a restore, run the same commands and compare:

```
stat restored/notes/report.txt
```

```
# compare Uid, Gid, Access mode with the baseline
```

Know what your tools preserve

Each tool has limits. `rsync --archive` preserves permissions, times, and symlinks, but needs the extra `-A` flag for ACLs and `-X` for extended attributes. `tar` stores ownership in the archive, but only restores it when you extract as root — as a regular user, everything becomes yours. Restore to a filesystem that supports the needed metadata and compare. Document any platform conversion you accept.

Where this bites

Do not assume a cloud sync tool or FAT-formatted disk preserves Unix ownership and permissions. FAT and exFAT — the default format on most external drives — have no concept of Unix owners or modes. Copy `/etc` to an exFAT disk and back, and every file returns owned by you with mangled permissions. Dropbox-style sync clients keep content and quietly discard the rest.

The fix is either a properly formatted destination (ext4, APFS) or an archive format like `tar` that carries the metadata inside the file itself.

Verify files with checksums

Generate and check cryptographic hashes to detect unexpected changes during storage or transfer.

A **checksum** fingerprints file bytes. Run a file through SHA-256 and you get a short hex string. Change one byte anywhere in the file, and the string comes out completely different. That property lets you detect corruption during storage or transfer — but only if you saved the expected hash beforehand. It detects change when the expected hash is protected and compared later.

This matters for backups because storage lies quietly. Disks develop bad sectors, transfers get interrupted, and a backup archive can rot for two years before you try to open it.

Create and verify a manifest

Generate the hash right after creating the archive, while you know it's good:

```
shasum -a 256 notes.tar.gz > notes.tar.gz.sha256
shasum -a 256 -c notes.tar.gz.sha256
```

The first command writes a manifest file containing the hash and the filename. The second reads that manifest and re-hashes the file. Success looks like this:

```
notes.tar.gz: OK
```

On Linux, `sha256sum` works identically. Run the `-c` check after every copy of the archive to a new destination, and again periodically on cold storage.

Prove the check works

Never trust a verification you've never seen fail. Change one byte in a disposable copy and confirm verification fails:

```
cp notes.tar.gz broken.tar.gz
printf 'x' | dd of=broken.tar.gz bs=1 seek=100 conv=notrunc
shasum -a 256 -c notes.tar.gz.sha256
```

```
# after renaming broken over the original:
# notes.tar.gz: FAILED
# shasum: WARNING: 1 computed checksum did NOT match
```

The non-zero exit status means you can script this and alert on failure.

Limits worth knowing

Store the expected manifest separately when tampering is in scope. An attacker who can modify the archive can regenerate the `.sha256` file sitting next to it in the same directory.

Timing matters too: a checksum taken from an already-corrupted file just fingerprints the corruption. Hash at creation time, not later.

And a checksum does not create another copy and does not tell you which version is correct by itself. It tells you *that* bytes changed, never *which* side is right. Detection, not recovery.

Check your understanding: file backup tools

Check the commands, decisions, safety boundaries, and failure cases from file backup tools.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Encrypted versioned backups

Create a restic repository, make snapshots, exclude disposable data, and inspect stored history.

Initialize a restic repository

Create an encrypted local practice repository and protect the password separately from the stored backup.

restic stores encrypted, deduplicated snapshots in a **repository** — a directory structure it fully controls, on a local disk, an SSH server, or object storage like S3. Everything inside is encrypted before it leaves your machine, so the storage provider never sees your data. The repository password is required to recover its keys and data.

Encryption plus versioning is what separates restic from plain `tar` and `rsync` copies: you get history, and the destination doesn't need to be trusted.

Create a practice repository

Initialize a disposable local repository:

```
mkdir restic-lab
export RESTIC_REPOSITORY=$PWD/restic-lab
restic init
```

restic prompts for a new password, then confirms:

```
created restic repository 3f8a91c2d4 at /home/flavio/restic-lab
Please note that knowledge of your password is required to access
the repository. Losing your password means that your data is
```

```
irrecoverably lost.
```

That warning is literal, and we'll come back to it.

