GIT & GITHUB SERIES · COURSE 3

Git & GitHub Advanced Mastery

Internals, interactive rebase, disaster recovery, submodules, advanced CI, security, large repos, bisect, and two final real-world scenarios.

10 Chapters

osztromok.com

CHAPTER 1: GIT INTERNALS

COURSE 3 · CH 1

Git Internals

Objects, refs, and the .git directory demystified — what's actually happening beneath every command from Courses 1 and 2

Courses 1 and 2 covered git at the command level — what to type, what happens. This chapter goes one layer deeper: what's actually stored inside `.git`, and how. Understanding this isn't required to use git well day to day, but it's exactly what turns "I memorised these commands" into "I understand why these commands work," and it's the foundation everything else in this advanced course builds on — interactive rebase, reflog recovery, and real submodules all make far more sense once the object model clicks.

Git Is a Content-Addressable Object Database

Strip away every porcelain command (`commit`, `branch`, `merge`) and underneath, git is fundamentally a simple key-value store: content goes in, a hash comes out, and that hash is forever the key to retrieve that exact content. This is what "content-addressable" means — the address *is* a hash of the content itself.

📄 Blob

The raw content of a single file, nothing else — no filename, no permissions, just bytes. Two files with identical content produce the exact same blob, even in completely unrelated commits.

🌳 Tree

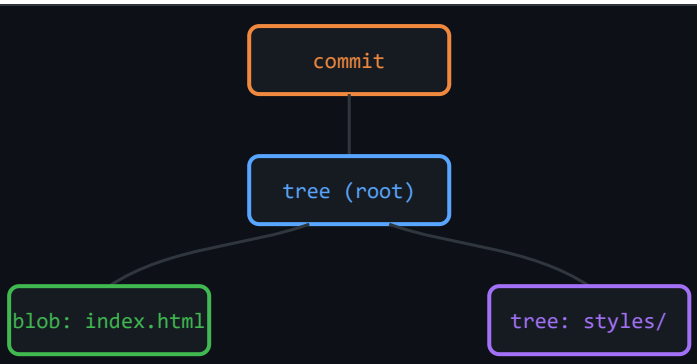
A directory listing — maps filenames and permissions to blob hashes (for files) or other tree hashes (for subdirectories). This is how git represents folder structure.

📦 Commit

Points to one tree (the project's complete state at that moment), one or more parent commits, plus author, message, and timestamp metadata.

🏷️ Tag (annotated)

Points to a commit, with its own message and metadata — covered practically in Course 2, Chapter 5.



Every commit points to exactly one root tree, which can point to blobs (files) and further trees (subdirectories) — the entire project state, addressed entirely by hashes

Seeing this for real

```
$ # Show what type of object a hash refers to
$ git cat-file -t HEAD
commit

$ # Show a commit object's raw content
$ git cat-file -p HEAD
tree 9f8e7d6c5b4a...
parent a1b2c3d4e5f6...
author Philip Osztromok <you@example.com> 1750000000 +0100
committer Philip Osztromok <you@example.com> 1750000000 +0100

Fix login validation bug
```

> **THIS IS WHY GIT IS SO FAST AT DETECTING UNCHANGED FILES**
>
> Because the hash is a function of content, git can tell two files are identical just by comparing hashes — no need to actually re-read and diff full file contents. This is also why renaming a file with no content changes is essentially free for git to recognise: the blob hash stays exactly the same, only the tree entry's name changes.

Refs — Human-Readable Names for Hashes

Nobody wants to type 40-character hashes constantly. A **ref** is simply a small file containing a hash, with a memorable name — this is the literal mechanism behind everything Course 1, Chapter 4 explained conceptually about branches being "just a movable label."

```
.git/
├── refs/
│   ├── heads/
│   │   ├── main ← contains just a commit hash, nothing more
│   │   └── feature/login
│   └── tags/
│       └── v1.2.0
├── HEAD ← usually contains "ref: refs/heads/main", a pointer to a pointer
└── objects/ ← every blob, tree, commit, and tag object, content-addressed
```

```
$ # See exactly what HEAD currently contains
$ cat .git/HEAD
ref: refs/heads/main

$ # And what that ref points to
$ cat .git/refs/heads/main
a1b2c3d4e5f6789...
```

This is the literal, file-level answer to "what is HEAD" from Course 1, Chapter 4 — it's a text file containing either a ref name (the normal case) or, occasionally, a raw commit hash directly (a "detached HEAD" state).

Where Everything Else from Courses 1 and 2 Fits

- **git commit** creates a new commit object, pointing at a new tree object built from your staged blobs, then moves the current branch's ref file to point at it.
- **git branch <name>** just creates a new file in `.git/refs/heads/` containing the current commit's hash.
- **git merge** creates a commit object with two parent hashes instead of one.
- **git rebase** (Course 2, Ch2) creates brand-new commit objects with different hashes, then moves the branch ref to point at the new chain — this is the literal mechanism behind "rebase rewrites history."
- **The broken gitlink incident** (Course 1 Ch9, Course 2 Ch10) — a gitlink is a special tree entry (mode `160000`) that records a commit hash from a *different* object database entirely, rather than a blob or tree from this one. When that other database (the nested .git) disappears, the reference becomes unresolvable.

YOU'LL ALMOST NEVER TOUCH THESE FILES DIRECTLY — AND YOU SHOULDN'T

Everything in this chapter is read-only exploration territory. Manually editing files inside `.git/` is how repositories get genuinely, sometimes unrecoverably, broken — every porcelain command (commit, branch, merge, rebase) exists specifically to make these changes safely and consistently. Use `git cat-file`, `cat .git/HEAD`, and similar read-only exploration to build understanding, never to modify.

Command Reference

COMMAND	WHAT IT DOES
`git cat-file -t <hash>`	Shows the object type (blob, tree, commit, tag) for a given hash
`git cat-file -p <hash>`	Pretty-prints the object's actual content
`git ls-files -s`	Shows tracked files with their mode and blob hash (used in Course 1/2's gitlink diagnosis)
`cat .git/HEAD`	Shows what HEAD currently points to — a ref name, or a raw hash if detached
`cat .git/refs/heads/<branch>`	Shows the exact commit hash a branch currently points to

CHAPTER 1 QUICK REFERENCE

- **Git is content-addressable** — every object's hash is a function of its own content
- **Four object types:** blob (file content), tree (directory listing), commit (snapshot + metadata), tag (annotated marker)
- **A ref** is just a small file holding a hash with a memorable name — the literal mechanism behind branches
- **HEAD** is a file, usually containing "ref: refs/heads/<branch>" — a pointer to a pointer
- **git cat-file -p / -t** — read-only tools for exploring objects directly
- **Never hand-edit files inside .git/** — porcelain commands exist to do this safely
- **Next chapter:** interactive rebase — squashing, reordering, and splitting commits safely

CHAPTER 2: INTERACTIVE REBASE

COURSE 3 · CH 2

Interactive Rebase

Squashing, reordering, splitting, and rewriting commits safely — cleaning up your own history before anyone else sees it

Course 2's rebase chapter covered moving a whole branch onto a new base. Interactive rebase is a different, more surgical use of the same underlying mechanism — Chapter 1's object model, where rebasing creates new commit objects — applied to *editing your own commit history directly*: reordering commits, combining several into one, splitting one into several, or rewriting messages, all before anyone else has seen them.

