SOFTWARE

practical git

git different scenarios for better understanding

BY

Milad Golfam

MIGOLFAM.COM

1ST EDITION

1ST EDITION

practical git

git different scenarios for better understanding

Milad Golfam

migolfam.com · 2026

practical git — git different scenarios for better understanding

1st edition · Version 1.0.0 · 2026

Published by migolfam.com.

git different scenarios for better understanding

Built with the open book-factory pipeline.

Preface

I wrote this book because of a moment I have lived through hundreds of times: that jolt of panic when something goes wrong in Git, the work feels lost, and the official docs explain every flag of the command that just hurt you while saying nothing about how to get your work back.

Git is the tool nearly every developer touches every day, and yet it is the one most of us understand the least. We learn three or four commands by copying others, we grab a fix from a forum thread when we are stuck, and we cross our fingers. That works right up until the day it doesn't — and then we are staring at a detached `HEAD`, a tangled rebase, or a force-push that wiped a teammate's branch, with no idea what happened or how to undo it.

The gap this book fills

There is no shortage of Git material, but almost all of it sits at one of two extremes. On one end is the dry command reference: complete, correct, and nearly useless in a crisis because it is organized around commands instead of around the trouble you are actually in. On the other end is the gentle beginner tutorial that teaches `add`, `commit`, and `push`, then stops right where real life begins.

The space in between is where developers actually live, and it is mostly empty. This book is my attempt to fill it. Instead of asking *"what does this command do?"*, it asks *"what situation are you in, and how do you get out of it cleanly?"* Every chapter is built around a real scenario — the kind you meet on a real team with real deadlines and real mistakes.

Who this book is for

This is a book for developers and maintainers who already use Git and want to stop being afraid of it. If you are fine making commits but break into a cold sweat at the word "rebase," you are exactly who I had in mind. If you maintain a project — reviewing other people's history, untangling their merges, guarding a shared branch — there is a whole layer here for you too.

You do not need to be an expert. You do need to be past your very first day: you can clone a repo, make a commit, and push. The book climbs steadily from there to advanced ground, and you can stop wherever your needs are met.

The one idea that makes Git click

Most of the fear around Git comes from a broken mental model. We are told it is "a folder of versions" or "like saving copies of your project," and that picture falls apart the moment things get real. So if you take one thing from this book, take this:

Commits are snapshots that never change

A commit is a frozen picture of your tracked files at a moment in time, plus a pointer to the commit before it. You never edit a commit; you make new ones. That is why a commit hash always means exactly the same thing, forever.

Branches and HEAD are just movable labels

A branch is not a copy of your code. It is a tiny note holding one commit hash — a sticky note that says "the tip of this work is *here*." `HEAD` says "you are here right now." Switching branches just moves `HEAD`; making a branch just writes a new note.

Almost nothing is ever truly lost

Because commits never change and Git logs everywhere its pointers have been, the work you think you destroyed is usually still there, reachable with `git reflog`. Recovery is the rule, not the exception.

Once you see commits as snapshots and branches as cheap, movable labels, the scary commands stop being magic. Merge, rebase, reset, cherry-pick — they are all just different ways of making commits and moving labels. Chapter 1 builds this model out properly; keep it in the back of your mind until then.

What makes this book different

- **Scenario-first.** You arrive with a problem and leave with a fix. Commands are taught where you will actually reach for them.
- **Every merge, rebase, and cherry-pick situation.** Not just the happy path, but the conflicts, the half-finished operations, and the "wait, that's not what I meant" cases.
- **Mistakes and recovery at the core.** Lost commits, bad resets, overwritten branches, detached states. The `reflog` is your friend, and I will introduce you properly.
- **Modern, AI-era practice.** The way we use Git in an age of AI assistants and automated tooling has shifted, and this book reflects how the craft is done today, not a decade ago.

How to read it

You do not have to read this cover to cover, though you are welcome to. There are three honest ways through it:

1. Keep the **cheat sheet up front** within reach and use it as a fast daily reference.
2. **Jump straight to the scenario** you are facing when something has gone wrong right now. The chapters stand on their own.
3. **Read it cover to cover** when you want deep, lasting confidence rather than a patch for one hole.

Break things on purpose

The best way to learn Git is to break it somewhere safe. Throughout the book I will invite you to follow along in a throwaway repo. Make one now:

```
mkdir git-scratch && cd git-scratch
git init
echo "hello" > file.txt
git add file.txt
git commit -m "first commit"
```

> **`git reflog` is your safety net.** Experiment freely in a scratch repo — reset, rebase, delete branches, make a mess. As long as you committed at some point, `git reflog` remembers where every pointer has been and lets you get back. You almost certainly cannot lose committed work by trying things, so try things.

This book is shared freely, because the knowledge in it was given freely to me by countless others over the years. If it saves you from one bad afternoon, that is reason enough for me. If you find it valuable and are able to support the work so I can keep writing and keep it open, I am grateful. Thank you for reading, thank you for sharing it with someone who needs it, and thank you for caring enough about your craft to keep learning.

— Milad Golfam

Enjoyed this book?

This book is the result of many hours of careful work, and it is shared freely. If it helped you learn, saved you time, or earned you money, please consider sending a little crypto to support more open books like it — any amount is deeply appreciated. After you donate, I would love to hear from you: drop me a line so I can thank you personally.

از این کتاب لذت بردید؟

اگر این کتاب به شما کمک کرد، لطفاً با یک کمک مالی کوچک از انتشار کتاب‌های آزاد بیشتری مانند این حمایت کنید – هر مبلغی مورد قدردانی است. می‌توانید مبلغ را به کارت بانکی (بلوبانک) داخل ایران واریز کنید یا از طریق ارز دیجیتال (سولانا، بایننس‌کوین، تتر یا ترون) بپردازید. پس از پرداخت، برای من ایمیل بفرستید تا شخصاً از شما تشکر کنم.