The `RESTIC_REPOSITORY` environment variable saves you from repeating `-r /path` on every command. For scripts, add `RESTIC_PASSWORD_FILE` pointing at a root-readable file, so the password never appears in the command line or shell history:

```
export RESTIC_PASSWORD_FILE=$HOME/.config/restic/lab-password
```

Verify it opens

Record the password in a protected temporary lab location, then use `restic snapshots` to confirm the empty repository opens:

```
restic snapshots
```

```
repository 3f8a91c2 opened (version 2)
```

An empty snapshot list with no error is exactly right. You've proven the password works *before* trusting the repository with data. Peek inside `restic-lab/` too: you'll see `config`, `keys/`, and `data/` directories full of opaque encrypted files. Nothing in there is readable without the password — that's the point.

The unforgiving part

If the password and key material are lost, encryption works against you too. There is no recovery mechanism, no support ticket, no brute-force shortcut. A perfectly intact repository becomes permanent noise.

So plan protected recovery before real backups: store the password in a password manager, and keep a second copy somewhere that survives the same disaster as your laptop. A backup you can't unlock is indistinguishable from no backup.

Create a restic snapshot

Back up one directory, inspect the snapshot, change data, and create a second deduplicated version.

Each restic backup creates a **snapshot** representing the selected paths at a point in time. Snapshots are what turn a copy into history: you can go back to any of them, and unchanged data can be reused inside the repository, so a hundred snapshots of a mostly-static directory cost barely more than one.

Create two snapshots

Back up, change something, back up again:

```
restic backup notes/
printf '%s\n' 'new line' >> notes/report.txt
restic backup notes/
restic snapshots
```

The first run reports everything as new:

```
Files:          42 new,     0 changed,     0 unmodified
Added to the repository: 1.204 MiB (698.221 KiB stored)
snapshot 4a72fb18 saved
```

The second run tells a different story:

```
Files:           0 new,     1 changed,    41 unmodified
Added to the repository: 1.841 KiB (1.203 KiB stored)
snapshot 9c31de07 saved
```

One changed file, and only kilobytes added. That's **deduplication** at work: restic split your data into content-defined chunks, saw that almost all of them already exist in the repository, and stored only the new ones. This is why frequent snapshots are cheap.

`restic snapshots` lists both versions:

```
ID        Time                 Host    Paths
------------------------------------------------------------
4a72fb18  2026-08-03 09:12:33  athena  /home/flavio/notes
9c31de07  2026-08-03 09:15:02  athena  /home/flavio/notes
```

Compare them

Compare snapshot IDs, times, paths, and file changes. Use `restic diff` between the two snapshot IDs:

```
restic diff 4a72fb18 9c31de07
```

```
M    /notes/report.txt

Files:           0 new,     0 removed,     1 changed
```

Exactly the one file we touched, marked `M` for modified. If a diff between two routine snapshots ever shows thousands of unexpected changes, investigate before doing anything else — that pattern is how you notice ransomware or a runaway process rewriting files behind your back.

Two habits to keep

A successful command still needs repository checks and restore tests. ”snapshot saved” proves the backup ran, not that you can get data back.

And keep the live data outside the repository path. Backing up a directory that contains its own repository makes every snapshot swallow the previous ones, and the repository balloons with encrypted copies of itself.

Exclude disposable and secret inputs

Remove caches and rebuildable files while deliberately including unique data and required configuration.

Not everything in a directory deserves backing up. A JavaScript project might be 50 MB of your code and 800 MB of `node_modules` that `npm install` rebuilds in a minute. Backing up the rebuildable part wastes time, storage, and — worse — buries the data you actually care about.

Exclusions reduce time and storage, but a broad pattern can silently omit critical data. Review what is selected before relying on the result.

Write an exclusion file

Create an exclusion file with one pattern per line:

```
node_modules/
.cache/
*.tmp

# use with:
# restic backup --exclude-file exclusions.txt project/
```

Each pattern matches anywhere in the tree, so `node_modules/` catches every nested copy across all your projects. The same file format works over time: you refine it as you discover new cache directories, and every scheduled backup picks up the changes.