Starting an Interactive Rebase

```
$ # Interactively edit the last 4 commits
$ git rebase -i HEAD~4
```

This opens your configured editor with a list of the targeted commits, oldest first, each prefixed with an action word (`pick` by default) — editing this list and saving it tells git exactly what to do with each commit.

```
pick a1b2c3d Add login form HTML
pick f9e8d7c fix typo
pick 4f3e2d1 Add validation logic
pick 9c8b7a6 WIP debugging, remove later

# Rebase 5d4c3b2..9c8b7a6 onto 5d4c3b2 (4 commands)
#
# Commands:
# p, pick = use commit
# s, squash = use commit, but meld into previous commit
# r, reword = use commit, but edit the commit message
# d, drop = remove commit
# e, edit = stop for amending
```

The Actions, One at a Time

`pick`

Keep this commit exactly as is. The default for every line — change nothing for commits you're happy with.

`squash`

Combine this commit into the one above it, merging their changes into a single commit. You'll be prompted to write a combined message.

`fixup`

Same as squash, but silently discards this commit's message entirely — useful for "oops, typo" commits that don't deserve their own line in the combined message.

`reword`

Keep the commit's changes, but stop and let you edit just its message.

`drop`

Remove this commit's changes from history entirely — exactly what removing a line (or this explicit action) does for a commit you no longer want at all.

`edit`

Pause the rebase right after applying this commit, letting you amend it, add more changes, or even split it into multiple commits before continuing.

Example: cleaning up the messy history above

```
pick a1b2c3d Add login form HTML
fixup f9e8d7c fix typo
pick 4f3e2d1 Add validation logic
drop 9c8b7a6 WIP debugging, remove later
```

Saving this turns four messy commits into two clean ones: "Add login form HTML" (now silently including the typo fix) and "Add validation logic" — with the leftover debug commit gone entirely. This is exactly the kind of cleanup worth doing before opening a PR (Course 1, Chapter 7), so reviewers see a clear, deliberate history rather than your real-time trial and error.

Splitting a Commit

Sometimes a single commit did too much and should really be two. Mark it `edit`, and when the rebase pauses there:

```
$ # Rebase paused after applying the commit you marked "edit"
$ # Undo just this commit, keeping the changes in your working directory
$ git reset HEAD~1
# Now stage and commit the changes as two (or more) separate, focused commits
$ git add login.js
$ git commit -m "Add login form validation"
$ git add styles.css
$ git commit -m "Style the login form"

$ # Continue the rebase with the rest of the original list
$ git rebase --continue
```

Reordering Commits

Simply reorder the lines in the editor before saving — git replays them in whatever order they appear in the list, top to bottom. Useful when a later fix logically belongs before an earlier feature commit, for a cleaner reading order.

> **REORDERING COMMITS THAT TOUCH THE SAME LINES CAN CAUSE CONFLICTS**
>
> If commit B was written assuming commit A's changes already existed, and you reorder B before A, git may hit a conflict trying to apply B without A's context — resolved the same way as any rebase conflict (Course 2, Chapter 3). Reordering is safe in principle, but not magically immune to logical dependencies between commits.

The Golden Rule Still Applies — Even More So Here

Every single action in this chapter rewrites commit history — new hashes, every time, exactly as Chapter 1's object model and Course 2's rebase chapter explained. Interactive rebase is specifically a tool for cleaning up **your own, not-yet-shared** commits. Running it on commits others have already pulled creates the exact force-push danger covered in Course 2, Chapter 9 — multiplied, since you're potentially rewriting several commits' worth of history at once rather than just rebasing onto a new base.

> **A PRACTICAL HABIT: CLEAN UP RIGHT BEFORE OPENING THE PR, NOT THROUGHOUT**

> Rather than interactively rebasing constantly while still actively working (where you might still want those messy WIP commits as a personal safety net), many developers do one clean interactive rebase pass immediately before opening a PR — turning a realistic, messy development process into a clean, reviewable set of commits, without losing the messy intermediate states until you're confident you don't need them.

Command Reference

COMMAND	WHAT IT DOES
`git rebase -i HEAD~N`	Opens an interactive rebase covering the last N commits
`pick / squash / fixup / reword / drop / edit`	Per-commit actions available in the rebase editor
`git rebase --continue`	Resumes the rebase after resolving a conflict or finishing an "edit" pause
`git rebase --abort`	Cancels the entire interactive rebase, restoring the original history (Course 2, Ch3)

CHAPTER 2 QUICK REFERENCE

- **git rebase -i HEAD~N** — opens an editable list of the last N commits
- **squash/fixup** — combine a commit into the previous one (fixup discards the message)
- **reword** — edit just the message; **drop** — remove the commit entirely
- **edit** — pause to amend, add changes, or split a commit into several
- **Reordering** — just rearrange lines before saving; can cause conflicts if commits logically depend on each other
- **Golden rule:** only on commits nobody else has pulled — same rewriting danger as Course 2, Chapter 9, multiplied
- **Practical habit:** one clean interactive rebase pass right before opening a PR
- **Next chapter:** reflog and disaster recovery — undoing even a bad reset or rebase

CHAPTER 3: REFLOG AND DISASTER RECOVERY

COURSE 3 · CH 3

Reflog and Disaster Recovery

The safety net behind almost every "I think I just destroyed my work" moment — including a bad reset, rebase, or even a force-push

This chapter has been referenced as a safety net throughout the entire series — Course 1's `reset --hard` warning, Course 2's force-push danger, Course 3's own interactive rebase chapter. Now it gets full treatment: what the reflog actually is, and the step-by-step recovery process for the most common "I broke it" scenarios.

What the Reflog Actually Tracks

Building on Chapter 1's object model: every time a ref (HEAD, a branch) moves — a commit, a checkout, a reset, a rebase step — git records that movement in the **reflog**, a local, personal log of where your refs have recently pointed. It's not part of the object database itself and isn't shared when you push — purely a local safety net.

```
$ git reflog
9c8b7a6 HEAD@{0}: commit: Add validation logic
4f3e2d1 HEAD@{1}: reset: moving to HEAD~1
a1b2c3d HEAD@{2}: commit: Add login form HTML
f9e8d7c HEAD@{3}: rebase (start): checkout main
```

Each line is a snapshot of where HEAD pointed at that moment, most recent first. **Crucially, this includes commits that no branch currently points to anymore** — exactly the case after a hard reset or a rewritten rebase, where the "old" commits still technically exist in git's object database, just unreachable by any current branch or tag.

THIS IS WHY A "DESTROYED" COMMIT USUALLY ISN'T ACTUALLY GONE

Course 1, Chapter 8 explained that even `git reset --hard` isn't always unrecoverable — the reflog is the literal mechanism why. The commit object itself (Chapter 1) still exists in `.git/objects/` even

> when no ref points to it anymore; the reflog is how you find its hash again to point something back at it.