How to donate with crypto

1. **Get some crypto.** Buy a little of any coin listed below on an exchange — global ones like CoinEx, Binance, Coinbase, KuCoin, OKX, or Kraken, or Iranian exchanges like Nobitex, Wallex, Tabdeal, Ramzinex, Bitpin, or Excoino — or use what you already hold in a self-custody wallet.
2. **Open your wallet** and tap *Send* (or *Withdraw* if you're on an exchange).
3. **Pick the matching network.** Send each coin on the network shown on its card (e.g. USDT on BSC/BEP20, TRX on TRON/TRC20). Using the wrong network can lose funds.
4. **Enter my address.** Scan the coin's QR code, or copy and paste the address shown under it. Always double-check the first and last few characters match.
5. **Choose any amount and confirm.** Send whatever you like — a small network fee applies.
6. **Say hello.** After it sends, email me so I can thank you personally.

New to crypto? Any wallet app will walk you through buying it in a few minutes — the wallet addresses and QR codes are on the next page.

پرداخت به کارت بانکی — بلوبانک

شماره کارت	6219 8618 0347 3585
شماره شبا	IR82 0560 6118 2800 6029 3124 01
به نام	میلاد گلفام

Crypto wallets

SOL · SOLANA

`DpxDrax2j2u7kXZ7GykS1Rs7bNtQLDcg8xo3Z2vVeuNV`

BNB · BNB SMART CHAIN · BEP20

`0xB61c318424993c28c160C6637edC25756fd59766`

USDT · BNB SMART CHAIN · BEP20

`0xB61c318424993c28c160C6637edC25756fd59766`

TRX · TRON · TRC20

`TRmcn68eRAjc5vsBaUtHwY8F5iGMD5BngF`

After you send, email **hi@migolfam.com** so I can thank you personally. 🙏

Thank you for supporting open books.

Contents

Git Command Cheat Sheet

A dense, one-stop reference to the Git commands you actually use. Skim it, keep it open, come back to it. For the *why* behind any command, jump to the matching chapter.

Setup & config

GOAL	COMMAND
Set your identity	`git config --global user.name "Name"` · `user.email "you@x.com"`
Default branch name	`git config --global init.defaultBranch main`
Merge (not rebase) on pull	`git config --global pull.rebase false`
Auto-set upstream on push	`git config --global push.autoSetupRemote true`
Default editor	`git config --global core.editor "code --wait"`
Colored output	`git config --global color.ui auto`
Make an alias	`git config --global alias.lg "log --oneline --graph"`
List / edit all config	`git config --list` · `git config --global --edit`
Cache credentials	`git config --global credential.helper cache`

Start a repo

GOAL	COMMAND
New repo here	`git init`
Clone a repo	`git clone <url>`
Clone into a folder	`git clone <url> <dir>`
Shallow clone (faster)	`git clone --depth 1 <url>`
Clone one branch	`git clone -b <branch> --single-branch <url>`

The daily loop

```
git status               # what changed? (run constantly)
git add <file>            # stage a file
git commit -m "message"  # save a snapshot
git pull                 # get teammates' changes
git push                 # share yours
```

Staging & committing

GOAL	COMMAND
Stage a file / everything	`git add <file>` · `git add -A`
Stage all in current dir	`git add .`
Stage interactively / by hunk	`git add -i` · `git add -p`
Unstage a file	`git restore --staged <file>`
Commit staged changes	`git commit -m "..."`
Stage tracked + commit	`git commit -am "..."`
Commit with body	`git commit -m "title" -m "body"`
Amend last commit	`git commit --amend`
Amend without editing message	`git commit --amend --no-edit`
Empty / WIP commit	`git commit --allow-empty -m "..."`
Remove a file (staged)	`git rm <file>` · keep on disk: `git rm --cached <file>`
Move / rename a file	`git mv <old> <new>`

Inspecting changes

GOAL	COMMAND
Working-tree changes	`git diff`
Staged changes	`git diff --staged`
Changes vs a commit / branch	`git diff <ref>` · `git diff main..feature`
Summary of changed files	`git diff --stat`
Show a commit	`git show <sha>`
Show a file at a revision	`git show <sha>:<path>`
Who changed each line	`git blame <file>`
What's ignored / tracked	`git status --ignored` · `git ls-files`

History & search

GOAL	COMMAND
Compact log	`git log --oneline`
Graph of all branches	`git log --oneline --graph --all`
Log with patch / stat	`git log -p` · `git log --stat`
Limit / filter	`git log -n 10` · `--author=x` · `--since="2 weeks"`
History of one file	`git log --follow <file>`
Search commit messages	`git log --grep="text"`
Search code changes ("pickaxe")	`git log -S "text"` · regex: `-G`
Commits on A not on B	`git log B..A --oneline`
Find the commit that broke it	`git bisect start` · `bad` · `good <sha>` · `reset`

Branching

GOAL	COMMAND
List branches (+remote / verbose)	`git branch` · `-a` · `-vv`
Create + switch	`git switch -c <name>`
Switch to existing	`git switch <name>`
Switch to previous branch	`git switch -`
Create branch from a commit	`git switch -c <name> <sha>`
Rename a branch	`git branch -m <old> <new>`
Delete merged / force delete	`git branch -d <name>` · `-D <name>`
Set upstream	`git branch -u origin/<name>`
Merged / unmerged branches	`git branch --merged` · `--no-merged`

Merge

GOAL	COMMAND
Merge a branch in	`git merge <name>`
Always create a merge commit	`git merge --no-ff <name>`
Only if fast-forward possible	`git merge --ff-only <name>`
Combine into one commit	`git merge --squash <name>`
Abort an in-progress merge	`git merge --abort`
Resolve, then finish	edit files · `git add <file>` · `git commit`

Rebase

Only rebase commits you have not shared.

GOAL	COMMAND
Replay branch on latest main	`git rebase main`
Interactive (squash / reorder / edit)	`git rebase -i HEAD~<N>`
Move branch to a new base	`git rebase --onto <newbase> <oldbase>`
Continue after fixing conflict	`git add <file> && git rebase --continue`
Skip / abort	`git rebase --skip` · `git rebase --abort`
Push a rebased branch safely	`git push --force-with-lease`

> In `rebase -i`, each line is a commit: `pick` keep, `squash` fold into the one above, `reword` edit message, `drop` delete, reorder lines to reorder commits.

Cherry-pick

GOAL	COMMAND
Apply one commit here	`git cherry-pick <sha>`
Apply a range	`git cherry-pick <sha1>^..<sha2>`
Apply but don't commit	`git cherry-pick -n <sha>`
Continue / abort	`git cherry-pick --continue` · `--abort`

Remotes, fetch, pull & push

`fetch` downloads without touching your files; `pull` = `fetch` + merge.

