Git
Made Crystal Clear

A practical, visual guide to version control with Git

CHRISTOPHE PAKA

Git

MADE CRYSTAL CLEAR

A practical, visual guide to version control with Git

Christophe Paka

First Edition · 2026-08-09

Git Made Crystal Clear · First Edition

A practical, visual guide to version control with Git

Author: Christophe Paka
Published 2026-08-09.

More books in the Made Crystal Clear series: https://madecrystalclear.dev

Sources: https://git-scm.com/ and the official Git documentation.

How to read this book

This is a hands-on, visual guide. Every chapter opens with **The gist**, explains a concept with a diagram before the prose, and closes with **Key takeaways** and a **Go deeper** link list. Recipes are numbered, copy-pasteable, and tell you exactly what a successful result looks like. Watch for these boxes as you read:

⚡ **The gist**
The 3 to 5 things a chapter delivers, up front.

📝 **Recipe**
Numbered steps with commands and a ✅ expected result to match.

📋 **Checklist**
Actionable items to tick off before moving on.

💡 **Tip**
A shortcut or clarification worth knowing.

⚠️ **Watch out**
A common mistake and how to avoid it.

💼 **Business angle**
Why the topic matters to cost, speed, or risk.

📌 **Key takeaways**
The chapter's summary, at the end.

🔗 **Go deeper**
The official docs and sources behind the chapter.

> ⚠️ **Watch out**
> This is a hands-on guide. Every command here is safe to try on a throwaway repository, and Chapter 8 shows how to undo almost anything. One rule keeps you out of trouble: never rewrite history you have already shared or pushed. Chapter 7 explains why, Chapter 8 shows how to recover.

Contents

1. About This Book

The gist

- This book teaches Git from zero: no prior version control needed, just a terminal and files you care about.
- By the end you can commit, branch, merge, work with remotes, rewrite local history safely, and undo almost anything.
- Twelve chapters, arranged as a learning path in four legs: Foundations, Hands-on, Team and scale, Reference.
- One safety rule runs through the whole book: never rewrite history you have already shared, and lean on the reflog when things go wrong.

Reading time: ~6 min

Who this is for

If you write code, notebooks, configs, or docs, this book is for you. It is aimed at developers, data and ML engineers, and anyone whose work lives in text files that change over time. You do not need any prior version control experience. If you are comfortable editing a file and running a command in a terminal, you are ready.

Git has a reputation for being confusing, and most of that confusion comes from learning commands without the mental model underneath them. This book fixes that order: you get the picture first, then the commands land in place. We keep prose short, put a diagram in every chapter, and give you recipes you can copy, run, and verify.

What you will be able to do

After working through these chapters, you will be able to set up Git, track a project, branch and merge without panic, push and pull with a team, clean up your local commit history, and recover from mistakes that would otherwise cost you a day.

The road ahead

The twelve chapters form a single path. You can read straight through, or jump to the leg you need. Figure 1.1 shows how they group.

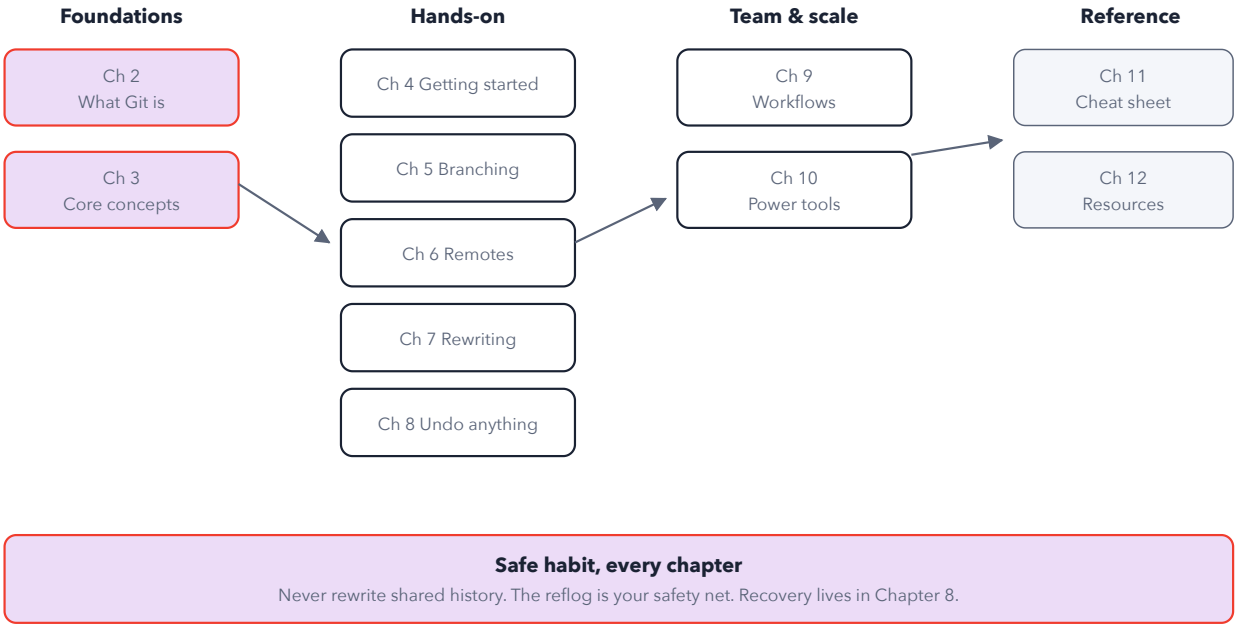


Figure 1.1 — Book roadmap: the twelve chapters as one learning path in four legs.

Foundations (Chapters 2 to 3) give you the mental model: what Git actually is, why snapshots beat diffs, and the three trees plus the object model that explain every command that follows.

Hands-on (Chapters 4 to 8) is the working core: install and first commit, branching and merging, remotes, rewriting local history, and undoing anything. This is where you build muscle memory.

Team and scale (Chapters 9 to 10) take you from solo to shared: collaboration workflows, pull requests, and the power tools for debugging history and handling large repositories.

Reference (Chapters 11 to 12) is what you keep open on a second monitor: a one-page cheat sheet with a 30/60/90-day path to fluency, and every official link in one place.

📋 What you need before you start

- [ ] A computer with a terminal (macOS Terminal, a Linux shell, or Git Bash / PowerShell on Windows).
- [ ] Comfort editing a text file and running a command by hand.
- [ ] Git installed, or the willingness to install it in Chapter 4 (we walk through it).
- [ ] A small project or folder you can practice on safely.
- [ ] Optional: a free account on a hosting service such as GitHub or GitLab for the remotes chapter.

📌 Key takeaways

- This book assumes no prior version control, only a terminal and files worth tracking.
- Twelve chapters in four legs: Foundations, Hands-on, Team and scale, Reference.
- Read straight through to learn, or jump to a leg when you need a specific skill.
- The through-line: never rewrite shared history (Chapter 7 explains why, Chapter 8 shows how to recover), and trust the reflog.

🔗 Go deeper

- Pro Git (git-scm.com) – the free, official book; a deeper companion to this guide.
- Git downloads – installers for macOS, Windows, and Linux.
- Git reference manual – the official docs for every command referenced in this book.

2. What Git Is, and Why It Won

> ⚡ **The gist**
>
> - Git is a distributed version control system: every clone is a full repository with the complete history.
> - Git stores **snapshots, not diffs** — each commit points to a full tree, and unchanged files reuse the same content.
> - Every commit is content-addressed by a hash, so history is tamper-evident and nearly every operation is fast and local.
> - Git won on cheap branching, offline work, speed, and the network effect of hosting platforms.
>
> *Reading time: ~8 min*

The pain Git removes

Before version control, a project's history lives in fear. You duplicate the folder before a risky change. You email a zip file and hope no one edited their copy. You end up with a drawer full of names like `report_final.zip`, `report_final_v2.zip`, and the immortal `final_v3_FINAL.zip` — and no idea which one is actually current.

That way of working costs you in four concrete ways:

- **Lost work.** A file gets overwritten and the previous version is simply gone.
- **No safe experiments.** Trying a bold idea means risking the working copy, so you do not try.
- **No blame or audit.** When a line breaks, there is no record of who changed it, when, or why.
- **Painful collaboration.** Merging everyone's zip files by hand is slow and error-prone.

A version control system fixes all four by recording history as a series of deliberate save points you can inspect, compare, and return to.

Git in one sentence

Git is a **distributed version control system (DVCS)**. "Distributed" is the key word: when you clone a Git project you receive a full repository with its complete history, not a thin checkout borrowed from a central server. Git was created by **Linus Torvalds in 2005** to manage development of the Linux kernel.

Because every clone is complete, you can commit, branch, view history, and compare versions without talking to any server. As the Pro Git book puts it, "nearly every operation is local."

Snapshots, not diffs

Most older systems store history as a list of changes (deltas) to files over time: version 1, then the difference to version 2, then the difference to version 3, and so on. Git works differently. It thinks of its data "more like a series of snapshots of a miniature filesystem."

Every time you commit, Git records a full *tree* — a snapshot of what every file looked like at that moment. If a file did not change since the last commit, Git does not store it again; the new snapshot simply **points to the identical previous copy**. So snapshots are cheap, and reconstructing any past version is just reading one tree rather than replaying a chain of diffs.

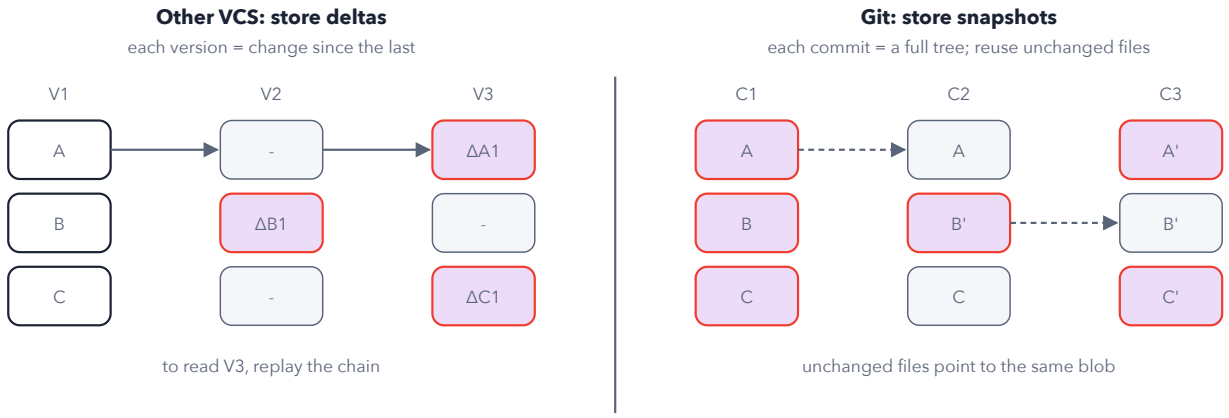


Figure 2.1 — Snapshots vs diffs. Other systems store the change per version and replay the chain to rebuild a file; Git stores a full snapshot per commit and lets unchanged files reference the identical previous copy.

Git has integrity

Everything in Git is content-addressed by a hash (a SHA checksum; historically SHA-1, with SHA-256 support in progress). A commit's identifier *is* the hash of its content, including a pointer to its parent. Change so much as one byte anywhere in the history and every hash downstream would change too. This makes Git history **tamper-evident**: you cannot quietly alter the past, and corruption is detected rather than silently propagated. In short, "Git has integrity."

Centralized vs distributed

Older centralized systems keep the one true repository on a server. Your working copy is a thin checkout, and to commit, branch, or read history you generally need the server. In Git, every peer holds a full repository. You commit locally, branch locally, and read the entire history locally; syncing with others is a separate, explicit step.

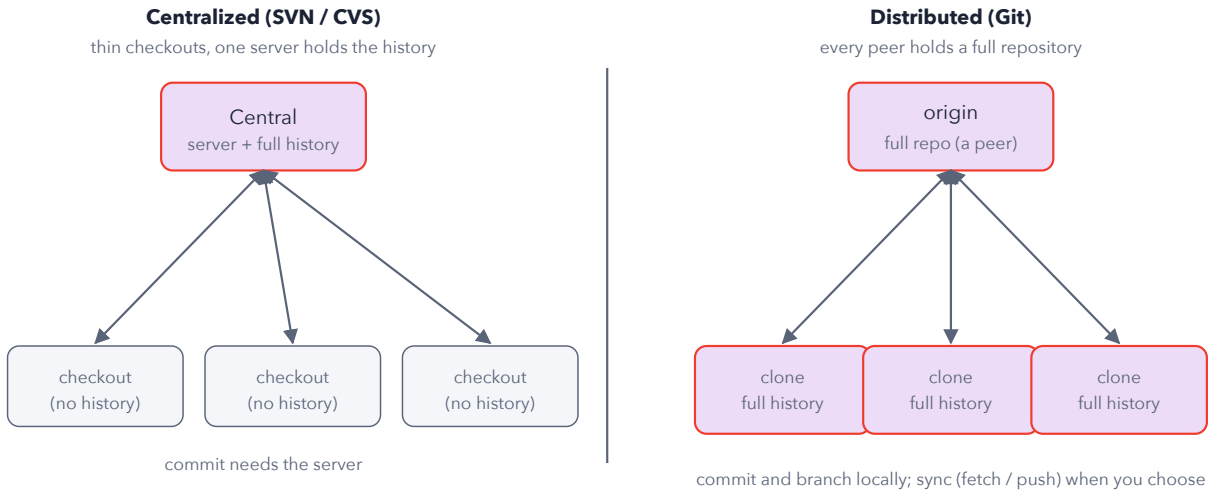


Figure 2.2 — Centralized vs distributed. A central server hands out thin checkouts and mediates every commit; in Git each clone is a complete repository, so work happens locally and syncing is a separate, explicit step.