Recovery Scenario: Undoing a Bad Hard Reset

```
$ # Accidentally went too far back
$ git reset --hard HEAD~3
HEAD is now at f9e8d7c Some much older commit

$ # Find where you were before the reset
$ git reflog
9c8b7a6 HEAD@{1}: commit: Add validation logic ← this is what you lost
f9e8d7c HEAD@{0}: reset: moving to HEAD~3

$ # Reset forward again, to the commit you actually wanted
$ git reset --hard 9c8b7a6
HEAD is now at 9c8b7a6 Add validation logic
```

Recovery Scenario: A Rebase Went Badly Wrong

Interactive rebase (Chapter 2) can be aborted mid-way with `--abort` — but what if it already finished, and the result is wrong? The reflog still has the pre-rebase state.

```
$ git reflog
4f3e2d1 HEAD@{0}: rebase (finish): returning to refs/heads/feature
9c8b7a6 HEAD@{4}: rebase (start): checkout HEAD~4 ← look just before this line
b2c1d0e HEAD@{5}: commit: Last commit before the rebase started

$ # Point your branch back at the pre-rebase state
$ git reset --hard b2c1d0e
```

Recovery Scenario: Recovering After a Force-Push

Course 2, Chapter 9 covered this from the perspective of the person who got overwritten — usually, their own local copy is fine since it was never touched. But if you're the one trying to

recover what you pushed over:

🔍 If you have a local clone from before the force-push

Your own reflog has the commits you overwrote — find them and push them back, exactly as in the hard-reset scenario above.

`git reflog → git push --force-with-lease`

🤝 If a teammate's commits were overwritten

Their own local copy almost certainly still has the commits — the reflog rescue happens on their machine, not yours, since the commits were never touched there.

`Ask them to git push their unaffected local branch`

⏰ If too much time has passed

The reflog has a default expiry (90 days for reachable, 30 for unreachable commits) and git's periodic garbage collection can eventually prune genuinely unreachable objects — recovery isn't guaranteed forever.

`git gc --prune=now destroys this safety net — never run it as a "fix"`

GIT GC AND REFLOG EXPIRY ARE THE REAL LIMITS OF THIS SAFETY NET

The reflog isn't infinite — it expires, and manually running aggressive garbage collection deletes genuinely unreachable objects for good. If you're mid-recovery, avoid running `git gc` until you've finished — it exists to clean up storage, not to be run during a rescue.

Finding a Specific Lost Commit, Not Just "the previous state"

If you're not sure which reflog entry you need, search by commit message or content rather than scanning the whole log manually:

```
$ # Search reflog entries by message text
$ git reflog --grep="validation"

$ # Search ALL commit objects (not just reachable ones) by content
$ git fsck --unreachable | grep commit
```

`git fsck --unreachable` is the deeper fallback when even the reflog has expired or doesn't have what you need — it lists every commit object still physically present in `.git/objects/`, including ones with no ref or reflog entry pointing to them at all, as long as garbage collection hasn't pruned them yet.

Command Reference

COMMAND	WHAT IT DOES
`git reflog`	Shows the local history of where HEAD/branches have pointed recently
`git reflog --grep="text"`	Searches reflog entries by message content
`git reset --hard <hash>`	Points your current branch back at a specific commit found via reflog
`git fsck --unreachable`	Lists commit objects with no current ref pointing to them, reflog or not
`git gc`	Garbage collection — can permanently prune unreachable objects; avoid mid-recovery

CHAPTER 3 QUICK REFERENCE

- **The reflog** is a local, personal log of where HEAD/branches have pointed — not pushed, not shared
- **Includes unreachable commits** — exactly what a hard reset or rewritten rebase leaves behind, still physically present
- **Recovery pattern:** git reflog → find the hash you want → git reset --hard <hash>
- **Force-push recovery:** your own reflog if you have it; otherwise the original author's untouched local copy
- **The reflog isn't infinite** — expires over time, and git gc can permanently prune unreachable objects
- **git fsck --unreachable** — deeper fallback for commits the reflog itself no longer references
- **Never run git gc mid-recovery** — it's the one command that can make this chapter's techniques stop working

- **Next chapter:** submodules and subtrees — proper setup, contrasted with the broken-gitlink trap from earlier courses

CHAPTER 4: SUBMODULES AND SUBTREES

COURSE 3 · CH 4

Submodules and Subtrees

Proper setup for embedding one repository inside another — and a clean alternative that avoids the gitlink trap entirely

This chapter closes the loop on something teased across the entire series — Course 1, Chapter 9's broken-gitlink incident, and Course 2, Chapter 10's full diagnosis. Both showed what happens when a folder ends up tracked as a gitlink *by accident*. This chapter covers the two real, supported ways to do this deliberately: submodules, set up correctly this time, and subtrees, an entirely different approach that sidesteps the gitlink mechanism altogether.

Submodules, Done Properly

Chapter 1 explained the mechanism: a submodule is a gitlink (tree entry, mode `160000`) pointing at a commit hash in a *separate* repository, plus a `.gitmodules` file recording the URL and path. Adding one correctly:

```
$ git submodule add https://github.com/someone/shared-ui-components.git libs/ui
Cloning into 'libs/ui'...
$ git status
new file: .gitmodules
new file: libs/ui
$ git commit -m "Add shared-ui-components as a submodule"
```

```
.gitmodules ← records URL + path, the thing the broken incident was missing entirely
[submodule "libs/ui"]
  path = libs/ui
  url = https://github.com/someone/shared-ui-components.git
```

The part everyone gets wrong: cloning a repo WITH submodules

A plain `git clone` creates the gitlink-tracked folder but leaves it **empty** — the submodule's actual content needs a separate, explicit step.

```
$ # Either clone with this flag from the start...
$ git clone --recurse-submodules https://github.com/you/project.git

$ # ...or, after a plain clone, populate them after the fact
$ git submodule update --init --recursive
```

AN EMPTY SUBMODULE FOLDER AFTER A PLAIN CLONE IS NORMAL, NOT BROKEN

Unlike the genuinely broken gitlink from earlier courses (pointing at a commit hash from a vanished repository), an empty-but-correctly-configured submodule folder is expected behaviour — the fix is simply `git submodule update --init`, not a diagnostic investigation. Checking for a valid `.gitmodules` entry (Course 2, Chapter 10's key distinguishing signal) tells you immediately which situation you're in.

Updating a submodule to a newer commit

```
$ cd libs/ui
$ git pull origin main
$ cd ../..
$ # The parent repo sees this as a change — the gitlink now points at a newer commit
$ git add libs/ui
$ git commit -m "Update ui submodule to latest"
```

Subtrees — A Different Approach Entirely

`git subtree` takes the opposite philosophy: instead of a pointer to another repository, it actually **copies** the other project's files directly into your repo, as ordinary tracked content — no gitlink, no `.gitmodules`, no separate clone step for anyone using the repo.

```
$ git subtree add --prefix=libs/ui https://github.com/someone/shared-ui-
components.git main --squash
```

After this, `libs/ui` contains ordinary tracked files — anyone cloning your repo gets them automatically, with zero extra steps, exactly like every other file in the project.

Submodules vs Subtrees

🔗 Submodule

Stored as: a pointer (gitlink) to a separate repository.

Cloning: requires an extra step (--recurse-submodules or submodule update).