GOAL	COMMAND
List / add / remove remotes	`git remote -v` · `git remote add <name> <url>` · `git remote remove <name>`
Change a remote URL	`git remote set-url origin <url>`
Download, don't merge	`git fetch` · all remotes: `git fetch --all`
See what arrived	`git log HEAD..origin/main --oneline`
Fetch + merge / rebase	`git pull` · `git pull --rebase`
Push / push new branch	`git push` · `git push -u origin <name>`
Push tags	`git push --tags`
Delete a remote branch	`git push origin --delete <name>`
Prune stale remote branches	`git fetch --prune`
Force push (safely)	`git push --force-with-lease`

Stash — set work aside

GOAL	COMMAND
Shelve changes / with a label	`git stash` · `git stash push -m "wip"`
Include untracked files	`git stash -u`
List / view	`git stash list` · `git stash show -p`
Reapply + drop / keep	`git stash pop` · `git stash apply`
Drop / clear	`git stash drop` · `git stash clear`
Branch from a stash	`git stash branch <name>`

Tags — mark releases

GOAL	COMMAND
Annotated / lightweight tag	`git tag -a v1.0 -m "Release 1.0"` · `git tag v1.0`
Tag an old commit	`git tag -a v1.0 <sha>`
List / show	`git tag` · `git show v1.0`
Push one / all tags	`git push origin v1.0` · `git push --tags`
Delete local / remote	`git tag -d v1.0` · `git push origin --delete v1.0`

Undo & reset

I WANT TO…	COMMAND
Discard file changes (uncommitted)	`git restore <file>`
Restore a file from a commit	`git restore --source <sha> <file>`
Unstage a file	`git restore --staged <file>`
Undo last commit, keep changes staged	`git reset --soft HEAD~1`
Undo last commit, keep changes unstaged	`git reset HEAD~1`
Discard last commit *and* changes	`git reset --hard HEAD~1`
Reset a branch to a commit	`git reset --hard <sha>`
Safely undo a pushed commit	`git revert <commit>`
Remove untracked files / dirs	`git clean -n` (preview) · `git clean -fd`

Recovery — "I broke it"

OH NO…	DO THIS
Lost commits after a bad reset	`git reflog`, then `git reset --hard <sha>`
Recover a deleted branch	`git reflog`, then `git switch -c <name> <sha>`
Committed to the wrong branch	`git switch -c right`, then reset the wrong one
Abort a merge / rebase	`git merge --abort` · `git rebase --abort`
Find a dangling commit	`git fsck --lost-found`
Committed a secret	remove it, **rotate the secret**, see the Recovery Toolkit

Power tools

GOAL	COMMAND
Multiple working trees	`git worktree add ../dir <branch>` · `git worktree list`
Submodules	`git submodule add <url>` · `git submodule update --init --recursive`
Apply a single commit as a patch	`git format-patch -1 <sha>` · `git apply <file>`
Archive a snapshot	`git archive -o out.zip HEAD`
Garbage-collect / verify	`git gc` · `git fsck`
Mark good/bad known revision	`git bisect run <test-cmd>`

Three rules that keep you safe: commit small and often; never rewrite history others have pulled (no force-push on shared branches); and when something looks lost, run `git reflog` before you panic — in Git, almost nothing is truly gone.

Terms in one line

Commit

An immutable snapshot of the whole project, identified by a hash like `a1b2c3d`.

Branch

A movable pointer to a commit — cheap, not a copy of your files.

HEAD

A pointer to where you are now, usually the tip of the current branch.

Index (staging area)

The holding zone between working files and the next commit.

Remote

A shared copy of the repo (e.g. `origin`); `push` sends, `fetch` / `pull` bring.

Notation

This page lists the typographic and command conventions used throughout the book. Skim it once now; refer back whenever a symbol, placeholder, or callout is unclear. Every scenario in *Practical Git* follows these rules consistently. Once you learn them, the recipes read the same way from cover to cover.

Command blocks

Commands you run are shown in monospaced blocks. A leading `$` represents the shell prompt and is **not** typed by you. Lines beginning with `#` are comments meant for the reader, not commands. Output is shown without a prompt so you can tell input from result at a glance.

```
# Inspect the current branch and its upstream
$ git status -sb
## main...origin/main
```

> Copy only the text after the `$`. If you paste the prompt by accident, your shell will complain about an unknown command.

Placeholders

Anything wrapped in angle brackets is a placeholder you must replace with your own value. For example, `git switch <branch>` means "type your real branch name where `<branch>` appears." Literal text outside the brackets is typed exactly as printed. So in `git push origin <branch>`, the words `git push origin` are literal and only `<branch>` changes.

Callout legend

Boxed callouts highlight information that does not belong in the main flow. Each type signals a different intent:

CALLOUT	SIGNALS
Note	Background or clarification that adds useful context.
Tip	A shortcut, best practice, or quality-of-life suggestion.
Key	A core idea worth remembering; the takeaway of a section.
Warning	A destructive or irreversible action. Read before running.
Example	A worked illustration applying the surrounding concept.

> Warnings flag commands that rewrite history, discard work, or delete refs (for instance `git reset --hard` or a force push). Once run, recovery may be difficult or impossible.

Example identifiers

To keep scenarios concrete and repeatable, the book reuses a small cast of made-up identifiers.

Commit SHAs

Shown abbreviated to seven characters, such as `a1b2c3d`. Your real hashes will differ; match by position in the graph, not by exact value.

Branch names

Common examples are `main`, `develop`, `feature/login`, and `release/1.4`.

Remotes

`origin` is your primary remote; `upstream` is the original project you forked from.

ASCII commit graphs

Commit history is a directed acyclic graph (DAG). The book draws it the way `git log --graph` does: an asterisk `*` marks a commit, a vertical bar `|` continues a branch line, and lines that fork or join show branching and merges. Read these graphs bottom-to-top, oldest to newest.

```
*   3f1a9c2  (main) Merge feature/login
|\
| * 9d4e8b1  (feature/login) Add login form
* | 7c2f0a5  Update README
|/
*   a1b2c3d  Initial commit
```

Here `feature/login` branched off `a1b2c3d` and was later merged back into `main` at the merge commit `3f1a9c2`.