Why Git won

Git was not the only version control system, and it was not the first to be distributed. It won because of a combination of technical strengths and timing:

- **SVN and CVS** are centralized: you need server access to commit, and their branching is comparatively weak and heavyweight.
- **Mercurial** is also distributed and has a simpler command line, but it grew a smaller ecosystem.
- **Perforce** is strong for very large binary and game assets, a niche where Git historically struggled.

Against that field, Git won on **cheap branching** (a branch is almost free to create), **offline work** (a full repo in your pocket), **speed** (nearly every operation is local), and the **network effect** of the hosting platforms — GitHub, GitLab, and Bitbucket — built on top of it. Today Git is the de-facto standard used by

the vast majority of professional developers.

> 💼 **Business angle**
>
> Standardizing on Git pays back on four fronts. **Speed**: local operations keep developers in flow instead of waiting on a server. **Cheap branching**: experiments and features get their own branch for free, so bold changes are low-risk. **Offline work**: a full history in every clone means productivity does not depend on the network. **Auditability**: hash-chained, tamper-evident history gives you blame, review, and painless rollback – reliability that compounds across a whole team.

Is Git right for this project?

For source code, docs, config, and most text-based work the answer is almost always yes. Use this quick check:

- [ ] The work is mostly text (code, docs, config) that benefits from line-by-line history.
- [ ] More than one person, or more than one machine, needs to collaborate or stay in sync.
- [ ] You want to experiment safely on branches without endangering the working version.
- [ ] You need an auditable record of who changed what, when, and why.
- [ ] You want to work offline and sync later, rather than depend on a central server.
- [ ] If the project is dominated by very large binary assets, plan for Git LFS or evaluate a tool like Perforce for that part.

> 📌 **Key takeaways**
>
> - Git is a distributed version control system created by Linus Torvalds in 2005; every clone is a full repository with the complete history.
> - It stores snapshots, not diffs: each commit references a full tree, and unchanged files reuse the identical previous copy.
> - Content-addressing by hash makes history tamper-evident, and nearly every operation runs locally and fast.
> - Git beat SVN/CVS, Mercurial, and Perforce on cheap branching, offline work, speed, and the network effect of hosting platforms.

> 🔗 **Go deeper**
>
> - Pro Git 1.3 – What is Git? – snapshots, the three states, integrity, and "nearly every operation is local."
> - Pro Git (free book) – the full reference behind this chapter.
> - Git home and reference manual – official documentation.

3. Core Concepts: The Three Trees and the Object Model

⚡ The gist

- Git moves your work through three areas: the working directory, the staging area (index), and the repository.
- `git add` and `git commit` push work forward; `git restore` and `git switch` pull it back.
- Under the hood there are just four object types (blob, tree, commit, tag), each named by the hash of its own content.
- A branch is only a movable pointer to a commit, which is why branching is instant.

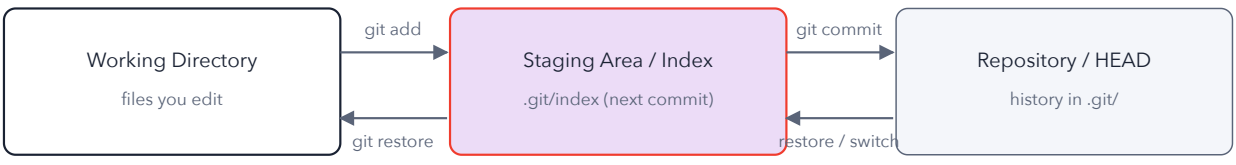
Reading time: ~9 min

The three trees

Almost every Git command you run moves your work between three areas. Get this picture right and commands like `add`, `commit`, and `restore` stop being magic and become predictable.

The **working directory** is the set of files you actually edit. The **staging area**, also called the **index**, lives in `.git/index` and holds the proposed next commit: the exact snapshot you are about to record. The **repository** is the committed history stored in `.git/`, and **HEAD** points at the latest committed snapshot.

Work flows forward with two commands. `git add` copies changes from the working directory into the index. `git commit` records whatever is in the index as a permanent snapshot in the repository. Work also flows backward: `git restore` pulls a file from the index (or a commit) back into the working directory, and `git switch` moves you between committed states.



Forward: stage, then commit. Backward: restore a file, or switch to another committed state.

Figure 3.1 – The three trees: work moves forward with `add` and `commit`, and back with `restore` and `switch`.

The everyday loop

In daily use you cycle through the same four commands. Check what changed, stage what you want, record it, and review the history.

```
git status                # what changed, what is staged
git add file.txt          # stage a file into the index
git commit -m "message"   # record the staged snapshot
git log --oneline         # review the history
```

`git status` is the compass: it tells you which files sit in the working directory, which are staged in the index, and how far ahead your history is. When you can read `status` against the three trees above, you always know where your work is.

The object model

Underneath the three trees, Git stores everything as just four kinds of object in `.git/objects`. Each object is named by the SHA hash of its own content, so identical content is stored once and any tampering changes the name.

Object	What it is	Points to
blob	The contents of a file (no name, no path)	Nothing; it is raw data
tree	A directory: a list of names + modes	Blobs and sub-trees
commit	A snapshot plus author, committer, and message	One tree + parent commit(s)
tag	An annotated, named pointer with its own message/author	One object (usually a commit)

So a commit does not store your files directly. It points to *one* tree (the top directory of your project), which points to blobs (file contents) and sub-trees (sub-directories). Because every object is content-addressed by hash, the whole chain is tamper-evident: change one byte in a file and its blob hash changes, which changes the tree, which changes the commit ID.

Branches, HEAD, and the DAG

Commits are linked into a **directed acyclic graph (DAG)**: each commit records its parent(s), so history is a chain that only ever points backward. A normal commit has one parent; a merge commit has two or more.

On top of that graph sit **refs**, which are just pointers to commits: branches live in `refs/heads/*`, remote-tracking branches in `refs/remotes/*`, and tags in `refs/tags/*`. A **branch is simply a movable pointer**: a small file containing one commit hash. That is why creating a branch is instant, no matter how big the project. **HEAD** is the pointer to your current branch (or directly to a commit, when it is "detached").

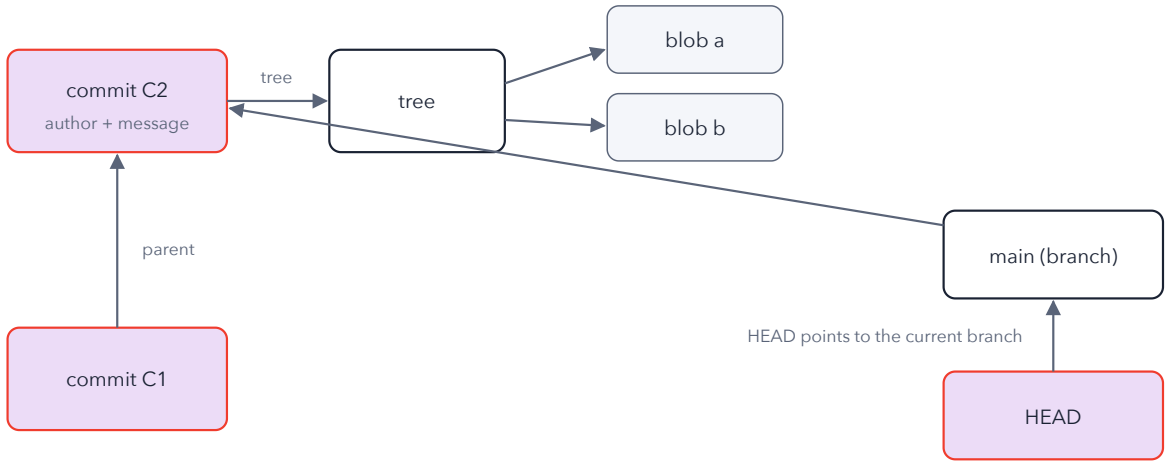


Figure 3.2 — Object model and a small DAG: a commit points to one tree (which points to blobs) and to its parent commit; a branch is a pointer to a commit, and HEAD points to the branch.

> 💼 **Business angle**
>
> Because a branch is a 41-byte pointer rather than a copy of the project, teams can spin up a branch per feature, experiment, or bug fix at zero cost. Cheap branching is what makes safe experimentation and fast review cycles practical at scale.

📋 The mental model in five sentences

- [ ] Your work lives in three areas: working directory, staging area (index), and repository.
- [ ] `git add` stages into the index; `git commit` records the index as a snapshot; `git restore` and `git switch` move work back.
- [ ] Everything is stored as four object types (blob, tree, commit, tag), each named by the hash of its content.
- [ ] A commit points to one tree and to its parent commit(s), so history is a directed acyclic graph.
- [ ] A branch is just a movable pointer to a commit, and HEAD points to your current branch.

📌 Key takeaways

- The three trees (working directory, index, repository) explain what every everyday command actually does.
- Blobs hold contents, trees hold directories, commits hold snapshots, and tags name releases.
- Content-addressing by hash is what gives Git its integrity: change anything and the ID changes.
- Branching is instant because a branch is only a pointer, and HEAD tracks where you are.

🔗 Go deeper

- Pro Git – Git Internals: Git Objects – blobs, trees, and commits in depth.
- Pro Git – Branches in a Nutshell – why a branch is a movable pointer, and how HEAD works.
- Pro Git – Reset Demystified – the three trees explained through `reset`.

4. Getting Started: Install, Configure, First Commit

> ⚡ **The gist**
>
> - Install Git, then tell it who you are with four one-time config commands.
> - `git init` makes a fresh repo; `git clone` copies an existing one with its full history.
> - The daily loop is small: edit, `git add`, `git commit`, and read progress with `git status` and `git log`.
>
> *Reading time: ~8 min*

Install Git

Download the installer for your operating system from git-scm.com/downloads and follow the prompts. When it finishes, open a terminal and confirm the install printed a version:

```
git --version
```

If you see a version number, Git is on your machine and on your `PATH`. That is all you need to begin.

First-time configuration

Git stamps every commit with a name and email, so set those once before you commit anything. While you are here, pick `main` as the default branch name and tell Git which editor to open when it needs a message from you.

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
git config --global init.defaultBranch main
git config --global core.editor "code --wait"
```

Where settings live: the three config levels

Git reads configuration from three files, and the most specific one wins. A value set in `--local` overrides `--global`, which overrides `--system`.

Level	Flag	File	Applies to
System	`--system`	system Git config	all users on the machine
Global	`--global`	`~/.gitconfig`	your user account
Local	`--local`	`.git/config`	this one repository (wins)

Use `--global` for your identity and editor. Use `--local` when one repo needs a different setting, such as a work email on a client project.

init vs clone: two ways to start

Every project begins one of two ways. If the code does not exist yet, `git init` turns the current folder into a brand-new, empty repository. If the project already lives on a server, `git clone <url>` downloads a full local copy, including its entire history.

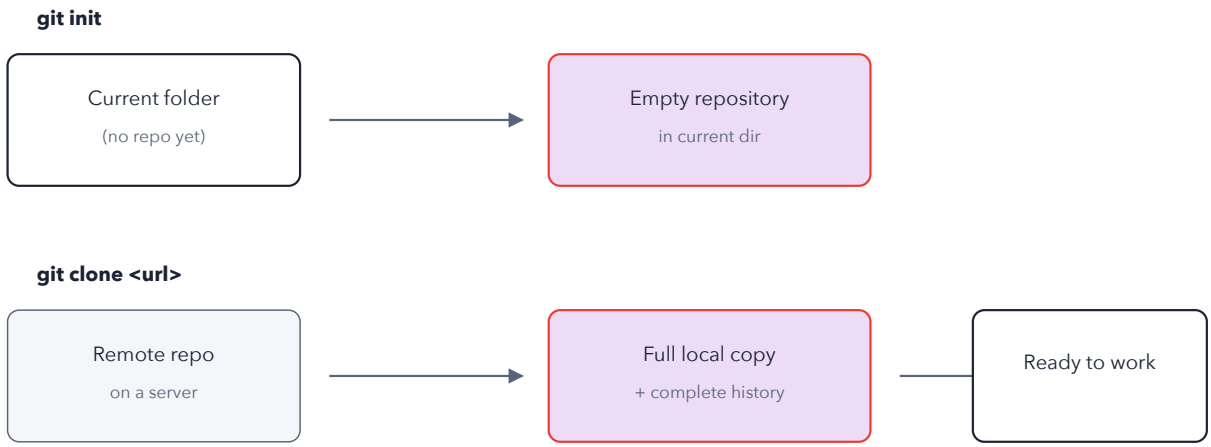


Figure 4.1 — init starts an empty repo in the current folder; clone copies a remote repo with its full history.

The add / commit loop

Day to day, Git work is a short rhythm. You edit files, *stage* the changes you want to keep with `git add`, then *record* them as a snapshot with `git commit`. Between each step, `git status` tells you exactly where things stand.