Best for: when you want to track an exact upstream version and occasionally pull in updates from the original project cleanly.

📋 Subtree

Stored as: a real, ordinary copy of the files, fully merged into your repo's own history.

Cloning: works exactly like any other clone — nothing extra required.

Best for: simpler day-to-day use, especially when contributors shouldn't need to learn submodule-specific commands at all.

Choosing Between Them

SITUATION	USE
Need to track an exact upstream version, pull updates occasionally	Submodule
Want zero extra steps for anyone cloning the repo	Subtree
Team unfamiliar with submodule workflow, wants simplicity	Subtree
Need to occasionally contribute changes back upstream	Submodule — keeps a clean separate history to push from
Just need a folder with normal tracked content, no real "other project"	Neither — this is the broken-gitlink trap; just track it normally

THE HONEST SUMMARY: MOST PROJECTS NEED NEITHER

Both tools solve a specific, relatively narrow problem — genuinely sharing code between separate, independently-versioned repositories. The far more common situation (a folder of files that's just part of your one project) needs neither — it should simply be tracked normally, exactly the fix applied in Course 2, Chapter 10 to undo the accidental gitlink. Reach for these tools only when there's a real second repository genuinely worth keeping separate.

Command Reference

COMMAND	WHAT IT DOES
`git submodule add <url> <path>`	Adds a properly configured submodule, creating/updating .gitmodules
`git clone --recurse-submodules <url>`	Clones a repo and populates its submodules in one step
`git submodule update --init --recursive`	Populates submodules after a plain clone left them empty
`git subtree add --prefix=<path> <url> <branch> --squash`	Copies another repo's files directly into your repo as ordinary content

CHAPTER 4 QUICK REFERENCE

- **A proper submodule** always has a matching .gitmodules entry — the exact thing the broken incident lacked
- **Plain git clone leaves submodules empty** — use --recurse-submodules or submodule update --init afterward; this is normal, not broken
- **git subtree** copies another project's files directly into your repo — no gitlink, no extra clone steps for anyone
- **Submodule** — best when tracking an exact upstream version and occasionally contributing back
- **Subtree** — best for simplicity, zero extra steps for contributors
- **Most projects need neither** — a folder of files that's just part of your project should be tracked normally
- **Next chapter:** advanced GitHub Actions — matrix builds, secrets, caching, and deployment pipelines

CHAPTER 5: ADVANCED GITHUB ACTIONS

COURSE 3 · CH 5

Advanced GitHub Actions

Matrix builds, secrets, caching, and a complete deployment pipeline — taking CI from "runs tests" to "ships your project"

Course 2, Chapter 8 covered a single straightforward test workflow. Real projects usually need more: testing across multiple versions at once, using credentials safely, avoiding redundant work on every run, and — the natural endpoint — automatically deploying when everything passes.

Matrix Builds — Testing Multiple Configurations at Once

A matrix runs the same job multiple times with different variable combinations — typically used to test against several language versions or operating systems simultaneously, in parallel, rather than one at a time.

```yaml
jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: ['18', '20', '22']
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node-version }}
      - run: npm test
```

This single definition produces three separate, parallel job runs:

Job	node-version	Result
test (1)	18	✅ Independent
test (2)	20	✅ Independent
test (3)	22	✅ Independent

MATRICES CAN COMBINE MULTIPLE DIMENSIONS AT ONCE

Adding a second array (e.g. `os: [ubuntu-latest, windows-latest, macos-latest]` alongside node-version) produces every combination automatically — 3 Node versions × 3 operating systems = 9 parallel jobs from a few lines of YAML.

Secrets — Using Credentials Safely in CI

Course 1, Chapter 6 established that secrets never belong in committed code. Actions workflows often genuinely need credentials (a deploy token, an API key for a test run) — GitHub Secrets is the answer: encrypted values stored at the repo or organisation level, injected into workflows at runtime without ever appearing in logs or the YAML file itself.

```
steps:
  - name: Deploy to production
    env:
      DEPLOY_TOKEN: ${{ secrets.DEPLOY_TOKEN }}
    run: ./deploy.sh
```

Set up under Settings → Secrets and variables → Actions → New repository secret. The actual value is never visible again after saving, even to repo admins — only usable inside workflow runs.

SECRETS CAN STILL LEAK IF YOU'RE NOT CAREFUL WITH WHAT YOU LOG

GitHub automatically masks a secret's exact value in logs — but only if the log output matches the secret exactly. Echoing a transformed or partial version of a secret (base64-encoded, concatenated with other text) can bypass this masking entirely. Never deliberately print a secret for "debugging" — find another way to verify it's set correctly.

Caching — Avoiding Redundant Work

Every job starts on a completely fresh virtual machine (Course 2, Chapter 8) — meaning `npm install` re-downloads every dependency, every single run, unless you cache them.

```
steps:
  - uses: actions/checkout@v4
  - uses: actions/setup-node@v4
    with:
      node-version: '20'
```

```
      cache: 'npm'  # caches node_modules-equivalent based on lockfile hash
  - run: npm ci
```

The `cache: 'npm'` option (built directly into `setup-node`) keys the cache to your lockfile's hash — if `package-lock.json` hasn't changed since the last run, dependencies restore from cache almost instantly instead of re-downloading.

A Complete Deployment Pipeline

Bringing it together — test, then deploy only if tests pass, only on the main branch:

```yaml
name: CI/CD

on:
  push:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with: { cache: 'npm' }
      - run: npm ci && npm test

  deploy:
    needs: test  # waits for "test" to succeed first
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Deploy
        env:
          DEPLOY_TOKEN: ${{ secrets.DEPLOY_TOKEN }}
        run: ./deploy.sh
```

needs:

Makes a job wait for another to finish successfully first — the deploy job here only runs if test passes, never in parallel with it.

Branch-scoped trigger

on: push: branches: [main] means this entire pipeline, including deployment, only fires for pushes to main — feature branches just run their own separate test-only workflow if configured.

MOST HOSTING PROVIDERS HAVE A READY-MADE DEPLOY ACTION — WRITE DEPLOY.SH ONLY AS A LAST RESORT

Vercel, Netlify, AWS, and most major platforms publish official Actions in the Marketplace (Course 2, Chapter 8's tip about not reinventing common tasks) — usually a few lines of `uses:` configuration replaces a hand-written deploy script entirely, and stays maintained as the platform's own deployment process changes.

Reusable Workflows — Avoiding Duplication Across Repos

For teams running near-identical CI across several repositories, a workflow can call another workflow as a reusable building block, rather than copy-pasting the same YAML everywhere.

```
jobs:
  call-shared-tests:
    uses: your-org/shared-workflows/.github/workflows/test.yml@main
```

CHAPTER 5 QUICK REFERENCE

- **strategy: matrix:** runs the same job across multiple parallel variable combinations
- **secrets.<NAME>** — encrypted credentials, injected at runtime, masked in logs (but not infallibly — never deliberately print them)
- **cache: 'npm'** (or equivalent) — skips redundant dependency downloads, keyed to your lockfile hash
- **needs:** — makes one job wait for another to succeed first, the basis of a test-then-deploy pipeline
- **Branch-scoped triggers** — restrict deployment-related jobs to main, keeping feature branches test-only
- **Check the Marketplace first** for deployment to major hosting providers, before hand-writing a deploy script
- **Reusable workflows** — call a shared workflow definition instead of duplicating YAML across repos
- **Next chapter:** branch protection, CODEOWNERS, required reviews, and signed commits — securing a serious repo

CHAPTER 6: SECURING A SERIOUS REPOSITORY

COURSE 3 · CH 6

Securing a Serious Repository

Branch protection in full, CODEOWNERS for automatic review routing, and signed commits for verified authorship

Course 2, Chapter 9 introduced branch protection as the structural fix for force-push dangers. This chapter covers the full set of protection options properly, adds CODEOWNERS for automatically routing reviews to the right people, and covers signed commits — proving a commit genuinely came from who it claims to.