Platform notes

Examples target **Git 2.30 or newer** on a POSIX shell (Linux, macOS, or Git Bash on Windows). Most commands are identical everywhere, but a few details differ on native Windows.

- On Windows `cmd` or PowerShell, path separators and quoting rules differ; the book assumes Git Bash, where POSIX syntax works.
- Line endings can vary; configure `core.autocrlf` appropriately if you mix platforms.
- Where a command genuinely differs by platform, a Note callout points it out inline.

> If your Git is older than 2.30, modern verbs like `git switch` and `git restore` may be missing. Upgrade, or fall back to `git checkout` as noted in the relevant scenarios.

PART 01

Foundations & Quick Reference

Mental model of Git plus a fast command cheat sheet you can keep open while you work.

How to Use This Book

This is a working manual for the moments when Git stops being a quiet background tool and becomes the thing standing between you and a deadline: a commit on the wrong branch, a force-push that ate a colleague's work, a rebase that left you stranded. This chapter explains the book's method — how each scenario is laid out — and gives you the small mental model of Git that makes every later fix feel obvious instead of magical. We assume you can already clone a repo, make a commit, and push.

The scenario-first approach

Most Git documentation answers "what does this flag do?" This book answers "I am in trouble — what do I type?" That difference shapes everything. Each chapter starts from a concrete, realistic mess and walks you out of it, instead of listing options you have to piece together yourself under pressure.

How chapters are organized

Scenarios are grouped by theme — undoing changes, branch surgery, collaboration accidents, history rewriting, and recovery from work that looks lost. Within a group, they go from common and mild to rare and dangerous. Read top to bottom, or treat the table of contents as a symptom index and jump to your problem.

The problem → commands → resolution pattern

Every scenario follows the same three beats, so you always know where you are:

1. **Problem** — a plain description of the situation and how you got here, including how to check that you are really in this case and not a look-alike.
2. **Commands** — the exact sequence to run, with comments explaining what each line changes and what it leaves alone.
3. **Resolution** — how to confirm the fix worked, what state you are now in, and how to avoid the problem next time.

Here is the pattern in miniature — you committed to `main` when you meant to commit to a feature branch:

```
# Problem: the last commit is on main but belongs on a feature branch.
git log --oneline -1          # confirm the stray commit is at the tip

# Commands: move it onto a new branch, then rewind main.
git switch -c feature/login   # create + switch, taking the commit with you
git switch main               # go back to main
git reset --hard HEAD~1       # drop the commit from main (it lives on the branch
now)
```

```
Switched to a new branch 'feature/login'
Switched to branch 'main'
HEAD is now at a1b2c3d Previous main commit
```

The **resolution** step would then have you run `git log --oneline --all --graph` to confirm the commit sits on `feature/login` and that `main` no longer points at it.

> **Warning:** `git reset --hard` throws away uncommitted changes in your working tree. The scenarios always tell you when a command is destructive — read those lines before you press Enter, especially on a shared branch.

How Git actually thinks: commits, refs, the DAG, and HEAD

Almost every confusing Git error clears up once you hold the right mental model. Git is not tracking diffs or files the way you might imagine. It is a small database of snapshots, plus a handful of pointers into that database. Learn the four ideas below and the rest of the book reads like common sense.

Commits are snapshots with parents

A commit is a complete snapshot of your tracked files at one moment, plus some metadata (author, message, timestamp) and one or more *parent* commits — the commit(s) it came from. Each commit has a unique id (a SHA) derived from its contents, which is why even a tiny change makes a brand-new commit. You can look at any commit object directly:

```
# Show the raw contents of the current commit object.
git cat-file -p HEAD
```

```
tree 9f2a1c4b8e7d6f5a3c2b1d0e9f8a7b6c5d4e3f2a
parent 7c6b5a4938271605f4e3d2c1b0a9f8e7d6c5b4a3
author Milad Golfam <milad.golfam@gmail.com> 1750800000 +0000
committer Milad Golfam <milad.golfam@gmail.com> 1750800000 +0000

    Add login form validation
```

Notice the `tree` (the snapshot of files) and the `parent` (the commit before this one). Follow parent links backward and you walk the whole history of the project.

Refs, branches, and tags are movable pointers

A commit's SHA is no fun to type, so Git gives commits human names called *refs*. A **branch** is just a ref that points at one commit and *moves forward on its own* as you commit. A **tag** is a ref that points at one commit and normally *never moves* — handy for marking releases. Both are just files under `.git/refs/` holding a SHA. That is the whole secret: making a branch is cheap because it only writes one line naming a commit that already exists.

REF TYPE	POINTS TO	MOVES WHEN YOU COMMIT?	TYPICAL USE
Branch	A commit	Yes, follows the tip	Active lines of work
Tag	A commit (or tag object)	No, stays put	Releases, fixed markers
HEAD	A branch (usually)	Indirectly, via its branch	"Where you are now"

The commit graph is a DAG

Because every commit records its parents, the full history forms a **directed acyclic graph** (DAG): *directed* because parent links point one way (toward older commits), *acyclic* because you can never become your own ancestor. Branches and merges are just shapes in this graph — a branch is a path, a merge is a commit with two parents joining two paths. You can see it any time:

```
# Draw the DAG with branch and merge structure.
git log --oneline --graph --all
```

```
*   d4e5f6a (HEAD -> main) Merge feature/login
|\
| * b2c3d4e (feature/login) Add login form validation
| * a1b2c3d Scaffold login route
|/
* 9f8e7d6 Update README
```

HEAD is "where you are"

HEAD is the pointer that answers "which commit will my next commit build on?" Normally HEAD points at a branch (which points at a commit), so committing moves both forward. Check it any time:

```
git rev-parse HEAD                # the exact SHA you are sitting on
git rev-parse --abbrev-ref HEAD   # the branch name HEAD follows, or "HEAD" if detached
```

```
d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0c1d2e3
main
```

> **Note:** If HEAD points straight at a commit instead of a branch, you are in *detached HEAD* state. Commits you make there are not on any branch and can be cleaned up by Git. A later scenario is devoted to rescuing them — for now, just know `git switch -c <name>` turns a detached HEAD back into a real branch.

The three areas

Finally, every tracked file lives in three places at once, and most everyday commands just move content between them:

- **Working tree** — the actual files on disk you edit.
- **Index (staging area)** — the snapshot you are building for the next commit; `git add` copies working-tree changes here.
- **Repository** — the committed history; `git commit` writes the staged snapshot here for good.