```
git status                # what changed and what is staged
git add file.txt          # stage one file
git add -A                # stage everything, including deletions
git add -p                # stage selected hunks interactively
git commit -m "message"   # record the staged snapshot
git commit -am "message"  # stage tracked changes and commit in one step
```

> 💼 **Business angle**
>
> Those four config lines take under a minute, but they make every future commit auditable: correct authorship, a consistent default branch, and a known editor mean less rework and cleaner history across the whole team.

Recipe: your first repository

📝 Recipe 4.1 — Zero to first commit

Prerequisites: Git installed and configured (name and email set).

1. Make a project folder and move into it:

```
mkdir my-project
cd my-project
```

2. Turn it into a Git repository:

```
git init
```

3. Create a file to track:

```
echo "# My Project" > README.md
```

4. See what Git noticed:

```
git status
```

5. Stage the new file:

```
git add README.md
```

6. Record your first commit:

```
git commit -m "Initial commit"
```

✅ Expected result: `git log --oneline` shows your commit with its short hash and message.

Recipe: copy an existing project

📝 Recipe 4.2 – Clone an existing repo

Prerequisites: Git installed and the repository URL.

1. Clone the repository (this creates a folder and downloads the full history):

```
git clone https://example.com/repo.git
```

2. Move into the new folder:

```
cd repo
```

3. Confirm the history came with it:

```
git log
```

✅ Expected result: `git log` shows the project's existing commit history, not an empty repo.

Reading what happened

Two commands answer almost every "what is going on?" question. `git log` shows the commits that are already recorded, and `git diff` shows changes that are not committed yet.

```
git log --oneline --graph --decorate   # compact, visual history
git diff                               # working directory vs the index
git diff --staged                      # index vs the last commit
```

Reach for `git diff` before you stage, and `git diff --staged` to review exactly what a `git commit` is about to capture.

> ⚠️ **Watch out**
>
> If a commit is refused with "Author identity unknown" or "please tell me who you are," your `user.name` or `user.email` is not set. Run the two `git config --global` identity commands from the setup section, then commit again.

📋 First-run checklist

- [ ] `git --version` prints a version number.
- [ ] `user.name` and `user.email` are set with `git config --global`.
- [ ] `init.defaultBranch` is set to `main`.
- [ ] I created a repo with `git init` and made a first commit.
- [ ] `git log --oneline` shows that commit.
- [ ] I can read pending changes with `git status` and `git diff`.

📌 Key takeaways

- Install from git-scm.com, verify with `git --version`, then set identity, default branch, and editor once with `git config --global`.
- Config has three levels, and the most specific wins: `--local` beats `--global` beats `--system`.
- Start a project with `git init`; join one with `git clone`.
- The loop is `git add` then `git commit`; inspect with `git status`, `git log`, and `git diff`.

🔗 Go deeper

- git-scm.com/downloads – official installers for every platform.
- Pro Git: First-Time Git Setup – the source for the config commands and levels.
- Pro Git: Getting a Git Repository – init vs clone and the add/commit loop.
- git config docs – the full reference for `--system`, `--global`, and `--local`.

5. Branching and Merging Without Fear

> ⚡ **The gist**
>
> - A branch is just a lightweight movable pointer to a commit, so creating one is instant.
> - Use `git switch -c feature` to branch, `git merge feature` to bring work back.
> - Merges are either a fast-forward (pointer moves) or a three-way merge (a merge commit with two parents).
> - Conflicts are normal: Git marks them, you edit, then `git add` and `git commit`.
>
> *Reading time: ~9 min*

A branch is a pointer

A branch in Git is simply a lightweight movable pointer to a commit. It is not a copy of your files and it is not a folder. It is a tiny reference that holds one commit ID, which is why creating a branch is instant no matter how large the project is.

HEAD is the pointer that says which branch you are on right now. When you commit, the current branch pointer moves forward to the new commit, and HEAD moves along with it. Because commits link back to their parents, the history forms a directed acyclic graph (DAG): each commit points to its parent, and a merge commit points to two parents.

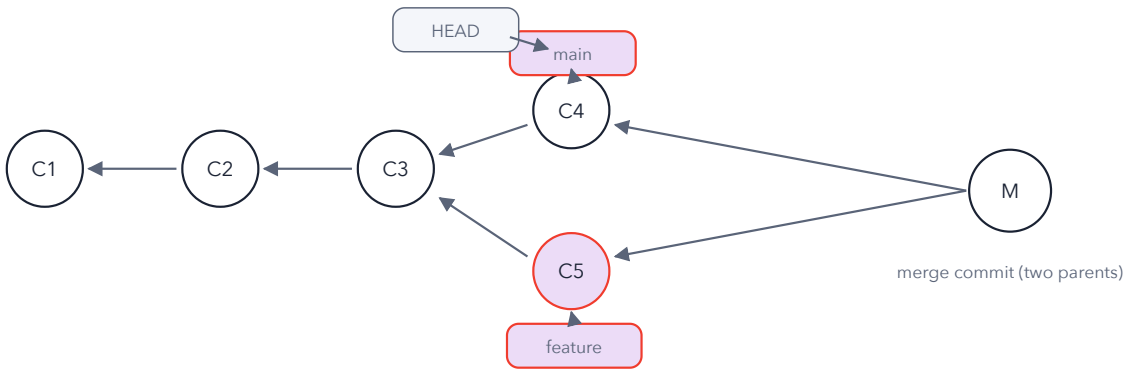


Figure 5.1 — Branches on the commit DAG. Each commit points back to its parent; `main` and `feature` are movable pointers, HEAD marks the current branch, and merge commit M has two parents.

Create, switch, and list branches

The modern command to make a branch and move onto it in one step is `git switch -c`. To move between existing branches, use `git switch`. To see what branches exist, use `git branch`.

```
git switch -c feature   # create + switch (older: git checkout -b feature)
git switch main         # switch back (older: git checkout main)
git branch              # list branches
```

Because a branch is only a pointer, switching does not touch your history. It just repoints HEAD and updates the files in your working directory to match the branch you moved to.

> 💼 **Business angle**
>
> Cheap branches change how a team works. When starting an experiment costs nothing and throwing it away costs nothing, people try more ideas, review smaller units of work, and ship with less risk. Safe experimentation compounds across a team into faster, calmer delivery.

Merging: fast-forward vs three-way

To bring another branch's work into the branch you are on, switch to the target branch and run `git merge`. What happens next depends on whether history diverged.

A **fast-forward** happens when the current branch is a direct ancestor of the branch you are merging: nothing diverged, so Git just slides the pointer forward. No merge commit is created. A **three-way merge** happens when both branches have new commits since they split: Git combines the two lines and records a merge commit with two parents. If you want a merge commit even when a fast-forward is possible, use `git merge --no-ff`.

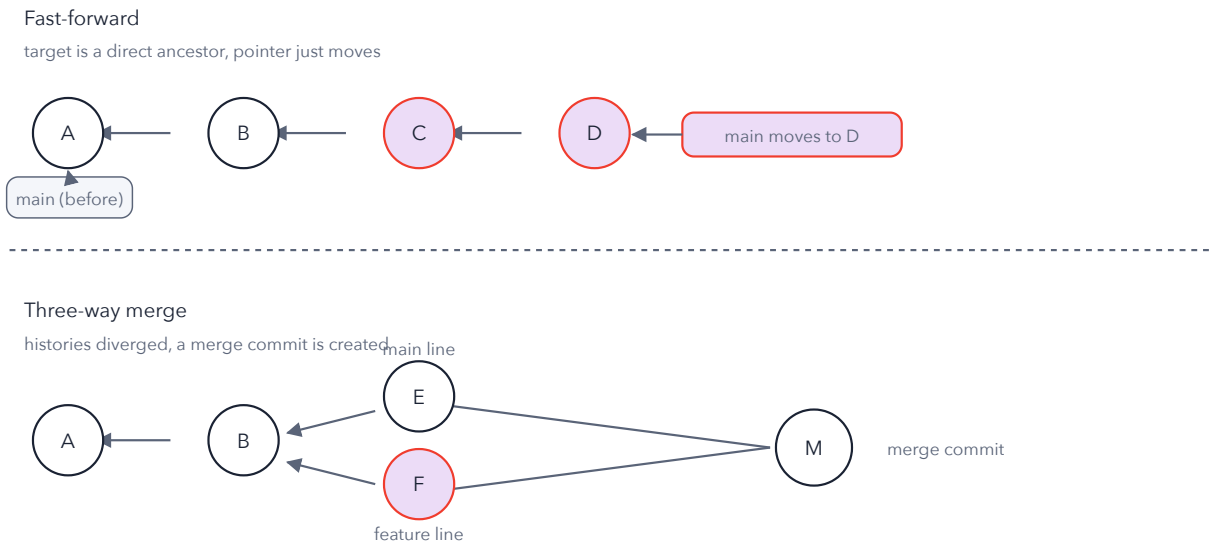


Figure 5.2 – Fast-forward vs three-way merge. Top lane: no divergence, so `main` simply moves forward. Bottom lane: both lines advanced, so Git records merge commit M with two parents.

Deleting branches

Once a branch is merged, its pointer has done its job. Delete it to keep the branch list tidy.

```
git branch -d feature   # delete a merged branch
git branch -D feature   # force-delete (even if not merged)
```

When a merge conflicts

A conflict happens when both branches changed the same lines and Git cannot decide which version wins. It does not lose anything. It stops, marks the spots in the file, and hands the decision to you. Git writes the two versions between conflict markers:

```
<<<<<<< HEAD
your version of the lines
=======
the other branch's version
>>>>>>> feature
```

You edit the file so it reads the way you want, remove the `<<<<<<<`, `=======`, and `>>>>>>>` lines, then stage and commit. If you would rather back out entirely, `git merge --abort` returns you to the state before the merge.

Recipe: branch, commit, merge

📝 Recipe 5.1 — Branch, commit, merge

Prerequisites: a repo with at least one commit on `main` (see Chapter 4).

1. Create a branch and move onto it:

```
git switch -c feature
```

2. Make a change, then stage and commit it:

```
git add -A
git commit -m "Add feature work"
```

3. Switch back to the branch you want to merge into:

```
git switch main
```

4. Merge the feature branch into `main`:

```
git merge feature
```

5. Delete the merged branch to tidy up:

```
git branch -d feature
```

✅ Expected result: `git log --graph --oneline` shows the feature commit now part of `main`'s history.

⚠️ Watch out

`git branch -D` force-deletes a branch even when its commits were never merged, which can strand work. Use the lowercase `-d` for everyday cleanup: it refuses to delete a branch that has not been merged, so it protects you from throwing away unmerged commits by mistake.

Recipe: resolve a merge conflict

📝 Recipe 5.2 – Resolve a merge conflict

Prerequisites: two branches that both changed the same line of the same file.

1. Start the merge and let it stop on a conflict:

```
git switch main
git merge feature
```

2. See which files are conflicted:

```
git status
```

3. Open the file and find the conflict markers. Edit the lines so the file reads correctly, then delete the `<<<<<<<`, `=======`, and `>>>>>>>` lines.

4. Stage the resolved file:

```
git add <file>
```

5. Complete the merge:

```
git commit          # or: git merge --continue
```

> ✅ Expected result: `git status` is clean and the merge commit is recorded. To bail out mid-merge instead, run `git merge --abort`.

📋 Conflict-resolution checklist

- [ ] I ran `git status` to list every conflicted file before editing.
- [ ] I opened each file and edited the lines to the version I actually want.
- [ ] I removed all three marker lines: `<<<<<<<`, `=======`, and `>>>>>>>`.
- [ ] I staged each resolved file with `git add <file>`.
- [ ] I finished with `git commit` (or `git merge --continue`).
- [ ] If it felt wrong, I remembered `git merge --abort` resets me to before the merge.

📌 Key takeaways

- A branch is a movable pointer to a commit, so branching and switching are instant.
- `git switch -c feature` creates and switches; `git switch main` moves back; `git branch` lists.
- A merge is a fast-forward (pointer moves) when nothing diverged, or a three-way merge (a merge commit with two parents) when both lines advanced; `--no-ff` forces a merge commit.
- Conflicts are marked, not lost: edit the file, remove the markers, `git add`, then `git commit`; `git merge --abort` backs out.

🔗 Go deeper

- Pro Git – Branches in a Nutshell – the source for the "branch is a pointer" model in Figure 5.1.
- Pro Git – Basic Branching and Merging – fast-forward, three-way merges, and conflict handling.
- git merge documentation – `--no-ff`, `--continue`, and `--abort` in full.
- git switch documentation – the modern create/switch command and its `git checkout` equivalents.

6. Working with Remotes

> ⚡ **The gist**
>
> - A remote is just a named reference to another copy of your repo. The default name is `origin`.
> - `git fetch` downloads new commits but changes nothing in your branch. `git pull` is fetch plus merge.
> - Remote-tracking branches like `origin/main` are your local read-only snapshot of the remote, refreshed by fetch.
>
> *Reading time: ~8 min*

What a remote is

Git is distributed: every clone is a full repository. A **remote** is simply a named reference to another copy of that repository, usually one hosted on a server the whole team can reach. You talk to it by name instead of retyping a long URL, and the conventional name for the first remote is `origin`.

When you `git clone` a repository, Git sets up `origin` for you automatically. If you started with `git init` instead, you add the remote yourself. Either way, you can see what is wired up with `git remote -v`.