Branch Protection, the Complete Picture

Settings → Branches → Add branch protection rule, applied to `main` (or any branch worth protecting):

Require a pull request before merging

No direct pushes at all (Course 2, Ch9) — every change goes through review.

Require approvals (and dismiss stale ones)

A minimum approval count, with the option to automatically dismiss an approval if new commits are pushed after it — preventing a reviewer's approval from silently covering changes they never actually saw.

Require status checks to pass

Ties to Course 2/3's Actions chapters — a failing CI run physically blocks the merge button.

Require conversation resolution

Every open review comment thread must be marked resolved before merging — stops feedback from being silently ignored.

Require signed commits

Rejects any commit that isn't cryptographically signed — covered fully below.

Restrict who can push

Limits direct push access (where it's allowed at all) to specific people or teams, even among repo collaborators.

Include administrators

Applies every rule above to repo admins too, with no bypass — removes the "but I'm the owner" exception that quietly undermines every other rule if left unchecked.

"INCLUDE ADMINISTRATORS" IS THE RULE MOST TEAMS FORGET

> Setting up strict protection that admins can casually bypass provides much less real protection than it appears to — most accidental force-pushes and skipped-review incidents in practice come from someone with elevated access taking a shortcut "just this once." Checking this option closes that gap.

CODEOWNERS — Automatic Review Routing

A `CODEOWNERS` file (in the repo root, or `.github/`, or `docs/`) maps file paths to the people or teams responsible for them — GitHub automatically requests their review the moment a PR touches a matching path.

```
# CODEOWNERS — paths and their required reviewers

# Default owner for everything not matched below
* @philipo

# Database migrations need a specific reviewer every time
/migrations/ @philipo @db-team

# CI/CD configuration
/.github/workflows/ @philipo
```

Combined with branch protection's "require approval from Code Owners," this turns review routing from a manual, easily-forgotten step ("don't forget to tag someone on the migration!") into something GitHub enforces automatically.

> **CODEOWNERS SCALES WELL EVEN FOR A SOLO MAINTAINER WITH OCCASIONAL CONTRIBUTORS**
>
> Even on a personal project, a CODEOWNERS file ensures that if someone else opens a PR touching a sensitive area (deployment config, database migrations), you're automatically requested as a reviewer — without needing to remember to check every incoming PR's file list manually.

Signed Commits — Proving Authorship

A commit's author name and email (set via `git config` back in Course 1, Chapter 1) are just text fields — trivially easy to set to anything, including someone else's identity. A **signed commit** adds a cryptographic signature, proving the commit was created by someone possessing a specific private key.

Setting up GPG signing

```
$ # Generate a GPG key (if you don't already have one)
$ gpg --full-generate-key

$ # Find the key ID
$ gpg --list-secret-keys --keyid-format=long

$ # Tell git to use it, and sign every commit by default
$ git config --global user.signingkey YOUR_KEY_ID
$ git config --global commit.gpgsign true

$ # Add the public key to GitHub: Settings → SSH and GPG keys → New GPG key
```

From here, every commit gets signed automatically, and GitHub shows a verification badge on the commit:

✓ Verified a1b2c3d Fix login validation bug — Philip Osztromok

Unverified f9e8d7c Some commit with the right name, wrong signature

SSH signing — a simpler alternative

Git also supports signing with the SSH key you may already have set up (Course 1, Chapter 5) — avoiding GPG entirely:

```
$ git config --global gpg.format ssh
$ git config --global user.signingkey ~/.ssh/id_ed25519.pub
$ git config --global commit.gpgsign true
```

SIGNED COMMITS PROVE THE COMMIT, NOT THE CODE'S QUALITY

A signature verifies *who created this commit* — it says nothing about whether the code is correct, secure, or well-reviewed. It's a complement to code review (Course 2, Chapter 7) and branch protection, not a replacement for either. Most valuable in contexts with multiple contributors where impersonation is a genuine, realistic risk — open source projects, regulated environments, anywhere identity verification matters beyond trust alone.

Command Reference

COMMAND	WHAT IT DOES
`git config --global commit.gpgsign true`	Signs every commit automatically going forward
`git config --global user.signingkey <id>`	Sets which key to sign with (GPG key ID or SSH public key path)
`git commit -S -m "msg"`	Signs a single commit explicitly (if not signing by default)
`git log --show-signature`	Shows signature verification status for recent commits

CHAPTER 6 QUICK REFERENCE

- **Branch protection (full set):** require PR, require approvals (dismiss on new commits), require status checks, require conversation resolution, require signed commits, restrict pushers, include administrators
- **"Include administrators"** closes the most commonly forgotten loophole — no bypass for repo owners
- **CODEOWNERS** — maps file paths to required reviewers, automatically requested on matching PRs
- **Signed commits** — cryptographically prove WHO created a commit, via GPG or SSH key signing
- **git config commit.gpgsign true** — signs every future commit automatically
- **A signature proves authorship, not code quality** — complements review and protection, doesn't replace them
- **Next chapter:** large repos — Git LFS, shallow clones, sparse checkout, and monorepo strategies

CHAPTER 7: LARGE REPOS

COURSE 3 · CH 7

Large Repos

Git LFS for big files, shallow clones and sparse checkout for big histories and big trees, and a look at monorepo strategy

Everything in this series so far assumes a repository of a size where `git clone` just works in a few seconds and nobody thinks about it. Past a certain size — large binary assets, years of deep history, or a single repo containing many independent projects — that assumption breaks down, and git offers specific tools for each distinct problem.

Identifying Which Large-Repo Problem You Actually Have

📦 Large binary files

Videos, design files, datasets, compiled assets — git's diffing and storage model handles text well, but stores every full version of a binary separately, bloating repo size fast.

Fix: Git LFS

📜 Deep, long history

Years of commits — cloning the entire history just to start contributing today is unnecessary overhead for most contributors.

Fix: shallow clone

🗂️ Huge file tree

A monorepo with dozens of independent projects — most contributors only ever touch one or two of them, but a normal clone downloads everything.

Fix: sparse checkout

Git LFS — Large File Storage

Git LFS replaces large files in your repo with small text pointers, while the actual file content lives in separate LFS storage — git's normal history-tracking machinery (Chapter 1's object

model) only ever sees the lightweight pointer, not the heavy binary itself.