Modern Git splits the old `git checkout` into two clearer verbs that map onto these areas: `git switch` changes which branch HEAD follows, and `git restore` copies files from a commit or the index back into your working tree or index. Both still work alongside `checkout`, which you will still see in older answers online.

```
git restore <file>            # discard working-tree edits, restore from the index
git restore --staged <file>   # unstage: move from index back, keep working-tree
edits
git switch <branch>           # move HEAD to another branch
```

> **Key idea:** Commits are snapshots that never change, sitting in a DAG; branches, tags, and HEAD are cheap movable pointers *into* that DAG; and your files flow working tree → index → repository. Nearly every recovery in this book is just repointing a ref or moving content between those three areas. Hold this model in mind and the scary commands stop being scary.

Recap

This book teaches Git by walking out of real problems, each laid out as **problem → commands → resolution**, with destructive steps clearly flagged. The foundation under all of it is one small model: snapshots in a DAG, movable refs, HEAD as your current position, and three areas your files travel through. With that in hand, turn to whichever scenario matches your trouble — and remember the scratch repo from the preface is always there to rehearse in first.

The Command Cheat Sheet

Want a bare list of commands to scan? Use the cheat sheet in the front matter. This chapter is the annotated version: the everyday commands grouped by job, each with an example and the traps worth knowing. Keep it open in a second window, jump to the topic you need, copy the example, and move on. Everything here reflects modern Git, including `git switch` and `git restore`, which split the overloaded `git checkout` into clearer tools.

> **Tip:** Placeholders are written as `<branch>`, `<commit>`, `<file>`, and `<remote>`. Replace them with your real values. `HEAD` means "the commit you are on right now", and `HEAD~1` means "one commit before that".

The 20 commands you use every day

If you memorize one table in this book, make it this one. These commands cover the vast majority of daily work.

COMMAND	WHAT IT DOES
`git status`	Show what is staged, unstaged, and untracked.
`git add <file>`	Stage changes for the next commit.
`git commit -m "msg"`	Record staged changes with a message.
`git diff`	Show unstaged changes (working tree vs. index).
`git diff --staged`	Show staged changes (index vs. last commit).
`git log --oneline`	Compact one-line-per-commit history.
`git switch <branch>`	Move to an existing branch.
`git switch -c <branch>`	Create a new branch and switch to it.
`git branch`	List local branches (current one marked).
`git restore <file>`	Discard unstaged changes in a file.
`git restore --staged <file>`	Unstage a file (keep the edits).
`git fetch`	Download remote changes without merging.
`git pull`	Fetch and integrate remote changes.
`git push`	Upload local commits to the remote.
`git merge <branch>`	Join another branch into the current one.
`git rebase <branch>`	Replay your commits on top of another branch.
`git stash`	Shelve uncommitted changes for now.
`git commit --amend`	Fix the most recent commit.
`git reset <commit>`	Move the branch pointer; reshape history.
`git reflog`	Show where HEAD has been (your safety net).

Setup and config

Before anything else, tell Git who you are and set it up the way you like. Config lives at three levels: `--system` (all users), `--global` (your user), and local (the current repo, used by default when no flag is given).

COMMAND	WHAT IT DOES	EXAMPLE
`git init`	Create a new repo in the current folder.	`git init my-project`
`git clone`	Copy a remote repo locally.	`git clone git@host:org/repo.git`
`git config`	Read or set config values.	`git config --global user.name "Ada"`
`git remote add`	Register a remote URL under a name.	`git remote add origin <url>`

```
# Identity (do this once per machine)
git config --global user.name "Ada Lovelace"
git config --global user.email "ada@example.com"

# Handy aliases
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.lg "log --oneline --graph --decorate --all"

# Sensible defaults
git config --global init.defaultBranch main
git config --global pull.rebase false
```

> **Note:** The email you set as `user.email` ends up in every commit and is public in shared history. Use an address you are fine exposing, or a provider's no-reply address.

Staging and committing

Git keeps three areas apart: the working tree, the staging area (index), and committed history. Staging lets you build a clean commit out of messy edits.

COMMAND	WHAT IT DOES	EXAMPLE
`git status`	Summarize working tree and index.	`git status -s`
`git add`	Stage files for the next commit.	`git add src/app.js`
`git add -p`	Stage selected hunks interactively.	`git add -p`
`git diff`	Show unstaged changes.	`git diff src/`
`git diff --staged`	Show what will be committed.	`git diff --staged`
`git commit`	Commit staged changes (opens editor).	`git commit`
`git commit -m`	Commit with an inline message.	`git commit -m "Fix login bug"`
`git restore --staged`	Unstage a file (keep edits).	`git restore --staged app.js`
`git restore`	Throw away unstaged edits.	`git restore app.js`

> **Warning:** `git restore <file>` permanently discards unstaged changes in that file — there is no commit to recover from. Run `git stash` first if you might want them back.

Branching and navigation

Branches are cheap, movable pointers. Use `git switch` to move between them and `git log` to see where you are.

COMMAND	WHAT IT DOES	EXAMPLE
`git branch`	List, create, or delete branches.	`git branch -d old-feature`
`git switch <branch>`	Switch to an existing branch.	`git switch main`
`git switch -c <branch>`	Create and switch to a new branch.	`git switch -c feature/x`
`git checkout`	Older all-in-one switch/restore.	`git checkout main`
`git log`	Show commit history.	`git log -5`
`git log --oneline --graph`	Visual, compact branch topology.	`git log --oneline --graph --all`
`git show <commit>`	Show one commit's message and diff.	`git show HEAD`

> **Tip:** `git switch -` jumps back to the previous branch, just like `cd -` in a shell.

Syncing with remotes

Remotes are other copies of your repo. Keep yours in sync on purpose: fetch to look, pull to integrate, push to share.

COMMAND	WHAT IT DOES	EXAMPLE
`git remote -v`	List configured remotes and URLs.	`git remote -v`
`git fetch`	Download remote refs, do not merge.	`git fetch origin`
`git pull`	Fetch then merge into current branch.	`git pull origin main`
`git pull --rebase`	Fetch then rebase local commits on top.	`git pull --rebase`
`git push`	Send commits to the remote.	`git push`
`git push -u`	Push and set upstream tracking.	`git push -u origin feature/x`
`git push --force-with-lease`	Force push only if the remote is unchanged.	`git push --force-with-lease`

> **Warning:** Never use a plain `git push --force` on shared branches; it can silently erase a teammate's pushed commits. Always prefer `--force-with-lease`, which refuses to overwrite work you have not seen.