Verify what gets selected

Never trust an exclusion pattern you haven't watched work. Run with `--dry-run --verbose` and inspect included and excluded paths:

```
restic backup --exclude-file exclusions.txt --dry-run --verbose project/
```

The verbose output prints a line per file. Scan it for two things: rebuildable junk that slipped through, and — much more important — real data that got excluded by an overly greedy pattern. A pattern like `*.tmp` looks harmless until you meet an application that stores permanent data in files named that way.

Then restore a sample before finalizing the pattern. The dry run shows what restic *plans* to store; a test restore proves the data you care about is really in the snapshot.

Disposable versus secret

Do not exclude a directory merely because it is large. Decide whether it is reproducible and document how. ”We don't back up `node_modules` because `npm ci` rebuilds it from the committed lockfile” is a decision. ”It was big so I skipped it” is a future incident.

Secrets need the same deliberate treatment in the other direction. Files like `.env` are tiny, unique, and often the one thing you can't recreate after a disaster. Inside an encrypted restic repository they're reasonably protected, so my advice is to include them — unless the repository is shared with people who shouldn't hold production credentials. Either way, make it a written decision, not an accident of a glob pattern.

Inspect and compare snapshots

List snapshots and files, locate a historical version, and compare two recovery points before restoring.

Versioned backups are useful only when operators can find the right point. During a real incident you won't restore ”a backup” — you'll need *the snapshot from before the bad thing happened*, and you'll need to find it under pressure. Tags, hosts, paths, and timestamps help narrow the history.

Explore the repository

Three commands cover most of the searching you'll ever do:

```
restic snapshots
restic ls latest
restic find report.txt
```

`restic snapshots` lists every recovery point with its ID, time, host, and paths:

```
ID        Time                 Host    Paths
------------------------------------------------------------
4a72fb18  2026-08-03 09:12:33  athena  /home/flavio/notes
9c31de07  2026-08-03 09:15:02  athena  /home/flavio/notes
```

`restic ls latest` lists the files inside the most recent snapshot, so you can confirm a path was actually captured. And `restic find report.txt` searches *all* snapshots for a filename:

```
Found matching entries in snapshot 4a72fb18 from 2026-08-03 09:12:33
/notes/report.txt

Found matching entries in snapshot 9c31de07 from 2026-08-03 09:15:02
/notes/report.txt
```

That's the "when did this file exist?" question answered in one command. It also works with glob patterns like `restic find '*.sql'`.

Compare two recovery points

Choose two snapshots and run `restic diff OLD NEW`:

```
restic diff 4a72fb18 9c31de07
```

```
M    /notes/report.txt

Files:           0 new,     0 removed,     1 changed
```

`M` marks a modified file, `+` an added one, `-` a removed one. Explain which one precedes the unwanted change — that's the practical skill here. If a config file broke on Tuesday, diff Monday's snapshot against Tuesday's and the changed paths point straight at the suspect.

Don't reflexively restore latest

Do not automatically restore `latest` after corruption or compromise. The newest snapshot may already contain the problem: ransomware-encrypted files, the corrupted database, the broken config. Backups taken *after* an incident faithfully preserve the incident.

Walk backwards through the history with `restic diff` until you find the last snapshot that's clean, and restore that one. The minute spent comparing is much cheaper than restoring the damage on top of itself.

Check your understanding: encrypted versioned backups

Check the commands, decisions, safety boundaries, and failure cases from encrypted versioned backups.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Restore and retention

Restore files and full trees, verify repositories, apply retention, and maintain an independent offsite copy.

Restore one file

Recover a historical file into a separate directory and compare it before replacing live data.

The most common restore isn't a disaster recovery. It's "I overwrote a file an hour ago and I want it back". This small case is where you build the habits that make the big case survivable.

The safest first restore leaves current data untouched. Recover into a temporary destination, inspect it, then choose whether to replace anything. Restoring straight over the live path is a one-way door: if you picked the wrong snapshot, you've now destroyed the current version too.