```
git remote -v                # list remotes (fetch + push URLs)
git remote add origin <url>    # name a remote "origin"
```

Local and remote, side by side

Your work lives in three local areas: the working directory, the index, and your committed history (HEAD). The remote is a separate copy. Two commands move commits between them: `git fetch` pulls the remote's state down into a read-only **remote-tracking branch** called `origin/main`, and `git push` sends your commits up.

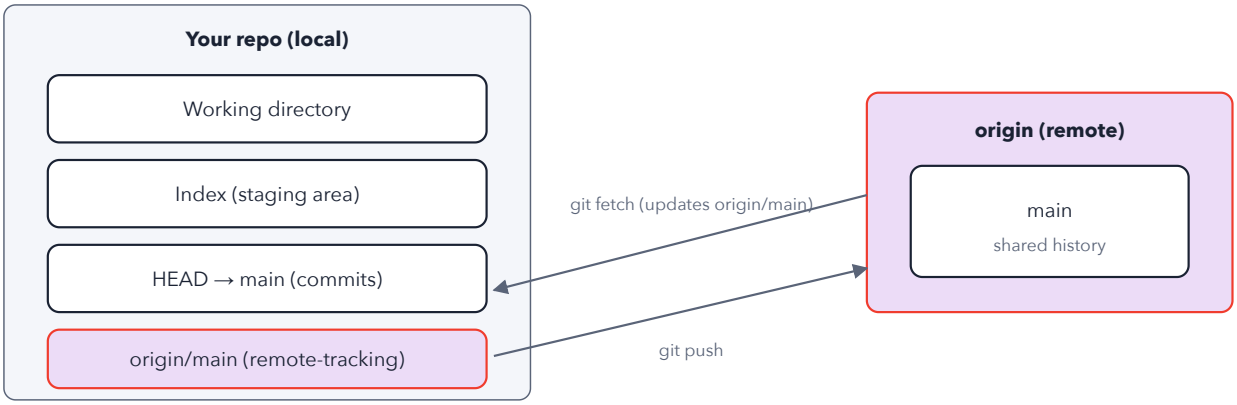


Figure 6.1 — Local vs remote: fetch brings the remote's commits down into `origin/main`; push sends your commits up.

Fetch vs pull

This is the distinction that saves the most confusion. `git fetch origin` downloads new commits from the remote and updates your `origin/main` pointer, but it does **not** touch your working branch. Nothing you can see in your files changes. It is always safe to run.

`git pull` is a convenience: it does the fetch and then immediately merges the fetched commits into your current branch. If you prefer a linear history instead of a merge commit, use `git pull --rebase`, which replays your local commits on top of what you fetched.

```
git fetch origin            # download, don't merge
git pull                    # fetch + merge (into current branch)
git pull --rebase           # fetch + replay your commits on top
```

Push and upstream

The first time you send a branch to the remote, use `git push -u origin main`. The `-u` sets the **upstream**, meaning it records that your local `main` tracks `origin/main`. After that, a bare `git push` (and `git pull`) knows where to go.

```
git push -u origin main       # push + set upstream
git push                      # afterwards, this is enough
git branch -vv                # show tracking relationships
```

Run `git branch -vv` to see which remote-tracking branch each local branch follows, and whether you are ahead or behind.

Recipes

📝 Recipe 6.1 — Push a repo to a new remote

Prerequisites: a local repo with at least one commit on `main`, and an empty remote repository whose URL you have copied.

1. Point Git at the remote and name it `origin`:

   ```
   git remote add origin <url>
   ```

2. Confirm it is wired up:

   ```
   git remote -v
   ```

3. Push your branch and set the upstream in one step:

   ```
   git push -u origin main
   ```

4. From now on, a bare push is enough:

   ```
   git push
   ```

   ✅ Expected result: the remote now shows your `main` branch with your commits, and `git branch -vv` lists `main` tracking `origin/main`.

📝 Recipe 6.2 – Sync with a shared branch

Prerequisites: a cloned repo (so `origin` is already set), and teammates who have pushed new commits.

1. Download the latest commits without changing your branch:

```
git fetch origin
```

2. Review what arrived before you integrate it:

```
git log origin/main
```

3. Merge it into your branch (or replay your work on top):

```
git pull
# or, for linear history:
git pull --rebase
```

4. Confirm you are current:

```
git branch -vv
```

✅ Expected result: `git branch -vv` shows your local `main` in sync with `origin/main` (not "behind"), and `git status` reports your branch is up to date.

📋 Before you push

- [ ] I ran `git status` and know exactly what is committed.
- [ ] I ran `git fetch origin` and reviewed `git log origin/main` for new work.
- [ ] I integrated the remote's changes with `git pull` (or `git pull --rebase`) so I am not behind.
- [ ] My branch has an upstream set (`git branch -vv` shows it tracking `origin/main`).
- [ ] I am pushing to the branch I intend to (`origin main`, not someone else's branch).

💼 Business angle

A remote is the single source of truth a team shares. Fetch-review-then-integrate turns "merge surprises" into predictable, auditable updates, so collaboration stays fast and rollbacks stay cheap.

📌 Key takeaways

- A remote is a named reference to another copy of the repo; `origin` is the default, and `git clone` sets it up for you.
- `git fetch` downloads without merging; `git pull` fetches and merges (use `--rebase` for linear history).
- `git push -u origin main` pushes and sets the upstream so later a bare `git push` works.
- Remote-tracking branches like `origin/main` are read-only local copies of the remote's state; `git branch -vv` shows the tracking relationship.

🔗 Go deeper

- Pro Git – Working with Remotes – the source for fetch vs pull and upstream tracking.
- git-remote docs – managing named remotes.
- git-push docs – the `-u` / `--set-upstream` flag in full.

7. Rewriting History: Amend, Rebase, Interactive Rebase

> ⚡ **The gist**
>
> - Git lets you tidy up local commits before you share them: fix the last one with `--amend`, and reshape a whole series with `rebase`.
> - Merge keeps the true, branching story and adds a merge commit. Rebase replays your commits onto a new base, giving a clean straight line but new commit IDs.
> - One rule keeps you safe: rewrite only history that lives on your machine and nowhere else.
>
> *Reading time: ~9 min*

Why rewrite history at all?

Real work is messy. You commit a typo, forget a file, or leave a trail of "wip", "fix", "actually fix" commits behind you. History rewriting lets you present a clean, readable story to your teammates instead of your rough drafts.

Git gives you two tools for this. `git commit --amend` replaces just the last commit. `git rebase` replays a whole run of commits onto a new starting point, and its interactive mode lets you reorder, combine, rename, edit, or drop them one by one.

Amend: fix the last commit

Amending swaps out your most recent commit for a new one. You can change the message, the content, or both. It is the fastest way to fix a commit you just made and have not shared yet.

```
git commit --amend          # replace the last commit (message and/or content)
```

Note that amend does not "edit" the old commit. It builds a brand new commit with a new ID and moves your branch to it. The old one is discarded. That is exactly why amending shared commits is dangerous, which we will get to.

Rebase: replay onto a new base

A rebase takes the commits on your current branch and re-applies them, one at a time, on top of a different commit. The classic use is catching a feature branch up with `main` without a merge commit.

```
git rebase main             # replay the current branch's commits onto main
```

Because each commit is re-applied on a new base, Git creates fresh commits with new IDs. The changes are the same; the identities are not.

Merge vs rebase

Both integrate work from one line into another, but they tell different stories. Merge preserves the true history: it keeps both lines of development and records a merge commit where they came back together. Rebase produces a linear history, as if you had started your work from the latest base all along, but it rewrites your commit IDs to do so.

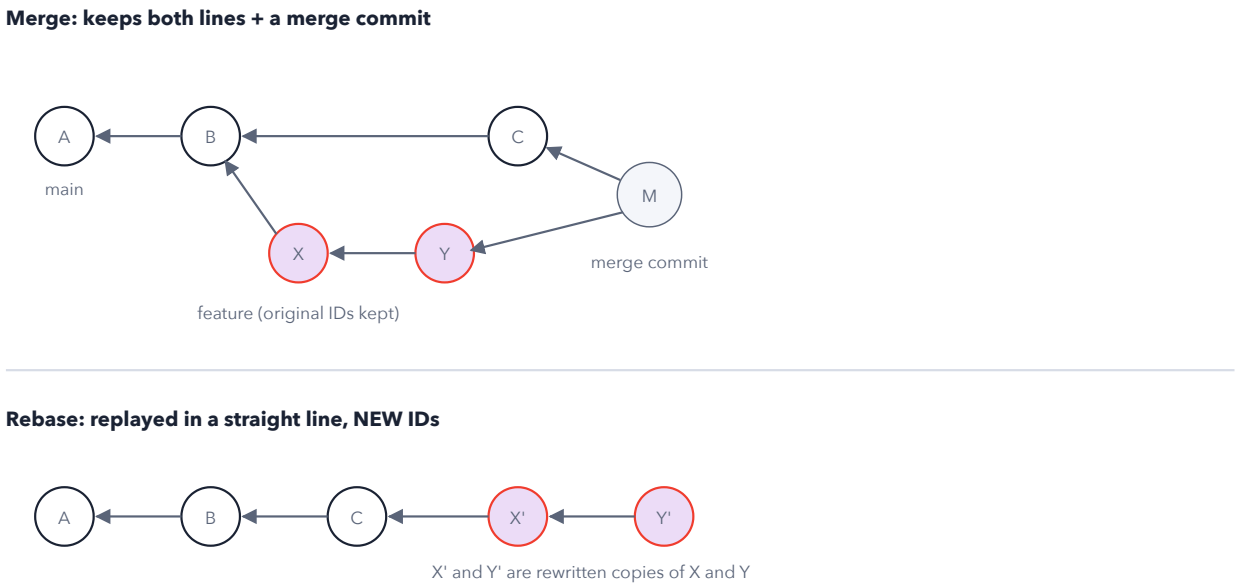


Figure 7.1 — Merge vs rebase: the same divergence resolved two ways. Merge keeps both lines and adds a merge commit; rebase replays the commits linearly with new IDs.

> 💼 **Business angle**
>
> A readable history is a team asset. Clean, linear commits make code review faster, make `git bisect` and blame trustworthy, and make rollbacks obvious. Time saved on "what does this commit even do?" compounds across every reviewer, on every pull request, for the life of the repository.

📝 Recipe 7.1 — Fix the last commit with amend

Prerequisites: a repository with at least one commit that you have *not* pushed or shared.

1. Make the fix, for example correct a file you forgot to stage, and stage it:

```
git add forgotten-file.txt
```

2. Fold the staged change into the previous commit and edit its message:

```
git commit --amend
```

3. Confirm there is still only one commit, now corrected:

```
git log
```

✅ Expected result: `git log` shows the corrected commit with your updated message and content, and no extra commit was added.

Interactive rebase: reshape a series

Interactive rebase is where the cleanup really happens. You give Git a starting point, and it opens an editor listing the commits since then, oldest at the top. You mark what to do with each one, save, and Git carries out your plan.