History rewriting

These commands change commits that already exist. They are great for cleaning up your local work and dangerous once history is shared.

COMMAND	WHAT IT DOES	EXAMPLE
`git commit --amend`	Replace the last commit.	`git commit --amend -m "Better msg"`
`git rebase <base>`	Replay commits onto a new base.	`git rebase main`
`git rebase -i`	Interactively reorder/squash/edit.	`git rebase -i HEAD~3`
`git reset --soft`	Move HEAD; keep index and files.	`git reset --soft HEAD~1`
`git reset --mixed`	Move HEAD; unstage; keep files (default).	`git reset HEAD~1`
`git reset --hard`	Move HEAD; discard index and files.	`git reset --hard HEAD~1`
`git revert <commit>`	New commit that undoes another.	`git revert abc123`

> **Warning:** `git reset --hard` deletes uncommitted work in your working tree with no easy undo. Commit or stash first. To undo a *public* commit safely, use `git revert` instead of rewriting history.

Moving work around

Sometimes you need to bring a change from one place to another, or set work aside for a bit.

COMMAND	WHAT IT DOES	EXAMPLE
`git merge <branch>`	Combine another branch into this one.	`git merge feature/x`
`git cherry-pick <commit>`	Apply a single commit here.	`git cherry-pick abc123`
`git stash`	Shelve uncommitted changes.	`git stash push -m "wip"`
`git stash pop`	Reapply and drop the latest stash.	`git stash pop`
`git stash list`	Show shelved stashes.	`git stash list`

```
# Classic "wrong branch" rescue
git stash            # shelve your edits
git switch correct-branch
git stash pop        # bring them back here
```

Investigation

When something is broken or surprising, these commands answer "who", "when", and "where".

COMMAND	WHAT IT DOES	EXAMPLE
`git blame <file>`	Show who last changed each line.	`git blame -L 20,40 app.js`
`git bisect`	Binary-search for the bad commit.	`git bisect start`
`git reflog`	List every position HEAD has held.	`git reflog`
`git show <commit>`	Inspect a specific commit.	`git show HEAD~2`
`git log -S"text"`	Find commits adding/removing a string.	`git log -S"apiKey"`
`git log -G"regex"`	Find commits whose diff matches a regex.	`git log -G"fetch"`
`git log --follow <file>`	Track a file across renames.	`git log --follow old.js`

```
# A full bisect session
git bisect start
git bisect bad                 # current commit is broken
git bisect good v1.4.0         # this old tag worked
# ... test, then mark each step ...
git bisect good   # or: git bisect bad
git bisect reset               # finish, return to where you were
```

The "oh no" recovery commands at a glance

Mistakes are normal, and almost everything is recoverable thanks to the reflog. Match your situation to the fix below.

I DID THIS BY MISTAKE	USE THIS
Committed on the wrong branch	`git reset --soft HEAD~1`, switch branch, then commit
Bad `--amend`, want the old commit	`git reset --hard HEAD@{1}` (from `git reflog`)
Lost a commit after reset/rebase	`git reflog` to find it, then `git reset --hard <sha>`
Want to undo a merge (already committed)	`git revert -m 1 <merge-sha>`
Staged a file by accident	`git restore --staged <file>`
Discarded edits, want them back	Check `git stash list`; otherwise often unrecoverable
Deleted a branch too soon	`git reflog`, then `git switch -c <name> <sha>`
Wrong commit message (last commit)	`git commit --amend`
Pushed a broken commit publicly	`git revert <sha>` then push

Key idea: Reset and rebase do not truly delete commits; they just leave them unreferenced until Git prunes them later. `git reflog` remembers where HEAD has been, so a lost commit is almost always one `reflog` lookup away.

Tip: Three habits prevent most disasters: commit small and often, run `git status` before destructive commands, and remember `git reflog` exists.

PART 02

Everyday Workflows

The daily loop: stage, commit, inspect, branch.

Staging, Committing & Inspecting

Most of your day with Git is spent in a tight loop: change files, choose what to record, write a commit, and look at what happened. Mastering this loop — and the three "trees" that underpin it — turns Git from a black box into a precise instrument. This chapter builds the mental model, then shows how to stage surgically and read history like a maintainer.

The three trees: working dir, index, and HEAD

Git manages your work across three structures. Understanding them is the single most useful thing you can learn for everyday work.

Working directory
: The actual files on disk that you edit. This is the only tree you touch directly with your editor.

Index (a.k.a. the staging area)
: A snapshot of what your *next* commit will contain. It sits between your edits and history.

HEAD
: A pointer to the latest commit on your current branch — the snapshot of the project as it was last committed.

`git add` copies content from the **working directory into the index**. `git commit` records the **index into a new commit** that HEAD then points to.

```
# Edit a file, then move it working dir → index
git add src/login.js

# Move index → HEAD by recording a commit
git commit -m "Fix login redirect"
```

Reading `git status`

`git status` reports the differences between these trees. Its two key sections answer two different questions.

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   src/login.js

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        notes.txt
```

- **Changes to be committed** = differences between the *index* and *HEAD*. These are staged and will be in your next commit.
- **Changes not staged for commit** = differences between the *working directory* and the *index*. Real edits that Git has not yet been told to record.
- **Untracked files** = files in the working directory that Git has never seen.

> **Tip:** A single file can appear in *both* "to be committed" and "not staged" at once. That means you staged it, then edited it again — the index holds the earlier version, the working dir the newer one.

Two diffs for two gaps

Because there are three trees, the diff you want depends on which gap you are inspecting.

COMMAND	COMPARES	ANSWERS
`git diff`	Working dir vs index	What have I changed but not yet staged?
`git diff --staged` (`--cached`)	Index vs HEAD	What will go into my next commit?
`git diff HEAD`	Working dir vs HEAD	Everything changed since the last commit, staged or not.