Restore into a separate directory

Restore one included path:

```
mkdir restore-test
restic restore SNAPSHOT_ID --target restore-test --include /notes/report.txt
```

Replace `SNAPSHOT_ID` with a real ID from `restic snapshots`, chosen because it predates the mistake. The `--include` path is the file's path *as stored in the snapshot* — check it with `restic ls SNAPSHOT_ID` if the filter matches nothing.

The output confirms what happened:

```
repository 3f8a91c2 opened (version 2)
restoring snapshot 4a72fb18 of [/home/flavio/notes] at 2026-08-03 09:12:33 to restore-test
Summary: Restored 2 files/dirs (1.184 KiB) in 0:00
```

If it says `Restored 0 files/dirs`, the include pattern didn't match — nothing was recovered, even though the command exited happily. Always read this line.

Verify before replacing

Open the restored file and compare timestamps, permissions, and content with the wanted version:

```
diff restore-test/notes/report.txt notes/report.txt
stat restore-test/notes/report.txt
```

The `diff` shows exactly what you'd be rolling back. The `stat` confirms metadata came through. Only after this comparison, copy the file into place yourself:

```
cp restore-test/notes/report.txt notes/report.txt
```

That manual copy is deliberate. You stay in control of the final overwrite, one file at a time.

The habit that matters

Use an exact snapshot ID and destination. Do not restore over live data until the recovered version is verified. `latest` is convenient in examples and dangerous in incidents — if the file was corrupted *before* the last backup ran, the latest snapshot contains the corrupted version, and restoring it over your live tree accomplishes nothing except making you feel like you fixed it.

Perform a full restore drill

Recover a complete dataset on a clean destination using only documented tools, credentials, and instructions.

An untested backup is a hope, not a plan. A full drill tests more than bytes. It tests account access, keys, software availability, time, storage capacity, and written procedures — every link in the chain you'll depend on when it's real.

The rule that makes a drill honest: use only what would survive the disaster. Documented instructions, stored credentials, downloadable tools. If you fill a gap from memory during the drill, the drill has found a bug in your runbook. Write it down.

Run the restore

Restore the latest known-good lab snapshot to a fresh location:

```
mkdir full-restore
time restic restore SNAPSHOT_ID --target full-restore
```

```
restoring snapshot 4a72fb18 of [/home/flavio/notes] at 2026-08-03 09:12:33 to full-restore
Summary: Restored 48 files/dirs (3.845 MiB) in 0:02

real    0m2.311s
```

Wrapping the command in `time` matters: measured elapsed time on a realistic dataset is the only honest input to your RTO. A 2-second lab restore scales very differently when the real dataset is 200 GB behind a slow uplink.

Verify like you mean it

Measure elapsed time and verify file counts, checksums, permissions, and application startup where relevant:

```
find full-restore -type f | wc -l        # expect the snapshot's file count
diff -r notes/ full-restore/notes/       # silence means identical
stat full-restore/notes/report.txt       # spot-check ownership and mode
```

For an application dataset, the real verification is starting the application against the restored files and clicking around. Files that exist but don't produce a working service are a failed restore that *looks* successful.

Record every missing dependency as you hit it: the password you had to look up, the restic version you had to guess, the disk that was nearly too small. Each one is a runbook fix, and finding them now is the entire value of the exercise.

Keep the drill safe

A drill must not endanger live data. Use a disposable destination and isolated service configuration. Never point a drill's restored application at production databases or credentials — a ”test” service that sends real emails to real customers is its own incident. The drill should be boring for everyone except your documentation.

Check repository integrity

Run repository checks, understand their scope, and schedule deeper data verification separately from restore drills.

A repository can sit on a slowly failing disk for years, accepting new snapshots while old data rots underneath. Repository checks validate structure and can read stored data to detect damage —

before the day you need that data back.

They complement, but do not replace, restoring usable files. A check proves the repository's internal bookkeeping and bytes are intact. Only a restore drill proves you can turn those bytes into a working system.