Git's history tracks a tiny pointer; LFS storage holds the actual large file, fetched on demand

```
$ # One-time setup per machine
$ git lfs install

$ # Tell LFS which file types to track (writes to .gitattributes)
$ git lfs track "*.mp4"
$ git lfs track "*.psd"

$ # From here, normal git commands just work
$ git add .gitattributes video.mp4
$ git commit -m "Add product demo video"
$ git push
```

LFS MUST BE SET UP BEFORE LARGE FILES ARE COMMITTED, IDEALLY

Converting a file already tracked normally into an LFS-tracked one after the fact requires rewriting history (a job for `git lfs migrate`) — every previous commit's full-size version is still sitting in the regular object database otherwise. Setting up `.gitattributes` at the start of a project, before the first large file is ever committed, avoids this entirely.

Shallow Clones — Downloading Only Recent History

```
$ # Only the most recent commit, no deep history at all
$ git clone --depth 1 https://github.com/big-org/huge-project.git

$ # Or the last 50 commits, if you need a bit more context
$ git clone --depth 50 https://github.com/big-org/huge-project.git
```

A shallow clone is dramatically faster for repos with years of history — most CI systems (Course 2/3's Actions chapters) use shallow clones by default, since a workflow run usually only needs

the current state, not the full history.

> **SOME OPERATIONS NEED FULL HISTORY AND WON'T WORK ON A SHALLOW CLONE**
>
> Bisecting (next chapter), inspecting old commits beyond the depth limit, or rebasing against history outside the shallow window all require more history than a shallow clone has. `git fetch --unshallow` converts it to a full clone if you later find you need that history after all.

Sparse Checkout — Downloading Only Part of the Tree

For a monorepo where the full history is needed but only a fraction of the file tree is actually relevant to you, sparse checkout downloads everything's history but only checks out the specific folders you choose into your working directory.

```
$ git clone --filter=blob:none --sparse https://github.com/big-org/monorepo.git
$ cd monorepo
$ git sparse-checkout set frontend/web-app shared/ui-components
/* only these two folders now appear in your working directory */
```

The `--filter=blob:none` flag avoids downloading file contents you haven't checked out yet, fetching them lazily only when actually needed — a meaningful speed difference on a genuinely large monorepo.

Monorepo Strategy — A Brief Orientation

A monorepo holds multiple, often-independent projects in a single repository, rather than splitting each into its own repo (the more traditional approach, and what every previous chapter implicitly assumes).

- **Advantages:** atomic cross-project changes in a single commit/PR, shared tooling and CI config, easier code sharing between projects without the submodule/subtree complexity from Chapter 4.
- **Trade-offs:** repo size and clone time grow with every project added (exactly what this chapter's tools mitigate), and CI needs to be smart enough to only test/build what actually changed rather than the entire repo on every push.

- **Tooling matters more at scale.** Beyond a certain size, dedicated monorepo tools (Nx, Turborepo, Bazel) handle the "only build what changed" problem far better than hand-rolled CI logic — worth researching directly if a monorepo grows large enough that this chapter's git-level tools alone aren't enough.

> **MOST PROJECTS NEVER NEED ANY OF THIS CHAPTER**
>
> Git LFS, shallow clones, and sparse checkout exist specifically for scale problems — a typical project, even a fairly large one, works fine with ordinary clones and normal git usage from every previous chapter. Reach for these tools when an actual, measured problem (slow clones, repo size complaints, genuinely large binary assets) appears, not preemptively.

Command Reference

COMMAND	WHAT IT DOES
`git lfs install`	One-time setup enabling LFS on a machine
`git lfs track "*.ext"`	Marks a file pattern to be stored via LFS, written to .gitattributes
`git clone --depth N`	Shallow clone — only the last N commits of history
`git fetch --unshallow`	Converts a shallow clone into a full one
`git clone --filter=blob:none --sparse`	Clones with full history but no file content yet, ready for sparse checkout
`git sparse-checkout set <paths>`	Chooses which folders actually appear in the working directory

CHAPTER 7 QUICK REFERENCE

- **Git LFS** — for large binary files; stores a lightweight pointer in git, real content in separate LFS storage
- **Set up LFS tracking before committing large files** where possible — converting after the fact needs a history rewrite
- **Shallow clone (--depth N)** — for deep history you don't need; common default for CI runners
- **Some operations (bisect, old-commit inspection) need full history** — git fetch --unshallow if needed later

- **Sparse checkout** — for a huge file tree (monorepo); full history, only checked-out folders you actually need
- **Monorepo trade-off:** easier cross-project changes and sharing, vs. growing repo size and CI complexity
- **Most projects never need this chapter's tools** — reach for them when a real, measured scale problem appears
- **Next chapter:** git bisect — finding exactly which commit introduced a regression

CHAPTER 8: GIT BISECT

COURSE 3 · CH 8

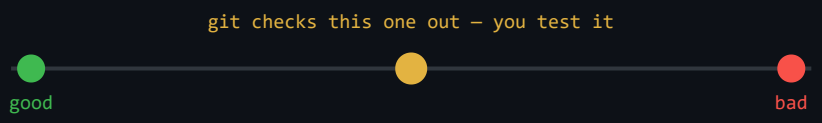
git bisect

Finding exactly which commit introduced a regression, in logarithmic time instead of checking every commit one by one

"This worked last month and doesn't now, but I don't know which of the 200 commits since broke it" is exactly the problem `git bisect` solves. Rather than checking commits one at a time from the start, it performs a binary search through history — finding the exact breaking commit in roughly $\log_2(N)$ steps instead of N.

The Core Idea

You tell git one commit you know was **good** (the bug wasn't present) and one you know is **bad** (the bug is present — often just the current commit). Git checks out a commit roughly halfway between them, and asks you: good or bad? Based on your answer, it narrows the search to one half and repeats — exactly the same algorithm as binary-searching a sorted list.




```
Each answer (good/bad) halves the remaining search space — ~200 commits resolves in about 8 steps, not 200
```

Running a Bisect

1. **Start the bisect**
   git enters bisect mode, tracking the search internally.

2. **Mark a known-bad commit (often just HEAD)**
   The point where you've confirmed the bug exists.

3 **Mark a known-good commit**

Any earlier point you're confident the bug wasn't present — a tagged release (Course 2, Ch5) is often a convenient, reliable choice.

4 **Test the commit git checks out, report good or bad**

Run your app/tests at this exact commit, then tell git the result. Repeat — git narrows the range each time.

5 **Git announces the first bad commit**

Once the range narrows to a single commit, bisect identifies it precisely — the exact commit that introduced the regression.

```
$ git bisect start
$ git bisect bad # HEAD is confirmed broken
$ git bisect good v1.4.0 # this tagged release definitely worked
Bisecting: 47 revisions left to test after this (roughly 6 steps)
[a1b2c3d] Refactor session handling

# Test the app at this checked-out commit...
$ git bisect good # bug NOT present here
Bisecting: 23 revisions left to test after this (roughly 5 steps)

# ...repeat, testing and reporting each time...

f9e8d7c is the first bad commit
commit f9e8d7c
Author: ...
Update token expiry check
```

```
$ # Always finish by returning to where you started
$ git bisect reset
```

DON'T FORGET GIT BISECT RESET

Bisect leaves your working directory checked out at whatever intermediate commit it was testing — not your actual branch. `git bisect reset` returns you to the branch/commit you were on before starting. Forgetting this step is the most common bisect mistake, leaving you confusingly "detached" afterward.