```
# Unstaged edits only
git diff

# Exactly what your commit will capture
git diff --staged

# Total change since last commit
git diff HEAD
```

> **Note:** `--staged` and `--cached` are synonyms. "Staged" reads better in conversation; "cached" is the older name and still works everywhere.

Partial staging with `add -p`

The `git add -p` command (short for *patch*) is one of the most powerful tools in Git. It lets you interactively review and stage your code changes line-by-line or block-by-block (called *hunks*), rather than staging an entire file at once.

It's great for splitting a big chunk of work into clean, logical commits.

How it works

When you run `git add -p`, Git presents you with a hunk of changes and asks what you want to do with it, using a prompt that looks like this:

```
# Walk through each change interactively
git add -p
```

```
diff --git a/src/login.js b/src/login.js
@@ -12,6 +12,7 @@ function redirect(user) {
   if (user.isAdmin) {
+    log("admin path");
     return "/dashboard";
   }
Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]?
```

The hunk keys

That list of letters looks intimidating, but you only need a few to be highly effective. The everyday keys are `y`, `n`, `q`, `s`, and `e`; the rest just move you between hunks or open help.

KEY	ACTION
`y`	Stage this hunk
`n`	Do not stage this hunk
`q`	Quit; do not stage this or any remaining hunk
`a`	Stage this and all later hunks in the file
`d`	Do not stage this or any later hunk in the file
`j`	Leave this hunk undecided, go to the next undecided hunk
`J`	Leave this hunk undecided, go to the next hunk
`g`	Jump to a hunk you choose from a list
`/`	Search for a hunk matching a regular expression
`s`	Split the hunk into smaller pieces
`e`	Edit the hunk manually before staging
`?`	Show help for these keys

Splitting and editing

If a hunk bundles two unrelated changes that happen to be near each other, press `s` to split it into smaller hunks. When even that is too coarse — for example two changes on adjacent lines — press `e` to open the hunk in your editor. There you delete lines you do not want by removing `+` lines, or turn unwanted `-` removals back into context lines by changing the leading `-` to a space.

> **Warning:** When editing a hunk by hand (`e`), never change the actual content of a line — only choose whether a line is included. A malformed hunk will simply fail to apply and Git will discard it.

Why use it?

- **Cleaner commit history.** If you fixed a bug but left a stray `console.log` or a formatting tweak in the same file, stage the fix and leave the junk out.
- **Built-in code review.** It forces you to look at your own diffs before committing — a great self-review safety net.

- **Context switching.** If you worked on two different features in the same file, you can separate them into two distinct commits.

A quick example

Imagine you added two functions to `app.js`. Running `git add -p` shows you the first function — you type `y`. It shows you the second — you type `n`. Now if you run `git status`, you'll see `app.js` listed under both "Changes to be committed" and "Changes not staged for commit". You're ready to make a highly targeted commit.

The same idea in reverse: `restore -p`

Patch mode is not just for staging. `git restore -p` selectively *discards* working-tree changes hunk by hunk, and `git restore --staged -p` selectively unstages.

```
# Selectively throw away unstaged edits (careful!)
git restore -p src/login.js

# Selectively unstage hunks back into the working tree
git restore --staged -p src/login.js
```

> **Key idea:** Atomic commits — one logical change each — make history bisectable, reviewable, and revertible. `add -p` is the everyday tool that makes atomic commits practical even when your working tree is messy.

Reading diffs and logs like a pro

Inspecting history well is what separates someone who *uses* Git from someone who *maintains* a project with it.

Diff flavors worth knowing

```
# Word-level highlighting instead of whole-line
git diff --word-diff

# Just the names of changed files
git diff --name-only

# A summary of added/removed lines per file
git diff --stat

# Compare two arbitrary commits or branches
git diff main..feature
```

A readable graph of history

The classic one-liner every maintainer keeps close. It shows a compact, decorated, branching view across all refs.

```
git log --oneline --graph --decorate --all
```

```
* 9f3a1c2 (HEAD → main, origin/main) Fix login redirect
* 1b7e004 Add admin dashboard route
| * 3c9d8e1 (feature/search) Wire up search box
|/
* a04f2b9 Initial commit
```

- `--oneline` — one compact line per commit.
- `--graph` — draw the branch/merge topology on the left.
- `--decorate` — show branch and tag names (on by default in modern Git).
- `--all` — include every branch, not just the current one.

Custom log formatting

Use `--pretty` or the more flexible `--format` with placeholders to shape output for scripts or scanning.

```
# Hash, author, relative date, subject
git log --pretty=format:"%h %an %ar %s"
```

```
9f3a1c2 Milad 2 hours ago Fix login redirect
1b7e004 Milad 5 hours ago Add admin dashboard route
```

Common placeholders: `%H` / `%h` (full/short hash), `%an` (author name), `%ad` / `%ar` (date/relative date), `%s` (subject), `%d` (ref names).

Filtering history

```
# Commits by a particular author
git log --author="Milad"

# Commits in a time window
git log --since="2 weeks ago" --until="yesterday"

# The "pickaxe": commits that added or removed a string
git log -S "redirect"

# Commits whose diff matches a regex
git log -G "function redirect"

# Limit history to one path (note the -- separator)
git log --oneline -- src/login.js
```

> **Tip:** The `--` separator tells Git "everything after this is a path, not a branch." Use it whenever a filename could be mistaken for a ref.

Inspecting a single commit with `git show`

```
# Full message plus the diff for one commit
git show 9f3a1c2

# Just one file as it appeared in that commit
git show 9f3a1c2:src/login.js
```

Line-by-line history with `git blame`

`git blame` annotates each line of a file with the commit, author, and date that last touched it — the starting point for "why is this line here?"