Two levels of checking

Check the practice repository:

```
restic check
restic check --read-data-subset=10%
```

Plain `restic check` verifies the repository structure: it confirms every snapshot's metadata is consistent and every data chunk that should exist is referenced correctly. It's fast because it reads almost none of the actual data.

```
create exclusive lock for repository
load indexes
check all packs
check snapshots, trees and blobs
no errors were found
```

`no errors were found` is the line you're looking for.

The `--read-data-subset=10%` variant additionally downloads 10% of the stored data and verifies it against its checksums. This is what catches bit rot and truncated uploads — problems invisible to the structural check.

Schedule rotating coverage

Reading all data on every check would hammer a large repository and burn bandwidth on remote storage. Record duration and result for your repository, then design a schedule that eventually covers all stored data without overwhelming the backup target.

The subset syntax supports rotation directly:

```
restic check --read-data-subset=1/10   # week 1: first tenth
restic check --read-data-subset=2/10   # week 2: second tenth
```

Cycle through `1/10` to `10/10` and every byte gets verified once per ten weeks, at a tenth of the weekly cost.

When a check fails

Investigate failures before pruning or rewriting the repository. A failed check means the repository holds damage, and your instinct will be to ”clean it up” — resist it. Pruning rewrites and deletes data, which can turn a partially damaged repository into a thoroughly broken one, and destroys the evidence of what went wrong.

Preserve evidence and another copy first: the error output, the affected files, and a second copy of whatever still restores. Then diagnose. Damaged repositories can often be partially recovered, but only if you didn't rewrite them in a panic.

Apply retention with care

Preview which snapshots a policy keeps, forget only reviewed history, and prune data in a separate controlled step.

Snapshots accumulate. Daily backups produce 365 recovery points a year, and most of them stop earning their storage after a few weeks. **Retention** controls storage growth while preserving useful recovery points. The danger runs the other way too: an overly aggressive policy can erase the last good version of something you haven't noticed is broken yet.

The standard shape is tiered: keep every recent snapshot, then thin out with age. Recent history is dense because recent mistakes are the ones you discover.

Preview before deleting

restic separates deciding from deleting. Preview a simple policy first:

```
restic forget --keep-daily 7 --keep-weekly 4 --keep-monthly 12 --dry-run
```

This keeps one snapshot per day for 7 days, one per week for 4 weeks, and one per month for 12 months. With `--dry-run`, nothing is deleted — restic prints two tables, `keep` and `remove`, with the reason each snapshot survives:

```
keep 11 snapshots:
ID        Time                 Reasons
----------------------------------------------
9c31de07  2026-08-03 03:00:01  daily snapshot
                               weekly snapshot
...
remove 23 snapshots:
```

Explain every retained and removed snapshot before dropping the flag. Apply `forget` only after the policy and current incident state are understood. If a snapshot you'd want is in the remove list, fix the policy now — the dry run is free, the mistake isn't.

Forget, then prune, then check

`forget` only removes the snapshot records. The underlying data stays until you prune:

```
restic forget --keep-daily 7 --keep-weekly 4 --keep-monthly 12
restic prune
restic check
```

`prune` finds data no snapshot references anymore, repacks what's partially used, and deletes the rest. Keeping it as a separate controlled step is deliberate: prune rewrites the repository, so you run it when things are calm, and you run `restic check` afterwards to confirm the repository is still healthy. Pruning on top of undetected damage makes the damage worse.

When not to prune

Do not prune during an active recovery or suspected compromise. Pruning removes unreferenced data and reduces options — including snapshots you just forgot in a hurry, which remain recoverable until prune runs. During an incident, storage cost is the cheapest problem you have. Freeze retention, finish the recovery, then clean up.

Check your understanding: restore and retention

Check the commands, decisions, safety boundaries, and failure cases from restore and retention.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Application recovery

Back up databases consistently, automate jobs, protect keys, alert on failure, and rehearse a complete disaster.

Back up a database consistently

Use the database backup mechanism instead of copying changing storage files as ordinary files.

Here's a mistake that produces backups which look perfect and restore to garbage: pointing your file backup tool at a running database's data directory.