Automating the Test Step

If the "is the bug present" check can be scripted — a specific test passing or failing, a command exiting with a particular code — `git bisect run` automates the entire process, no manual good/bad reporting at all.

```
$ git bisect start
$ git bisect bad HEAD
$ git bisect good v1.4.0
$ git bisect run npm test -- --grep "session expiry"
running npm test -- --grep "session expiry"
...
f9e8d7c is the first bad commit
```

Git runs the given command at each candidate commit, treating exit code 0 as "good" and any non-zero exit code as "bad" — turning a search that might've taken an afternoon of manual testing into something that completes unattended in the time it takes to run the test suite a handful of times.

> **A GOOD "GOOD" COMMIT MATTERS MORE THAN PEOPLE EXPECT**
>
> Choosing a known-good commit that's genuinely far enough back (well before the regression could plausibly have been introduced) avoids accidentally narrowing the search to the wrong half if your assumption about "good" turns out to be wrong. A tagged release you're confident worked correctly, rather than an arbitrary recent commit, is usually the safer starting point.

Command Reference

COMMAND	WHAT IT DOES
`git bisect start`	Begins a bisect session
`git bisect bad [<commit>]`	Marks a commit (default: current) as exhibiting the bug
`git bisect good [<commit>]`	Marks a commit as not exhibiting the bug

COMMAND	WHAT IT DOES
`git bisect run <command>`	Automates testing using a script's exit code instead of manual reporting
`git bisect reset`	Ends the session, returning to your original branch/commit — don't forget this

CHAPTER 8 QUICK REFERENCE

- **git bisect** finds a regression's exact introducing commit via binary search through history
- **~$\log_2$(N) steps**, not N — roughly 8 steps for 200 commits, not 200 individual checks
- **Flow:** bisect start → mark one bad, one good → test each checked-out commit, report good/bad → repeat until git identifies the culprit
- **git bisect run <command>** — fully automates the process if the check can be scripted
- **Always finish with git bisect reset** — returns you from the detached intermediate state to your actual branch
- **Pick a genuinely reliable "good" commit** — a tagged release is usually safer than an arbitrary recent one
- **Next chapter (Scenario):** force-push recovery after a history rewrite goes wrong across a team

CHAPTER 9: SCENARIO - TEAM FORCE-PUSH RECOVERY

COURSE 3 · CH 9 · SCENARIO

Force-Push Recovery Across a Team

When a history rewrite goes wrong with several people already pulled in — coordinating recovery, not just running one command

Course 2, Chapter 9 introduced why force-push is dangerous; Course 3, Chapter 3 covered recovering your own lost commits via the reflog. This chapter combines both into the genuinely hard version: a history rewrite has already gone out to a shared branch, **multiple people have already pulled the bad version**, and recovery now needs coordination, not just a local fix.

The Scenario

YOU	Rebase a shared feature branch to clean up history, then force-push — not realizing two teammates have it checked out with their own local commits on top.
TEAMMATE A	Pulls normally. Git reports a confusing divergence — their local commits don't match anything on the remote anymore, since the rebase gave everything new hashes (Chapter 2).
TEAMMATE B	Already had uncommitted work in progress on top of the old history; their next push is rejected outright.
RESULT	Three different local states, all technically diverged from the rewritten remote, and growing confusion about which version is "correct."

Step 1: Stop — Don't Let Anyone Push Yet

The single most important first action is communication, not commands. A quick message — "I just rewrote history on feature/checkout, please don't push anything to it until I say it's safe" — prevents the situation from getting worse while you sort out the recovery plan.

DON'T LET ANYONE FORCE-PUSH "TO FIX IT" IN A PANIC

The most common way this incident compounds is someone else force-pushing their own version in an attempt to "fix" the divergence, without coordinating first — now there are two competing rewritten histories instead of one, and untangling that is genuinely harder than the original problem.

Step 2: Establish What Everyone Actually Has

1. **Ask everyone NOT to do anything yet, just report their local state**
   git log --oneline -10 from each person, shared in a team channel — establishes the full picture before any commands run.

2. **Identify whose local commits genuinely don't exist anywhere else**
   Anyone with uncommitted or unpushed local work on top of the OLD history has the only copy of that work — their local repo is now the most valuable thing in this recovery, more so than your reflog.

3. **Decide: keep the rewritten history, or restore the original?**
   If the rebase genuinely improved things and only a few people are affected, recovering everyone onto the NEW history is usually less disruptive than reverting your work and starting the rebase over later.

Step 3: Recovering Each Person onto the New History

Assuming the decision is to keep your rewritten history and bring everyone else's unpushed work onto it — for each affected teammate, individually:

```
$ # Teammate identifies which of their local commits are genuinely new (not in the old shared history)
$ git log --oneline origin/feature/checkout..HEAD
4f3e2d1 My genuinely new work, not yet pushed anywhere

$ # Fetch your rewritten history
$ git fetch origin

$ # Cherry-pick just their genuinely new commits onto the new history (Course 2, Ch5)
$ git switch -c feature/checkout-recovered origin/feature/checkout
$ git cherry-pick 4f3e2d1

$ # Verify everything looks right, then this becomes their new feature/checkout
```

CHERRY-PICK IS THE RIGHT TOOL HERE SPECIFICALLY BECAUSE IT DOESN'T CARE ABOUT SHARED ANCESTRY

Course 2, Chapter 5 introduced cherry-pick for backporting a single commit between branches with different histories — exactly the situation here. Each teammate's genuinely new work gets

individually replayed onto the new, correct base, regardless of how different the rest of the history now looks.

Step 4: If You Need to Recover the OLD History Instead

If the team decides the rewrite itself was the mistake and should be undone entirely — your own reflog (Chapter 3) almost certainly still has the pre-rewrite state, since your local repo's reflog isn't affected by what you pushed.

```
$ git reflog
b2c1d0e HEAD@{6}: commit: Last commit before the rebase

$ # Restore the branch to its pre-rewrite state...
$ git reset --hard b2c1d0e
$ # ...and push it back, undoing your own force-push
$ git push --force-with-lease origin feature/checkout
```

IF YOU DON'T HAVE THE OLD HISTORY IN YOUR REFLOG EITHER, CHECK TEAMMATES' LOCAL COPIES

Anyone who pulled the old history before your force-push, and hasn't reset or rebased locally since, still has that old history sitting in their own repo, completely intact — even if your own reflog has already expired or been pruned. Their local branch is a full, valid backup of the state you're trying to restore.

Preventing the Next One

- **Branch protection's "block force pushes"** (Course 2/3, Chapter 9/6) on genuinely shared branches removes this entire category of incident as a possibility.
- **Reserve interactive rebase and history rewriting** (Chapter 2) for branches you're confident are still personal, even if technically pushed somewhere.
- **If a shared branch genuinely needs cleanup, communicate first, every time** — the actual technical recovery in this chapter is straightforward; the damage comes from skipping the coordination step.