```
git rebase -i HEAD~5          # interactive rebase over the last 5 commits
```

In the list you can **reorder** lines to change commit order, **squash** to fold a commit into the one above it, **reword** to change a message, **edit** to stop and amend a commit, or **drop** to remove a commit entirely.

If a replay hits a conflict, Git pauses so you can sort it out. You then drive the rebase with:

```
git rebase --continue         # resume after resolving
git rebase --skip             # skip the current commit
git rebase --abort            # bail out, back to where you started
```

Reach for `--abort` any time you feel lost. It puts the branch back exactly as it was before the rebase began.

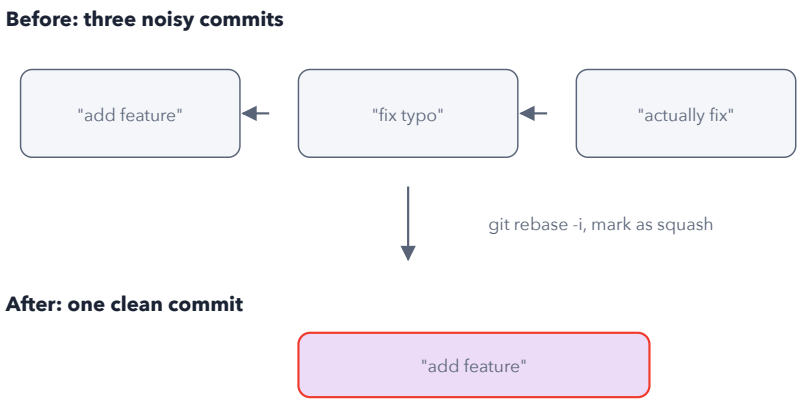


Figure 7.2 — Interactive rebase: three small work-in-progress commits squashed into one clean commit.

📝 Recipe 7.2 — Squash a messy branch with interactive rebase

Prerequisites: a local branch with three commits you have not pushed, for example "add feature", "fix typo", and "actually fix".

1. Start an interactive rebase over the last three commits:

```
git rebase -i HEAD~3
```

2. An editor opens listing the three commits, oldest first, each prefixed with `pick`.
3. Leave the first commit as `pick`. Change the word `pick` to `squash` on the second and third lines so they fold into the first:

```
pick   1a2b3c  add feature
squash 4d5e6f  fix typo
squash 7g8h9i  actually fix
```

4. Save and close the file. Git combines the three into one and opens a second editor so you can write the final commit message. Save that too.
5. Confirm the branch now holds a single commit:

```
git log --oneline
```

✅ Expected result: `git log --oneline` shows one clean commit in place of the three, with your chosen message.

The one rule that keeps you safe

Amend and rebase both change commit IDs. That is harmless while the commits live only on your machine. It becomes a problem the moment those commits exist somewhere else, because your rewritten history no longer matches the copy your teammates already pulled.

> ⚠️ **Watch out**
>
> **The golden rule of rewriting history:** never rebase or amend commits that exist outside your own repository and that others may have based work on. Rewriting changes hashes, so once history is shared, a rewrite makes your version and everyone else's diverge, forcing painful re-syncs and lost work. Rewrite **local history only**. Once commits are pushed and shared, prefer `git revert` to undo them safely.

Safe to rewrite?

Before you amend or rebase, run down this quick check. If every box is ticked, rewrite away.

- [ ] These commits live only in my local repository and have not been pushed.
- [ ] No teammate has pulled, branched from, or based work on these commits.
- [ ] I am rewriting to clean up, not on a shared branch like `main`.
- [ ] If it is already shared, I will use `git revert` instead of amend or rebase.
- [ ] I know I can back out mid-rebase with `git rebase --abort`.

> 📌 **Key takeaways**
>
> - `git commit --amend` replaces the last commit; `git rebase main` replays your branch onto a new base.
> - Interactive rebase (`git rebase -i HEAD~5`) lets you reorder, squash, reword, edit, or drop commits; steer it with `--continue` , `--skip` , and `--abort` .
> - Merge preserves the true, branching history with a merge commit; rebase gives a clean linear history but rewrites commit IDs.
> - The golden rule: rewrite local history only. Never rebase or amend commits others may have based work on; use `git revert` on shared history.

> 🔗 **Go deeper**
>
> - Pro Git 3.6 – Rebasing – merge vs rebase and the golden rule, the source for this chapter.
> - git-rebase documentation – full reference for interactive rebase and its options.
> - git-commit documentation – details on `--amend` .

8. Undo Anything: Restore, Reset, Revert, Reflog, Stash

> ⚡ **The gist**
>
> - Different messes need different tools: `restore` for working-dir changes, `reset` to move a branch pointer, `revert` to undo a shared commit safely.
> - The three resets touch different trees: `--soft` keeps changes staged, `--mixed` keeps files, `--hard` throws work away.
> - `git reflog` is your safety net: even after a bad reset, the old commit is still there for about 90 days.
> - `git stash` shelves dirty work; `git cherry-pick` copies one commit onto your branch.
>
> *Reading time: ~9 min*

Which undo do I need?

Almost every "help, I broke it" moment maps to one command. Match your situation to the row, then read the section it points to.

- [ ] I edited a file and want the last saved version back: `git restore file` (older: `git checkout -- file`).
- [ ] I staged a file by mistake but want to keep the edits: `git restore --staged file`.
- [ ] I want to undo my last commit but keep the changes ready to re-commit: `git reset --soft HEAD~1`.
- [ ] I want to undo my last commit and unstage, keeping the files as-is: `git reset --mixed HEAD~1` (the default).
- [ ] I want to undo my last commit and throw the changes away: `git reset --hard HEAD~1` (dangerous).
- [ ] I need to undo a commit that is already pushed or shared: `git revert <commit>`.
- [ ] I ran a bad reset and lost a commit: `git reflog`, then recover it.
- [ ] I have dirty changes and need a clean tree right now: `git stash`, then `git stash pop` later.
- [ ] I want one specific commit from another branch: `git cherry-pick <commit>`.

Discard working-dir changes, and unstage

The smallest undo is throwing away edits you have not committed. `git restore file` replaces the file in your working directory with the version from the last commit. The older spelling is `git checkout -- file`; both do the same thing.

If instead you staged something too early, `git restore --staged file` unstages it. This moves the file out of the index but leaves your working-directory edits untouched, so nothing you typed is lost.

> 💼 **Business angle**
>
> A team that trusts its undo tools experiments more. When rolling back a change costs seconds instead of an afternoon, developers try the bolder fix, and safe experimentation compounds into faster delivery.

The three resets, and the trees they touch

Recall the three trees from Chapter 3: the working directory, the staging area (index), and the repository (where `HEAD` and your branch pointer live). Every `git reset` moves the branch pointer; the flag decides how far it also rewinds the index and the working directory.

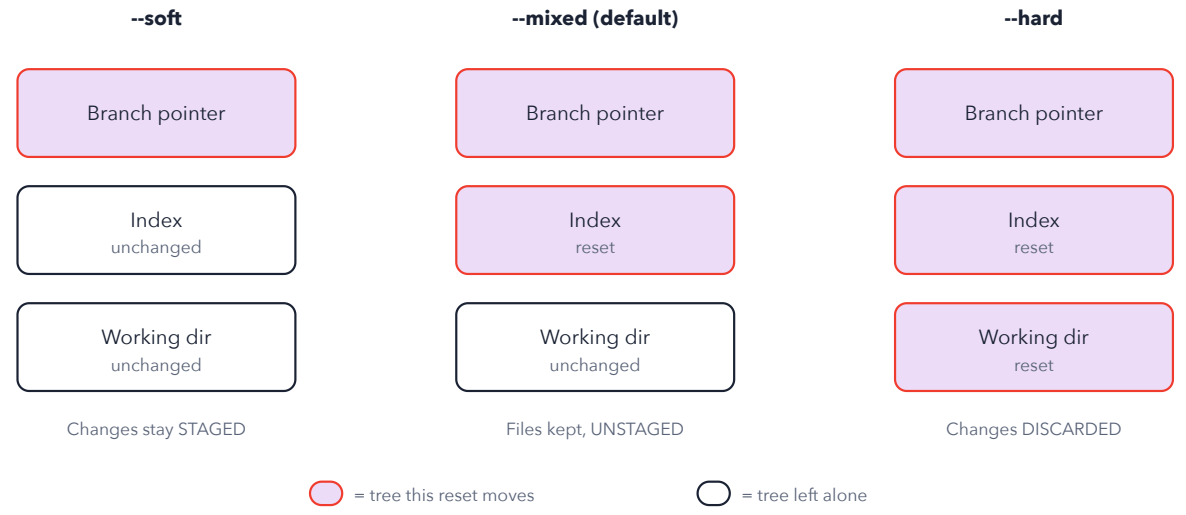


Figure 8.1 — `reset --soft` moves only the branch pointer; `--mixed` also resets the index; `--hard` also resets the working directory.

Read the figure top to bottom: `--soft` moves just the pointer, so your changes stay staged and ready to re-commit. `--mixed` (the default) also resets the index, so files are kept but unstaged. `--hard` resets all three trees, so uncommitted work is gone.

> ⚠️ **Watch out**
>
> `git reset --hard` deletes uncommitted changes in your working directory with no confirmation and no trash can. Commit or `git stash` first if there is any chance you want that work back. The commit you rewound past is recoverable via reflog; unsaved edits are not.

📝 Recipe 8.1 – Undo the last commit three ways

Prerequisites: a local branch with at least one commit you have not pushed, so it is safe to rewrite.

1. Undo the commit but keep the changes staged (ready to re-commit):

```
git reset --soft HEAD~1
git status                # changes shown as "to be committed"
```

   ✅ Expected result: the commit is gone from `git log`; your files are staged, ready to commit again.

2. Undo the commit and unstage, but keep the files (the default):

```
git reset --mixed HEAD~1
git status                # changes shown as "not staged"
```

   ✅ Expected result: the commit is gone; your edits are still on disk but no longer staged.

3. Undo the commit and discard the changes entirely (dangerous):

```
git reset --hard HEAD~1
git status                # "nothing to commit, working tree clean"
```

   ✅ Expected result: the commit and its changes are gone; the working tree is clean. Only use this when you are sure.

4. Confirm which commit you are now on:

```
git log --oneline -3
```

   ✅ Expected result: the top commit is the one before your undone commit.

Revert: the safe undo for shared history

Reset rewrites history by moving a branch pointer backward. That is fine for local commits, but on a branch others have pulled it breaks their history (the same golden rule from Chapter 7). For anything already pushed or shared, use `git revert <commit>` instead.

Revert does not delete anything. It creates a **new** commit that applies the inverse of the target commit, so the change is undone while the history everyone shares stays intact and moves forward.

Situation	Reach for	Why
Commit is local, not pushed	`git reset`	Rewriting your own history is safe
Commit is pushed or shared	`git revert <commit>`	Adds a new undo commit; nobody's history breaks

Reflog: you lose the pointer, not the commit

When a reset seems to destroy a commit, the commit object usually still exists. Git just moved the pointer off it. `git reflog` is a log of every position `HEAD` has held: each commit, checkout, reset, and merge. The "lost" commit's SHA is right there in that list.

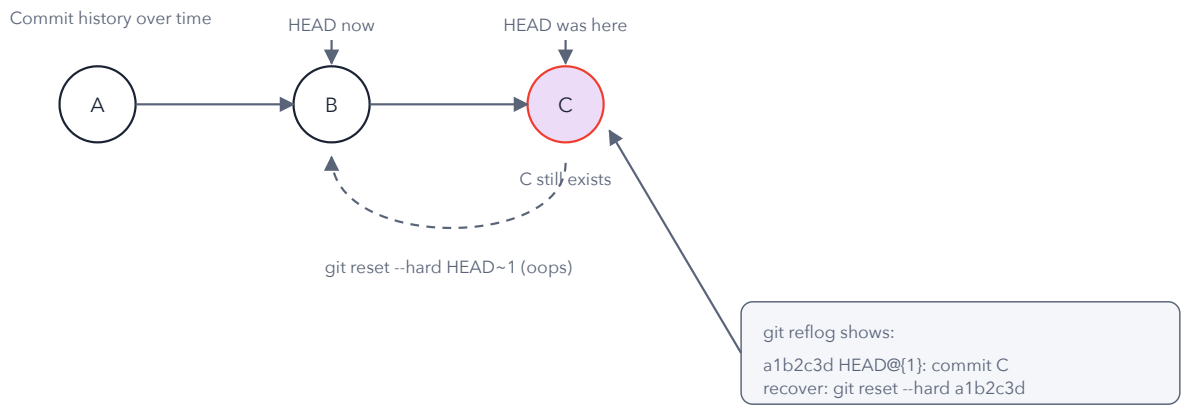


Figure 8.2 – A bad `reset --hard` moves HEAD back to B, but commit C is still reachable through its SHA in the reflog.

To get C back, find its SHA in the reflog and point a branch at it again, either with `git reset --hard <sha>` to move your current branch, or `git switch -c rescue <sha>` to create a fresh branch on it.

> 💡 **Tip**
>
> The reflog keeps entries for about 90 days by default, so a commit you "lost" this morning is almost certainly still recoverable. In Git you rarely lose a commit; you lose the pointer to it, and reflog hands the pointer back.

📝 Recipe 8.2 – Recover after a bad hard reset with reflog

Prerequisites: you just ran `git reset --hard HEAD~1` and realize you needed that commit back.

1. List where HEAD has been:

```
git reflog
```

✅ Expected result: a list like `a1b2c3d HEAD@{1}: commit: ...` ; your lost commit is one of the recent entries.

2. Identify the SHA of the commit you want back (the one just before the reset).
3. Recover it, either by moving your current branch to it:

```
git reset --hard a1b2c3d
```

or by creating a new branch that starts at it:

```
git switch -c rescue a1b2c3d
```

✅ Expected result: your branch (or the new `rescue` branch) now points at the recovered commit.

4. Verify the commit is back:

```
git log --oneline -3
```

✅ Expected result: the commit you thought you lost is at the top of the log again.

Stash and cherry-pick

Sometimes you need a clean working tree right now (to switch branches or pull) but you are mid-edit. `git stash` shelves your uncommitted changes and gives you a clean tree; `git stash pop` brings them back when you are ready.

The opposite need is picking up a single commit from elsewhere. `git cherry-pick <commit>` applies just that one commit onto your current branch, without merging the whole branch it came from.

- [ ] I can discard a file's edits with `git restore` and unstage with `git restore --staged`.
- [ ] I can name what each of `--soft`, `--mixed`, and `--hard` keeps and discards.
- [ ] I reach for `revert` (not `reset`) on anything already pushed.
- [ ] I know how to find a lost commit in `git reflog` and recover it.
- [ ] I can shelve work with `git stash` and copy one commit with `git cherry-pick`.

📌 Key takeaways

- `git restore file` discards working-dir edits; `git restore --staged file` unstages while keeping the edits.
- The three resets differ by how many trees they touch: `--soft` keeps changes staged, `--mixed` keeps files unstaged, `--hard` discards them.
- `git revert` adds a new commit that undoes another; it is the safe choice for shared or pushed history, where `reset` would break others.
- `git reflog` keeps a trail of where HEAD has been for about 90 days, so a "lost" commit is usually one `reset --hard <sha>` or `switch -c rescue <sha>` away.
- `git stash` shelves dirty changes; `git cherry-pick` copies a single commit onto your branch.

🔗 Go deeper

- Pro Git – Reset Demystified – the three trees and what each reset touches (source for Figure 8.1).
- git-reflog documentation – how the reflog records HEAD movement (source for Figure 8.2).
- git-revert documentation – undoing commits safely on shared branches.
- git-restore documentation – discarding and unstaging working-dir changes.
- git-stash and git-cherry-pick documentation – shelving work and copying single commits.

9. Collaboration Workflows

The gist

- The everyday team pattern: branch off `main`, commit, open a pull request for review, merge back.
- Trunk-based vs Git Flow: short-lived branches for continuous delivery, longer-lived branches for scheduled releases.
- A clean `.gitignore` and a `pre-commit` hook keep junk and broken code out of the repo automatically.

Reading time: ~9 min

Feature branches and pull requests

A team workflow is mostly a set of shared habits around one cheap operation: creating a branch. Because a branch is just a movable pointer, you can give every unit of work its own line of history, then bring it back through review.

The pattern is small enough to memorize. You start a branch off `main`, do your work as a series of commits, then open a **pull request** (PR): a request that your changes be reviewed and merged. Teammates read the diff, comment, and approve. Once it passes review, the branch merges back into `main`.

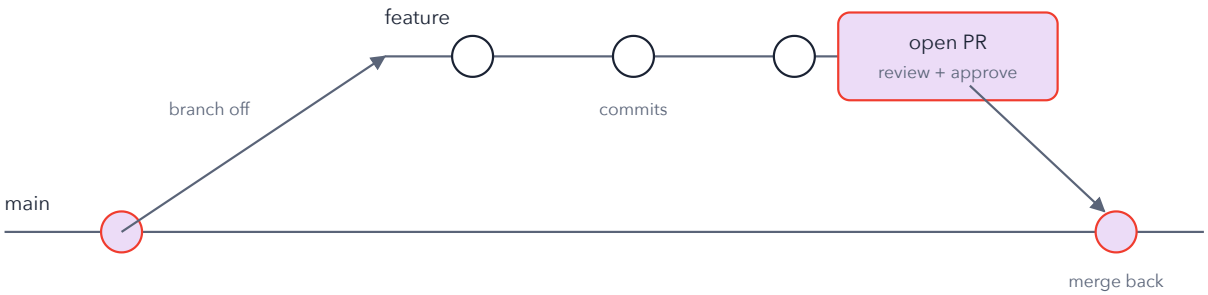


Figure 9.1 — Feature-branch and pull-request flow: branch off `main`, commit, open a PR for review, then merge back.

The value is that `main` stays releasable. Work in progress lives on its own branch, review happens on a self-contained diff, and nothing lands on the shared line until it has been read by another human.

Trunk-based vs Git Flow

Teams differ on how long branches live and how many long-running branches exist. Two named patterns anchor the spectrum.

Trunk-based development keeps branches short-lived and merges them into `main` (the "trunk") frequently, often several times a day. Small, frequent merges reduce divergence and pair naturally with continuous integration and continuous delivery (CI/CD). **Git Flow** uses several long-lived branches, typically a `develop` integration branch plus dedicated `release` and `hotfix` branches. It is heavier to run but gives a clear structure for scheduled, versioned releases.

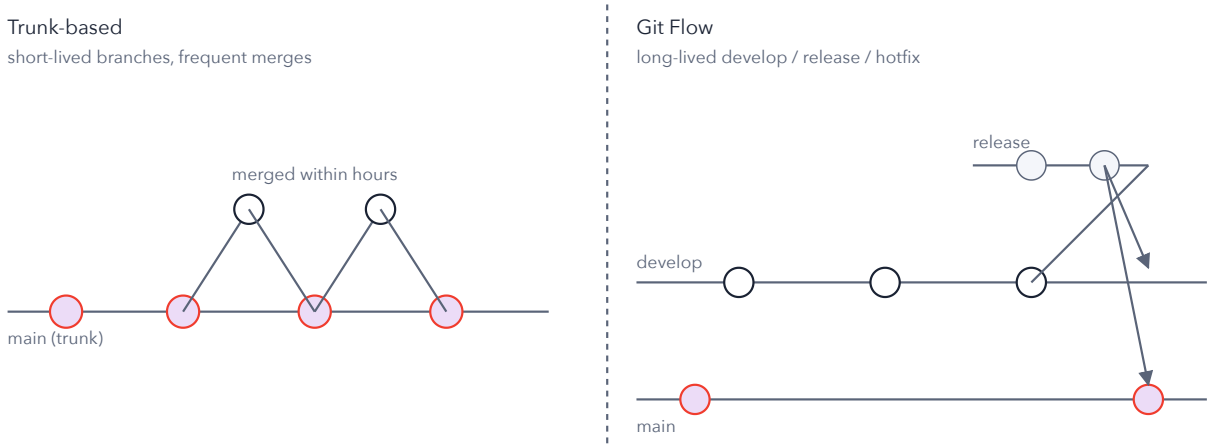


Figure 9.2 – Trunk-based (short-lived branches merged often, ideal for CI/CD) vs Git Flow (long-lived branches, ideal for scheduled releases).

Neither is "correct." Pick trunk-based when you deploy continuously and want minimal ceremony; pick Git Flow when you ship on a fixed schedule and need release and hotfix branches to coordinate versions.

> 💼 **Business angle**
>
> Short-lived branches shorten the feedback loop: smaller diffs review faster, merge cleaner, and reach customers sooner. That review-cycle speed is where most teams recover the cost of adopting a workflow at all.

.gitignore done right

A `.gitignore` file lists patterns for untracked files Git should not track (the pattern format is documented in `gitignore(5)`). The point is to keep the repository to source only: ignore build artifacts, downloaded dependencies, secrets, and OS clutter.

Ignore rules come from three places, applied in this precedence order:

1. `.git/info/exclude` – patterns local to your clone, never shared.

2. Per-directory `.gitignore` files – committed and shared with the team.
3. Global `core.excludesFile` – your personal, machine-wide ignores.

Good defaults to ignore: build output (`dist/` , `build/` , `*.o`), dependency folders (`node_modules/` , `vendor/`), secrets and env files (`.env` , `*.key`), and OS files (`.DS_Store` , `Thumbs.db`). Commit source and config; ignore anything reproducible or private.

Commit-message conventions

A commit message has two jobs: name the change and explain the reasoning. The common convention is an **imperative subject line** (write "Add retry to upload", not "Added" or "Adds"), kept short, followed by a blank line and a body that explains *why* the change was made, not just what it does.

```
Add retry logic to file upload

Uploads failed on flaky networks with no recovery. Retry
three times with backoff so a transient error no longer
loses the user's work.
```

The subject is what teammates scan in `git log --oneline` ; the body is what a future maintainer reads when they run `git blame` and ask "why is this here?"

Hooks: automate the rules

Git **hooks** are scripts in `.git/hooks/` that run automatically on events. The common ones are `pre-commit` (runs before a commit is recorded, e.g. to lint or run fast tests), `commit-msg` (validates the message, e.g. enforce a format), and `pre-push` (runs before you push, e.g. a fuller test suite). A hook that exits non-zero aborts the operation, so a failing check blocks the commit.

Hooks in `.git/hooks/` are **not versioned by default**, so they do not travel with a clone. To share hooks across a team, teams use tools such as pre-commit or Husky, which store hook config in the repo and install the hooks for everyone.

Recipe: a .gitignore and a pre-commit hook

📝 Recipe 9.1 — Set up a .gitignore and a pre-commit hook

Goal: keep junk out of the repo, and block any commit that contains a forbidden marker.

1. Create a `.gitignore` at the repo root and commit it:

```
printf "node_modules/\ndist/\n.env\n.DS_Store\n" > .gitignore
git add .gitignore
git commit -m "Add .gitignore for deps, build output, and secrets"
```

2. Create the hook file `.git/hooks/pre-commit` that rejects any staged change containing the word `DONOTCOMMIT`:

```
cat > .git/hooks/pre-commit <<'EOF'
#!/bin/sh
if git diff --cached | grep -q "DONOTCOMMIT"; then
  echo "pre-commit: found DONOTCOMMIT marker. Commit blocked."
  exit 1
fi
exit 0
EOF
```

3. Make the hook executable:

```
chmod +x .git/hooks/pre-commit
```

4. Try to commit a file that contains the marker:

```
echo "x = 1  # DONOTCOMMIT" > work.txt
git add work.txt
git commit -m "Test the hook"
```

✅ Expected result: the commit is refused. Git prints `pre-commit: found DONOTCOMMIT marker. Commit blocked.` and `git log` shows no new commit. Remove the marker line and commit again to confirm it now succeeds.

📋 Pull-request readiness checklist

- [ ] My branch is up to date with the latest `main`.
- [ ] The change is scoped to one thing, small enough to review in one sitting.
- [ ] Each commit has an imperative subject and a body explaining why.
- [ ] No secrets, build artifacts, or debug markers are in the diff (`.gitignore` and the hook did their job).
- [ ] Tests and linters pass locally.
- [ ] The PR description says what changed and how to verify it.

📌 Key takeaways

- The team loop is: branch off `main`, commit, open a PR for review, merge back. It keeps `main` releasable.
- Trunk-based means short-lived branches merged often (great for CI/CD); Git Flow means long-lived develop/release/hotfix branches (great for scheduled releases).
- Use `.gitignore` to keep only source in the repo; remember the precedence: `.git/info/exclude`, per-dir `.gitignore`, global `core.excludesFile`.
- Write imperative subjects and explain why in the body; automate checks with hooks (`pre-commit`, `commit-msg`, `pre-push`), sharing them via tools like pre-commit or Husky.

🔗 Go deeper

- gitworkflows(7) – Git's own guide to recommended workflows.
- gitignore(5) – the ignore pattern format and precedence rules.
- githooks documentation – every hook, when it runs, and how exit codes control the operation.
- pre-commit – a framework for sharing hooks across a team.
- Atlassian Git tutorials – walk-throughs of feature-branch, trunk-based, and Git Flow workflows.

10. Power Tools and Scale

⚡ The gist

- Turn history into a debugger: `blame`, the `log -S` pickaxe, and `--follow` across renames.
- `git bisect` binary-searches your commits to name the exact one that broke something.
- Ship clean releases with annotated tags, and keep giant repos fast with submodules, LFS, and partial clone.

Reading time: ~10 min

History is a debugger

Once your project has a real history, that history becomes a search engine for "who, what, and when." Three commands do most of the work.

`git blame` annotates every line of a file with the commit, author, and date that last touched it. It answers "who wrote this line, and why."

`git log -S` is the "pickaxe": it finds the commits that added or removed a given string, so you can trace when a function, config value, or magic constant first appeared or vanished. **`git log --follow`** keeps a file's history intact even after it was renamed or moved.