A running database changes related files together. While your backup tool walks the directory, the database keeps writing — so the copy gets table files from 2:00 AM and index files from 2:03 AM. That torn mixture was never the database's real state at any moment. Restored, it's corruption: the server may refuse to start, or worse, start and serve subtly wrong data.

The fix is to ask the database for a backup instead of taking one behind its back. A logical dump or supported physical backup creates a consistent recovery representation.

Dump with pg_dump

Create a PostgreSQL custom-format lab dump:

```
pg_dump --format=custom --file app.dump appdb
pg_restore --list app.dump | head
```

`pg_dump` runs inside a single transaction, so the dump reflects one consistent instant no matter how long it takes or how busy the database is. The custom format is compressed and lets `pg_restore` restore selectively.

The `--list` command doubles as a quick sanity check — it reads the dump's table of contents:

```
; Archive created at 2026-08-03 03:00:02 CEST
;     dbname: appdb
;     Format: CUSTOM
...
```

If the file is truncated or corrupt, this fails immediately. A zero-byte `app.dump` created by a dump that failed silently in cron is a classic way to discover, months later, that you have no database backups at all. Check the exit status and the file size after every dump.

Prove it restores

Restore into a separate empty database, never over the live one:

```
createdb appdb_restore
```

```
pg_restore --dbname appdb_restore app.dump
psql appdb_restore -c 'select count(*) from orders;'
```

Then run application-level checks on row counts and important relationships. Matching row counts and intact foreign keys between live and restored are the proof that this backup is worth keeping.

Two operational notes to close. Protect credentials and dumps — a logical dump is your entire database in readable form, so encrypt it at rest, for example by backing it up into a restic repository. And coordinate database version compatibility and extensions with the recovery plan: a dump from PostgreSQL 17 with `postgis` installed needs a target that can provide both.

Automate and alert on backups

Schedule backups with locking, deadlines, logs, and failure notifications that someone will act on.

A backup that depends on memory will eventually stop. You'll be busy, then on vacation, then it's been four months. Automation fixes that — and creates a new failure mode: jobs that break silently and keep ”running” as no-ops for months. Automation must also make silence and repeated failure visible.

Wrap the backup with an honest exit

Create a wrapper result suitable for a scheduler:

```
if restic backup /srv/data; then
  printf '%s backup=success\n' "$(date -u +%FT%TZ)"
else
  status=$?
  printf '%s backup=failure status=%s\n' "$(date -u +%FT%TZ)" "$status" >&2
  exit "$status"
fi
```

The script logs a timestamped result line and, crucially, exits non-zero on failure. Schedulers and alerting can only react to what the script reports.

Schedule it with cron, capturing output:

```
# /etc/cron.d/backup
0 3 * * * root /usr/local/bin/backup.sh >> /var/log/backup.log 2>&1
```

Cron is where silent failures breed. It runs with a minimal environment: a stripped `PATH`, no `RESTIC_REPOSITORY`, no `RESTIC_PASSWORD_FILE`. A script that works perfectly in your shell fails at 3 AM because the variables were only in your session. Export everything the job needs inside the script itself, and test by running it from a bare environment: `env -i /usr/local/bin/backup.sh`.

Test the failure path

Now force a permission and destination failure — make the repository unreachable, or revoke read access to `/srv/data`. Confirm the scheduler records non-zero status and the alert reaches the responsible person. An alert channel nobody checks is decoration. If you've never seen your backup alert fire, you don't have alerting; you have optimism.

Alert on silence, not just errors

There's a failure the error path can't catch: the job never ran at all. Disabled cron, powered-off machine, deleted crontab. No run means no error means no alert.

A ”job did not run” condition needs detection too. Alert on missing recent successful snapshots. The dead-man's-switch pattern works well: the wrapper pings a monitoring URL after each success, and the monitor alerts when pings *stop* arriving:

```
restic backup /srv/data && curl -fsS https://hc-ping.com/6c2ea1d4-backup
```

Alternatively, a daily check that the newest snapshot is younger than 24 hours (`restic snapshots --json` plus a timestamp comparison) catches the same silence from the repository side.