CHAPTER 9 QUICK REFERENCE

- **First action: communicate, don't compute** — stop anyone else from pushing or "fixing it" before you have a plan
- **Establish what everyone has** before running any recovery commands — unpushed local work is the most valuable thing in play
- **Cherry-pick each person's genuinely new commits** onto the corrected history — works regardless of how different the rest of the history now looks
- **To undo the rewrite entirely:** your own reflog (Ch3), or any teammate's untouched local copy if yours has expired
- **git push --force-with-lease** (not plain --force) when pushing the restored history back
- **Prevention:** branch protection blocking force pushes, reserving rewrites for genuinely personal branches, communicating before any history-rewriting operation on shared work
- **Next chapter (final, Scenario):** full repo migration — splitting/merging repos while preserving history, moving hosts, handling LFS/submodules

CHAPTER 10: SCENARIO - FULL REPO MIGRATION

COURSE 3 · CH 10 · FINAL CHAPTER · SCENARIO

Full Repository Migration

Splitting and merging repos while preserving history, moving between hosts, and handling LFS/submodules along the way

This final chapter is the natural endpoint of the series — and a direct, larger-scale relative of Course 1, Chapter 9's "moving your project folder" scenario. Instead of moving a folder on disk, this is about restructuring entire repositories: splitting one into several, merging several into one, or moving a whole project to a different hosting provider — all while keeping the history that makes git useful in the first place.

The Three Migration Types

✂️ Splitting a repo

Extracting one subfolder of a large repo into its own standalone repository, ideally keeping that folder's commit history intact rather than starting fresh.

🔗 Merging repos

Combining two previously separate repositories into one — the reverse problem, and the legitimate use case behind Chapter 4's `git subtree`.

🏠 Moving hosts

GitHub → GitLab, self-hosted → GitHub, or similar — conceptually the simplest of the three, but with real gotchas around what doesn't transfer automatically.

Splitting a Repo with History Intact — git filter-repo

Naively copying a subfolder into a new repo and running `git init` loses everything — every commit, every "why was this written this way" buried in old commit messages. `git filter-`

`repo` (the modern, recommended replacement for the older, slower `git filter-branch`) rewrites history to extract just one path's story.

```
$ # Clone a fresh copy specifically for this operation — filter-repo rewrites destructively
$ git clone https://github.com/you/big-monorepo.git extracted-project
$ cd extracted-project

$ # Keep ONLY history relevant to this one subfolder
$ git filter-repo --path frontend/web-app
Parsed 1,847 commits
New history written in 4.2 seconds...

$ # This clone's history now ONLY contains commits touching that path,
$ # with paths rewritten as if it had always been the repo root
$ git log --oneline | wc -l
312 # only the commits that actually touched frontend/web-app
```

```
$ # Point at a brand-new, empty GitHub repo and push the extracted history
$ git remote add origin https://github.com/you/web-app.git
$ git push -u origin main
```

FILTER-REPO REWRITES EVERY COMMIT HASH — WORK ON A THROWAWAY CLONE, NEVER YOUR ONLY COPY

Like every history-rewriting tool in this course (interactive rebase, Chapter 2), filter-repo creates entirely new commit objects. Run it on a fresh clone made specifically for the migration, never on the team's actual working clone — if anything goes wrong, you simply delete the throwaway clone and start again from the untouched original.

Merging Repos While Preserving Both Histories

Chapter 4's `git subtree` command is the practical tool here — merging another repository's full history into a subfolder of your current one, rather than losing it.

```
$ git subtree add --prefix=libs/old-utils https://github.com/you/old-utils-repo.git main
```

Without `--squash` (contrast with Chapter 4's earlier example), this preserves the full individual commit history of the merged-in repository, interleaved into your main repo's own history — useful when that history itself still has ongoing value, not just the current file state.

Moving Between Hosting Providers

1. **Mirror-clone the existing repo**
   `git clone --mirror <old-url>` — captures every branch, tag, and ref exactly, not just the default branch a normal clone would give you.

2. **Create the new, empty repository on the destination host**
   No README, no initial commit, no .gitignore template — genuinely empty, so the push doesn't conflict with anything.

3. **Push the mirror to the new host**
   `git push --mirror <new-url>` — transfers every branch, tag, and the complete commit history.

4. **Migrate the things git history doesn't carry**
   Issues, PR discussions, wiki content, branch protection settings, GitHub Actions secrets — see the checklist below.

What doesn't transfer automatically

⚠️ **Issues and PR discussion history**
Lives in the platform's own database, not in git itself — needs its own export/import process (GitHub's API, or third-party migration tools) if that history matters.

⚠️ **Branch protection rules and CODEOWNERS enforcement**
CODEOWNERS the file transfers with the repo (Chapter 6), but the platform-level protection rules wrapping it need recreating on the new host.

⚠️ **Actions secrets and CI configuration**
Secrets (Chapter 5) never transfer between hosts even in principle — they're encrypted, platform-specific, and need re-entering manually on the new side.

⚠️ **Git LFS objects**

`--mirror` doesn't automatically carry LFS-tracked file content (Chapter 7) — run `git lfs fetch --all` then `git lfs push --all <new-remote>` explicitly.

⚠ **Submodule references**

If a submodule's own URL (Chapter 4) is also moving, .gitmodules needs updating to point at its new location too — otherwise contributors hit exactly the kind of broken-reference confusion this whole series has returned to repeatedly.

KEEP THE OLD HOST READ-ONLY, DON'T DELETE IT, FOR A TRANSITION PERIOD

Leaving the original repository accessible (even just as an archived, read-only reference) for a few weeks after migrating gives anyone who missed the announcement a way to find the new location, and provides a safety net if something about the migration turns out incomplete.

Command Reference

COMMAND	WHAT IT DOES
`git filter-repo --path <path>`	Rewrites history to keep only commits touching the given path
`git subtree add --prefix=<path> <url> <branch>`	Merges another repo's full history into a subfolder (no --squash)
`git clone --mirror <url>`	Clones every branch, tag, and ref exactly — for host migration
`git push --mirror <url>`	Pushes a complete mirror to a new host
`git lfs fetch --all / push --all`	Explicitly transfers LFS object content, which --mirror alone doesn't carry

CHAPTER 10 QUICK REFERENCE — AND SERIES WRAP-UP

- **git filter-repo --path** — extracts one subfolder's history into a standalone repo; always on a throwaway clone
- **git subtree add** (no --squash) — merges another repo's full history into a subfolder, preserving both
- **git clone --mirror / git push --mirror** — the correct pair for moving a complete repo between hosts

- **Doesn't transfer automatically:** issues/PR history, branch protection settings, Actions secrets, LFS objects (needs explicit lfs fetch/push), submodule URLs pointing at old locations
- **Keep the old host accessible**, read-only, for a transition period after migrating
- **Series recap (Course 3):** internals (Ch1) → interactive rebase (Ch2) → reflog/recovery (Ch3) → submodules/subtrees (Ch4) → advanced Actions (Ch5) → securing a repo (Ch6) → large repos (Ch7) → bisect (Ch8) → team force-push recovery (Ch9) → full migration (Ch10)
- **Full series complete:** 30 chapters across Foundations, Teams, and Advanced Mastery — from "what is git" to recovering a team from a history-rewrite disaster