```
git blame app.py            # who last changed each line
git log -S"MAX_RETRIES"     # commits that added/removed the string
git log --follow app.py     # history across renames
```

Bisect: find the commit that broke it

When something worked "a while ago" but is broken now, you do not have to read every commit. `git bisect` does a binary search: you mark one *bad* commit and one older *good* commit, and Git checks out the midpoint for you to test. You say `good` or `bad`, it halves the range again, and it repeats until only one commit is left standing.

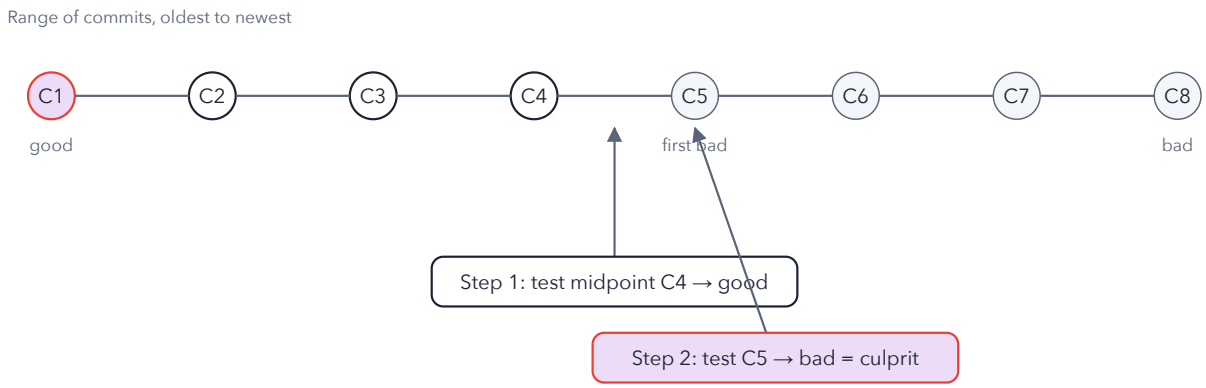


Figure 10.1 — git bisect halves the range each round: mark the ends good and bad, test the midpoint, and repeat until one first-bad commit remains.

📝 Recipe 10.1 — Find the commit that broke it with bisect

Prerequisites: a clean working tree, a bug you can test for now, and a commit (or tag) where it definitely worked.

1. Start a bisect session:

```
git bisect start
```

2. Mark the current commit as broken:

```
git bisect bad
```

3. Mark a known-good older commit (a SHA or tag):

```
git bisect good v1.0.0
```

4. Git checks out the midpoint. Test it, then tell Git the result. Repeat until done:

```
git bisect good      # if this commit works
git bisect bad       # if this commit is broken
```

> ✅ Expected result: Git prints `<sha> is the first bad commit`, with the author and message of the culprit.

5. End the session and return to where you started:

```
git bisect reset
```

💼 Business angle

Bisect turns "somewhere in the last 300 commits" into about 8 tests: binary search finds a bad commit among a thousand in roughly ten steps, so a regression hunt that used to eat a day becomes a coffee break.

Tags and releases

A tag is a permanent name for a specific commit, ideal for marking releases. There are two kinds. A **lightweight tag** is just a name pointing at a commit. An **annotated tag** also stores a message, author, and date as its own object, which is why it is the preferred choice for real releases.

```
git tag v1.0.0                    # lightweight
git tag -a v1.0.0 -m "release 1.0" # annotated (preferred)
```

One surprise catches everyone: **tags are not pushed by default.** A normal `git push` sends branches, not tags. You push a tag explicitly, or push them all at once.

```
git push origin v1.0.0   # push one tag
git push origin --tags   # push all tags
```

Repos inside repos, and files too big for Git

As projects grow, two problems appear: you want to include another repository, and you have large binaries that would bloat history forever. Git has a tool for each.

A **submodule** is a repository nested inside your repository, pinned to a specific commit of the other project. Your repo records *which commit* of the dependency it uses, so checkouts are reproducible. When cloning, pull the submodules along too.

Git LFS (Large File Storage) keeps large binaries out of the main history. You track a file pattern, and Git stores a small pointer in history while the heavy bytes live out-of-band, so clones stay lean. For very large monorepos, **sparse-checkout** and **partial clone** (`--filter=blob:none`) let you fetch only the files and objects you actually need.

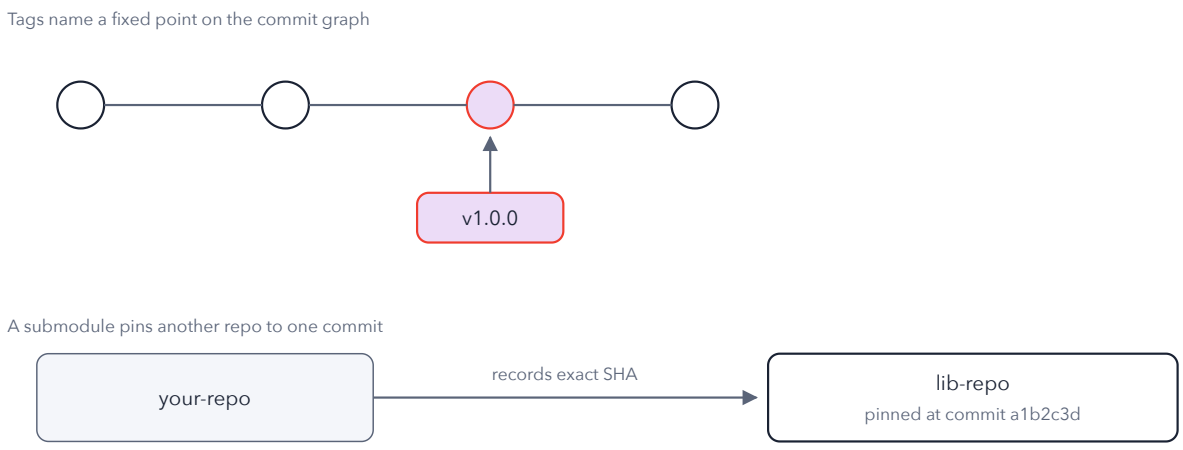


Figure 10.2 — A tag is a named pointer to one commit (a release marker); a submodule is another repo embedded in yours, pinned to a specific commit for reproducible checkouts.

```
git submodule add https://host/lib.git vendor/lib
git clone --recurse-submodules https://host/app.git
git lfs track "*.psd"
git clone --filter=blob:none https://host/monorepo.git
```

📝 Recipe 10.2 – Tag and push a release

Prerequisites: the commit you want to release is committed and pushed, and a remote named `origin` exists.

1. Create an annotated tag on the current commit:

```
git tag -a v1.0.0 -m "release 1.0"
```

2. Push the tag to the remote (a plain push does not send it):

```
git push origin v1.0.0
```

3. Confirm the tag reached the remote:

```
git ls-remote --tags origin
```

✅ Expected result: `refs/tags/v1.0.0` is listed on the remote, so the release is tagged for everyone.

📋 Big-repo hygiene checklist

- [ ] I use annotated tags (`-a`) for releases, not lightweight ones.
- [ ] I remember tags are not pushed by default, so I push them with `git push origin <tag>` or `--tags` .
- [ ] Large binaries (art, datasets, media) go through Git LFS, not straight into history.
- [ ] I clone submodule-using projects with `--recurse-submodules` .
- [ ] For a giant monorepo I reach for partial clone (`--filter=blob:none`) or sparse-checkout.
- [ ] I debug with history first: `blame` , `log -S` , and `bisect` before guessing.

📌 Key takeaways

- `git blame`, `git log -S`, and `git log --follow` turn history into a debugger for who, what, and when.
- `git bisect` binary-searches commits to name the exact one that introduced a bug, in about ten tests for a thousand commits.
- Use annotated tags for releases and push them explicitly; tags do not travel with a normal push.
- Submodules pin a nested repo to a commit; Git LFS, sparse-checkout, and partial clone keep big repos fast.

🔗 Go deeper

- git bisect – the source for Recipe 10.1 and Figure 10.1.
- git submodule – nesting a repo pinned to a commit (Figure 10.2).
- Git LFS – keeping large binaries out of the main history.
- Pro Git, chapter 7 (Git Tools) – bisect, submodules, and more power tools.
- Git reference manual – full pages for blame, log, and tag.

11. Cheat Sheet and Learning Path

⚡ The gist

- Every command you reach for daily, on one page, grouped by job.
- A "which undo do I need?" lookup so panic never costs you work.
- A 30/60/90-day path from first commit to teaching others.

Reading time: ~7 min

The one-page cheat sheet

Keep this open in a second window for your first month. Everything here appears earlier in the book with full context; this is the fast lookup.

Figure 11.1 — One-page command cheat card: the commands you use most, grouped by the job they do.

Everyday

Command	What it does
`git status`	Show what changed and what is staged.
`git add file.txt`	Stage a file (add it to the index).
`git commit -m "message"`	Record the staged snapshot.
`git log --oneline --graph --all --decorate`	Visual history of commits and branches.
`git diff`	Working directory vs the index.
`git diff --staged`	Index vs the last commit.

Branch and merge

Command	What it does
`git switch -c feature`	Create a branch and switch to it.
`git switch main`	Switch to an existing branch.
`git merge feature`	Merge `feature` into the current branch.
`git branch -d feature`	Delete a merged branch (`-D` forces).

Remote

Command	What it does
`git clone <url>`	Copy a repo plus its full history.
`git remote add origin <url>`	Point a local repo at a remote.
`git fetch origin`	Download remote changes without merging.
`git pull`	Fetch and merge (or `--rebase`).
`git push -u origin main`	Push and set the upstream branch.

Undo and rescue

Command	What it does
`git restore file.txt`	Discard working-directory changes to a file.
`git restore --staged file.txt`	Unstage a file, keeping the edits.
`git reset --soft HEAD~1`	Undo the last commit, keep changes staged.
`git reset --mixed HEAD~1`	Undo the last commit and unstage, keep files.
`git reset --hard HEAD~1`	Undo the commit and discard changes (dangerous).
`git revert <commit>`	New commit that undoes one (safe for shared history).
`git reflog`	Log of where HEAD has been; recover lost commits.
`git stash`	Shelve dirty changes; `git stash pop` restores them.
`git cherry-pick <commit>`	Apply one commit onto the current branch.

Which undo do I need?

Find your situation on the left; run the command on the right. When in doubt, prefer the reversible options near the top.

Situation	Command
Discard an unsaved change in a file	`git restore file.txt`
Unstage a file (keep the edits)	`git restore --staged file.txt`
Undo your last local commit, keep the changes	`git reset --soft HEAD~1` or `git reset --mixed HEAD~1`
Undo a commit you already pushed	`git revert <commit>`
Recover a commit you thought was lost	`git reflog`, then `git reset --hard <sha>`

> ⚠ **Watch out**
>
> `git reset --hard` throws away uncommitted work with no prompt. On anything already pushed, reach for `git revert` instead: it undoes the change with a new commit and leaves shared history intact.

The golden rules

- **Never rewrite shared history.** Do not rebase, amend, or hard-reset commits that already exist outside your repository and that others may have based work on. Undo those with `git revert`.
- **Commit small and often.** Small, focused commits are easier to review, revert, and understand later. A good message explains the "why".
- **The reflog is your safety net.** In Git you rarely lose a commit, you lose the pointer to it. A bad reset is recoverable: the old SHA sits in `git reflog` for about 90 days, ready for `git reset --hard <sha>`.

Your 30/60/90-day path

Fluency is a sequence, not a leap. Spend the first month owning the everyday loop, the second gaining recovery reflexes, and the third reaching for the power tools.

Figure 11.2 – A 30/60/90-day path from your first commit to teaching Git to others.

Stage	Focus	You can...
Day 0-30 Foundations	The commit loop, branches, remotes	Stage, commit, branch with `git switch -c`, clone, and `git push -u origin main` without looking it up.
Day 30-60 Fluency	Merge and rebase, undo, workflows	Resolve a merge conflict, pick the right undo, and run a feature-branch workflow end to end.
Day 60-90 Mastery	Power tools and teaching	Use `git bisect` to hunt a bug, add a submodule or Git LFS, wire a hook, and explain the three trees to a teammate.

> 💼 **Business angle**
>
> A team that reaches "Fluency" together ships faster and safer: branches are free, reviews are quick, and every change is auditable and reversible. That reliability compounds across a codebase and a career.

📋 Fluency checklist

- [ ] I run the everyday loop (`status` , `add` , `commit` , `log` , `diff`) from memory.
- [ ] I branch with `git switch -c` and merge without hesitation.
- [ ] I can clone, add a remote, and push with `git push -u origin main` .
- [ ] I pick the right undo for the situation from the lookup table above.
- [ ] I reach for `git revert` on anything already pushed.
- [ ] I recover a "lost" commit with `git reflog` .
- [ ] I can explain to someone else why the reflog makes Git safe.

📌 Key takeaways

- One card covers 90% of daily Git: everyday, branch/merge, remote, and undo.
- Match the situation to the undo: `restore` for edits, `reset` for local commits, `revert` for pushed ones, `reflog` to recover.
- The golden rules: never rewrite shared history, commit small and often, trust the reflog.
- Follow the 30/60/90 path from Foundations to Mastery, then teach someone else.

🔗 Go deeper

- Official Git reference (git-scm.com/docs) – the manual page for every command on this card.
- Pro Git, 2nd ed. (Chacon & Straub) – free book; the source for the golden rules and mental models.

12. Resources

The gist

- Every official and high-quality Git link, grouped and one click away.
- A map from your question to the doc that answers it.
- Three starting points if you only open one tab today.

Reading time: ~4 min

Find the right doc fast

You rarely need to read Git top to bottom. You need the one page that answers the question in front of you. Use this map to jump straight there.

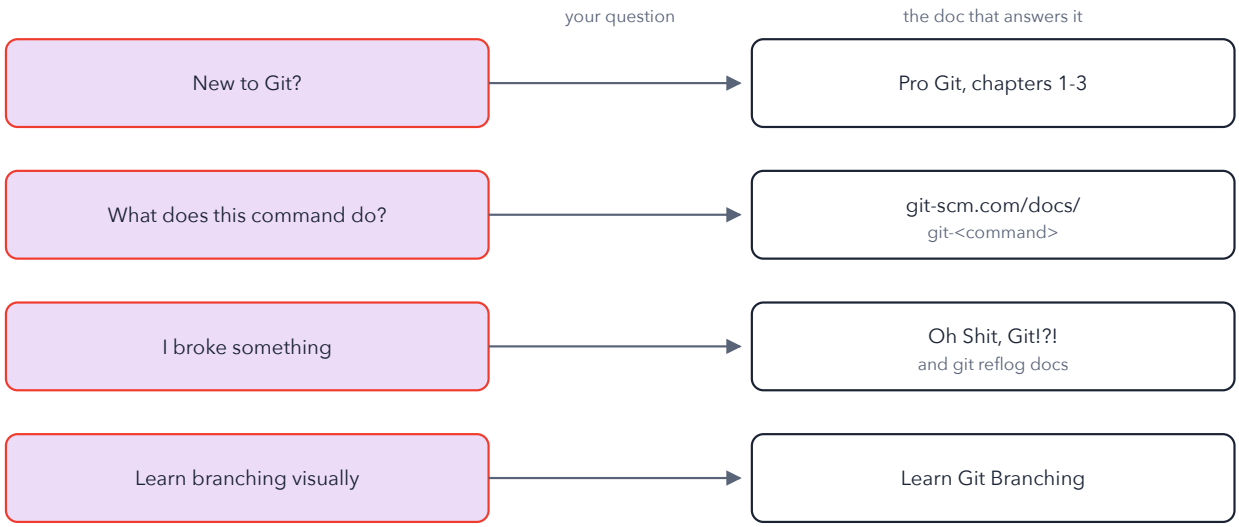


Figure 12.1 – Docs map: from the question in front of you to the resource that answers it.

Official docs & Pro Git

Start here. These are authoritative and free.

- Git home – the canonical site: downloads, docs, and the full book in one place.
- Pro Git (free) – the definitive book; read cover to cover or dip into a chapter.
- Pro Git 1.3 – What is Git? – snapshots and the three states, the mental model everything builds on.
- Pro Git 2 – Git Basics – the everyday add/commit loop, explained slowly.
- Pro Git 3 – Git Branching – why branches are cheap and how merging really works.
- Pro Git 3.6 – Rebasing – rebase done right, including the golden rule.
- Pro Git 7 – Git Tools – reset demystified, plus bisect and submodules.

- Pro Git 10 – Git Internals – objects and refs, for when you want to see the machinery.
- Reference manual – the complete, versioned command reference.
- Downloads – install Git on macOS, Windows, or Linux.
- gitignore(5) – the exact pattern syntax for ignore files.
- gitworkflows(7) – the maintainers' own take on branching workflows.

Command references

One page per command, always current. Bookmark the pattern `git-scm.com/docs/git-<command>` and you can reach any of them.

- git config – every setting, including the first-run identity you set once.
- git switch and git restore – the modern, purpose-built replacements for checkout.
- git merge – fast-forward versus three-way merges and their options.
- git rebase – replay commits onto a new base, interactive mode included.
- git reset – the three modes and exactly which tree each one touches.
- git revert – the safe undo for history you have already shared.
- git reflog – the log of where HEAD has been; your recovery net.
- git stash – shelve work in progress and come back to it later.
- git bisect – binary-search your history to pinpoint the bad commit.
- git submodule – pin one repo inside another at a specific commit.
- git hooks – the events you can hook for linting, tests, and message checks.
- Git LFS – store large binaries without bloating the repo.

Learning (interactive & recovery)

When you learn best by doing, or by getting unstuck fast.

- Learn Git Branching – an interactive, visual playground; the fastest way to build branch intuition.
- Oh Shit, Git!?! – plain-language fixes for the "how do I undo this" moments.
- Atlassian Git tutorials – clear, well-illustrated guides across every core topic.

> 📌 **Key takeaways**
>
> - Match the question to the doc: onboarding goes to Pro Git, "what does this command do" goes to git-scm.com/docs, and "I broke something" goes to Oh Shit, Git and the reflog page.
> - The git-scm.com/docs/git-<command> pattern reaches any command reference directly.
> - Learn Git Branching builds branch intuition faster than any amount of reading.

🔗 **Go deeper**

- Pro Git (free) – the one book to read if you read only one.
- Learn Git Branching – the best hands-on way to make branching click.
- Oh Shit, Git!?! – the page to keep open the day something goes wrong.

Conclusion

You started this book because Git felt like a set of incantations, powerful when they worked, terrifying when they did not. The fix was never more commands to memorize. It was a mental model, a few safe habits, and the confidence that comes from knowing you can undo almost anything.

You now have all three. You can read the commit graph and predict what a command will do to each of the three trees. You can branch and merge without flinching, sync with a remote, rewrite messy local history into something clean, and recover a commit you thought was gone. The tool stopped being a black box and became something you drive on purpose.

If you keep one idea, keep this: **Git does not lose your work, it loses the pointer to it.** History is a graph of snapshots, each one immutable and addressable, and the reflog remembers where you have been. Once you trust that, experimentation stops being risky. Branch freely, try the bold refactor, rebase the ugly history, because getting back is always a command away.

So make a scratch repository and break things on purpose. Reset too hard, rebase the wrong branch, then bring it all back with the reflog. The fastest way to lose the fear is to practice the recovery. Use the 30/60/90 path in Chapter 11 as your map, and keep the one golden rule close: never rewrite history you have already shared.

📌 **Key takeaways**

- Git is a graph of immutable snapshots; commands move pointers between the working directory, the index, and the repository.
- You almost never lose a commit, only the pointer to it, and the reflog gets it back.
- Branch and experiment freely; recovery is always one command away.
- The one rule that keeps a team safe: do not rewrite history that has left your machine.

A note on this guide

This book is an independent guide, written to be useful rather than official. Every command and concept in it was drawn from the official Git documentation at git-scm.com and the freely available Pro Git book, and checked against Git's own behavior. Where the tools have evolved, the book teaches the modern form first, such as `git switch` and `git restore`, and notes the older `git checkout` that you will still see in the wild.

A few opinions did make it in, mostly about safety: commit small and often, prefer `revert` to `reset` on anything you have shared, and never rewrite history that has left your machine. Those are judgements, stated as such, and Chapters 7 and 8 explain the reasoning so you can disagree on purpose.

One honest limitation: Git keeps improving, and defaults shift over time, from the `master` to `main` branch name to the ongoing move from SHA-1 to SHA-256 hashes. Treat version-specific details here as a strong starting point and confirm anything load-bearing against the current docs. The mental model, snapshots in a graph you can always navigate back through, ages very well.

> 💡 **Tip**
> Found something out of date, or a place where the book could be clearer? That feedback is the most valuable thing a reader can send, and it makes the next edition better for everyone.

Acknowledgments

A book about a tool this good stands entirely on the work of others.

Thank you to Linus Torvalds and the hundreds of contributors who have built and maintained Git as free software since 2005, and to the maintainers of the reference documentation at git-scm.com, which is thorough enough to teach from.

Thank you to Scott Chacon and Ben Straub, whose Pro Git book (released under a Creative Commons license) has taught Git to a generation of developers and grounded much of the mental model in these pages. Any errors that remain are the author's, not theirs.

And thank you, the reader, for choosing to understand your tools rather than just survive them. Version control is a quiet superpower, and the people who wield it with care and good habits are the reason collaboration at scale works at all.

About the author

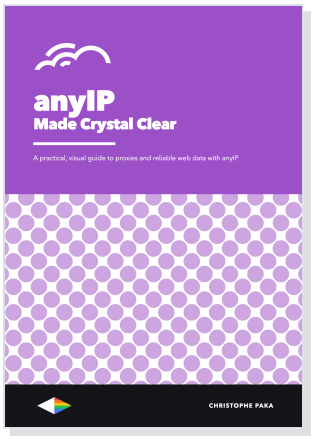
Christophe Paka is a self-taught developer and digital nomad with a deep passion for technology. He works at the intersection of building and hiring, as a technologist who recruits technologists rather than a recruiter who happens to work in tech. His YouTube channel on careers and jobs in tech has passed 700,000 views. He created this series to teach established and emerging technologies to more people, in a straightforward, visual format that gets to the point. He is always open to connecting with startup ecosystems across America, Europe, Asia, and Africa.

Thank you for reading

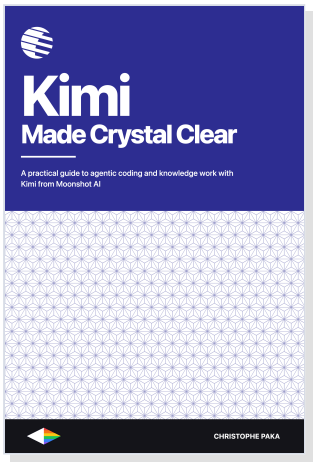
Thank you for spending your time with this book. If it made Git a little clearer and left you more confident to use it, it did its job. Keep it on a second monitor, come back to the recipes, and make the checklists your own.

Keep exploring the series

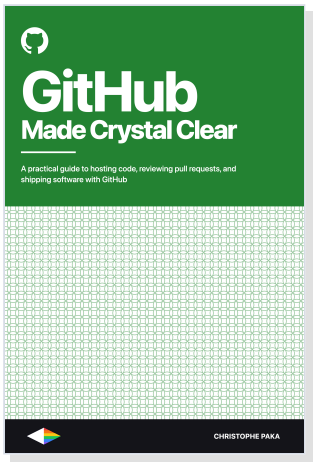
More short, visual guides in the **Made Crystal Clear** series:



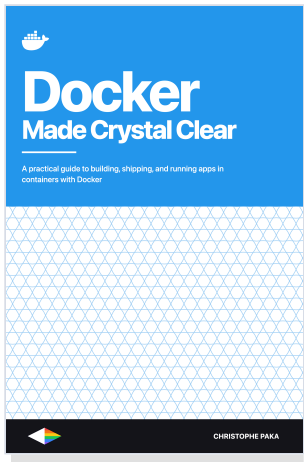
anyIP



Kimi



GitHub



Docker

We would love your feedback

Found an error, a rough spot, or something that clicked? Tell us and help improve the next edition of this book. Scan the code or visit:

madecrystalclear.dev/git/feedback

You can pick the chapter, choose a category, add a note, and attach a screenshot.

GIT · MADE CRYSTAL CLEAR

Stop fighting your own history.

Git is the tool every developer touches daily and few truly understand. This guide swaps memorized incantations for a clear mental model, so you can branch, merge, rebase, and recover with total confidence, explained visually and hands-on.

WHAT YOU'LL LEARN

- [ ] The Git mental model: snapshots, the commit graph, and the three trees
- [ ] Branch, merge, and resolve conflicts without fear
- [ ] Rewrite history safely with rebase, amend, and interactive rebase
- [ ] Undo almost anything with reflog, restore, reset, and revert
- [ ] Collaborate with remotes, pull requests, and clean branching workflows
- [ ] Everyday power tools: stash, cherry-pick, bisect, blame, and tags
- [ ] Scale to big repositories with submodules, LFS, and hooks

Who it's for: *Developers, data and ML engineers, and anyone who writes code or docs and wants version control they actually understand.*

Christophe Paka
Christophe Paka is a self-taught developer, digital nomad, and creator of a 700K-view YouTube channel on tech careers. He builds this series to make established and emerging tech clear for everyone.

Explore the full **Made Crystal Clear** series and more books at **madecrystalclear.dev**