Protect recovery credentials

Keep repository passwords, storage credentials, encryption keys, and recovery instructions available without exposing them to every source host.

Recovery credentials must survive the same disaster as the data. This sounds obvious and gets violated constantly, usually through a circular dependency: the restic password lives in the password manager, the password manager's vault is backed up in the restic repository, and now each one requires the other. The circle looks fine every day — until the day both ends are gone.

There's a second, opposite pressure. Credentials also need narrower access than routine backup writing where possible. The server that creates backups every night should hold a credential that can *write* snapshots but not *delete* them. Otherwise anyone who compromises the server can destroy the history too — the exact ransomware scenario backups exist for.

Map every credential

Create a credential map without values — locations only, safe to print and circulate:

```
restic password: password manager emergency record
storage write credential: server secret store
storage delete credential: separate administrator account
recovery instructions: offline runbook copy
```

Each line answers one recovery question. Where is the repository password when the laptop is dead? Where is the storage credential when the server is gone? Who can delete old data, and is that person's account separate from the machines doing daily writes?

Then apply the survival test to each row: does this location share a fate with the data it unlocks? A password manager emergency kit printed and stored at a relative's house survives your house fire. A `password.txt` on the backed-up laptop doesn't. The **offline runbook copy** matters for the same reason — instructions stored only in the wiki are unavailable exactly when the wiki's server is what died.

Drill it

The map is a claim; a drill is proof. Have an authorized second recovery path locate the required credentials in a drill without revealing them in notes or logs. A partner, a colleague, or you-in-six-months should be able to go from the map to an opened repository using only what the map says. If they get stuck, the map is wrong — fix it now, not during the incident.

The rule that anchors it all: do not store the only decryption key beside the only backup copy or only on the protected machine. Any single place that holds *the only* copy of a recovery credential is a single point of failure, no matter how secure it is.

Run a disaster recovery exercise

Simulate loss of the source system and rebuild service from documentation, clean infrastructure, backups, and protected credentials.

Everything in this course converges here. A restore drill tests one dataset; a **disaster recovery exercise** tests the whole story. The final exercise begins as if the original machine is gone. Not "unavailable for a bit" — gone, along with every convenient shortcut it held: shell history, SSH keys, that one script only it had.

The discipline that makes it real: use a clean disposable system and the same recovery boundaries a real incident would require. A fresh VM or spare machine, your written runbook, your protected credentials. Nothing from memory. Every time you catch yourself typing something the runbook doesn't say, you've found a gap — the exercise is working.

Follow the runbook in order

```
1. establish clean recovery host
2. install verified tools
3. obtain protected credentials
4. select known-good recovery point
5. restore data and configuration
6. verify application behavior
7. record time and gaps
```

Each step fails in its own way, which is why each one is on the list. Step 2 catches undocumented tool versions. Step 3 is where circular credential dependencies surface. Step 4 forces you to *choose* a snapshot and justify it, not reflexively grab `latest`. Step 6 is the honest finish line: files on disk mean nothing until the application starts and behaves correctly against them.

Keep a timer running and a notes file open. Timestamps per step, every surprise, every improvisation:

```
14:02  host ready
14:19  restic installed, version pinned in runbook? NO - gap
14:31  credentials located (map was accurate)
15:58  restore complete, app starts, orders count matches
```

Score it against your targets

Compare the measured recovery point and time with RPO and RTO. If the exercise took four hours against a one-hour RTO, you now know — cheaply, calmly — that the target is fiction. Either improve the process or renegotiate the target. Then turn every manual guess into an updated instruction or automation, and schedule the next exercise, because runbooks rot as systems change.

One safety boundary throughout: keep the exercise isolated from production names, credentials, and writable storage until the restored system is verified. A recovered application with stale config can email real customers or write to the real database. The exercise should prove recovery works — quietly.

Check your understanding: application recovery

Check the commands, decisions, safety boundaries, and failure cases from application recovery.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/backup-and-restore/.