```
# Annotate every line of a file
git blame src/login.js

# Only lines 10-20, and follow code moved from other files
git blame -L 10,20 -M src/login.js
```

```
9f3a1c2 (Milad 2026-06-25 12:04 12)     log("admin path");
1b7e004 (Milad 2026-06-25 09:30 13)     return "/dashboard";
```

Example: Found a suspicious line? Run `git blame -L` on it to get the commit hash, then `git show <hash>` to read the full change and its message in context — blame tells you *who/when*, show tells you *why*.

Recap: Git revolves around three trees — working directory, index, and HEAD. `git add` moves changes working → index; `git commit` moves index → HEAD. `git diff` shows the working-vs-index gap, `git diff --staged` the index-vs-HEAD gap. Use `git add -p` and `git restore -p` to stage and discard surgically for atomic commits. Inspect history with `git log --oneline --graph --decorate --all`, shape and filter it with `--format`, `--author`, `-S`, and `-- path`, and drill into specifics with `git show` and `git blame`.

Branching Strategies in Practice

Branches are the cheapest, most powerful tool Git gives you. But they are also the source of most day-to-day confusion: stale remotes, "up to date" messages that lie, and branches that refuse to delete. This chapter focuses on the mechanics you use every day — creating and renaming branches, wiring them to upstreams, and building an accurate mental model of how your local branches relate to what lives on the server.

Creating, switching, deleting, renaming

A branch in Git is nothing more than a movable pointer to a commit. Creating one is instant and costs almost no disk space, so branch freely. Modern Git separates the two jobs that the old `git checkout` overloaded: `git switch` moves between branches, and `git restore` restores files. Both `switch` and `restore` have been stable since Git 2.23 and are the recommended commands today, though `git checkout` still works exactly as before.

Creating and switching

```
# Create a new branch and switch to it in one step
git switch -c feature/login

# Equivalent older syntax (still valid)
git checkout -b feature/login

# Create a branch without switching to it
git branch feature/login

# Switch to an existing branch
git switch main

# Create a branch starting from a specific commit or another branch
git switch -c hotfix/typo origin/main
```

> **Tip:** `git switch -` jumps back to the branch you were previously on, the same way `cd -` returns to your previous directory. It is great for ping-ponging between a feature branch and `main`.

Deleting: `-d` versus `-D`

Deleting a branch only removes the pointer; the commits themselves survive until garbage collection if anything still references them. Git protects you from losing work with a *safe delete*.

```
# Safe delete: refuses if the branch has unmerged commits
git branch -d feature/login

# Force delete: removes the pointer even if commits are not merged
git branch -D feature/login
```

```
error: the branch 'feature/login' is not fully merged.
If you are sure you want to delete it, run 'git branch -D feature/login'.
```

> **Warning:** `git branch -D` can orphan commits. They are not gone immediately — you can usually recover them via `git reflog`. But once `git gc` runs (or after the reflog expiry, typically 30–90 days) they are truly lost. Only force-delete when you are certain.

Renaming

```
# Rename the branch you are currently on
git branch -m new-name

# Rename a specific branch without checking it out
git branch -m old-name new-name
```

Renaming a local branch does **not** rename it on the remote. After renaming, you typically delete the old remote branch and push the new one (covered below).

Deleting a remote branch

```
# Delete a branch on the remote named 'origin'
git push origin --delete feature/login

# Older equivalent: push an empty ref to the branch
git push origin :feature/login
```

Listing branches

```
# List local branches (current one marked with *)
git branch

# List local branches with their upstream and tracking status
git branch -vv

# List remote-tracking branches
git branch -r

# List everything, local and remote
git branch -a
```

```
* feature/login  9af21cd [origin/feature/login: ahead 2] add session cookie
  main           1b3d7e0 [origin/main] initial commit
```

The `-vv` output is dense but invaluable: the SHA, the upstream in square brackets, the ahead/behind status, and the subject of the last commit. If a branch shows no bracketed upstream, it is not tracking anything — the next section explains why that matters.

Tracking branches and upstreams

An *upstream* (also called a tracking relationship) is a stored association between your local branch and a branch on a remote. Once set, Git knows where `git push` and `git pull` should go without you naming the remote and branch every time. It can also report how far ahead or behind you are.

Setting an upstream

```
# Push and set 'origin/feature/login' as the upstream in one step
git push -u origin feature/login

# Set or change the upstream for the current branch explicitly
git branch --set-upstream-to=origin/feature/login

# Shorthand for the current branch
git branch -u origin/feature/login
```

> **Tip:** The first time you push a new branch, always use `-u` (short for `--set-upstream`). After that, a bare `git push` and `git pull` will just work for that branch.

How status shows "ahead" and "behind"

"Ahead" and "behind" are measured by comparing your local branch against its *remote-tracking ref* (for example `origin/main`), not against the live server. Ahead means you have commits the remote-tracking ref does not; behind means the remote-tracking ref has commits you do not.

```
$ git status
On branch feature/login
Your branch is ahead of 'origin/feature/login' by 2 commits.
  (use "git push" to publish your local commits)
```

STATUS	MEANING	USUAL ACTION
ahead N	You have N commits not yet pushed	`git push`
behind N	The remote-tracking ref has N commits you lack	`git pull` (or merge/rebase)
ahead N, behind M	Both branches have diverged	Integrate, then push
up to date	Matches the remote-tracking ref	Nothing — but see below

`git fetch` versus `git pull`

This distinction is the single most useful thing to understand about remotes.

- **`git fetch`** downloads new commits and updates your remote-tracking refs (`origin/*`). It never touches your working branch — it is completely safe and changes nothing you are editing.
- **`git pull`** is `git fetch` followed by an integration step (`git merge` by default, or `git rebase` if configured). It *does* change your current branch.

```
# Just update knowledge of the remote; safe, no working changes
git fetch origin

# Fetch and then merge the upstream into the current branch
git pull

# Fetch and rebase your local commits on top of the upstream
git pull --rebase
```

> **Key idea:** `git pull` = `git fetch` + integrate. When unsure what the server has, run `git fetch` first, inspect with `git log main..origin/main`, and only then decide whether to merge or rebase. Reaching for `fetch` instead of `pull` avoids surprise merge commits and unexpected conflicts.

Local vs remote branch mental model

Confusion about remotes almost always comes from mixing up three different things that happen to share a name. Pin these down and most "why is Git lying to me" moments go away.

The three references

Local branch (`main`)
: A pointer you move by committing. This is what your working tree reflects.

Remote-tracking ref (`origin/main`)
: A read-only local cache of where `main` was on the remote the *last time you fetched*. You cannot commit onto it directly; Git updates it for you.

The actual branch on the server
: The real `main` living in the remote repository. Your machine has no live view of it — only the snapshot captured by your last fetch.

> **Note:** `origin/main` is not the server. It is your local *memory* of the server. It only changes when you run `git fetch` or `git pull`. Between fetches it can be arbitrarily out of date, no matter what teammates have pushed.

How fetch updates remote-tracking refs

```
# Before fetch: origin/main is whatever you last saw
git log --oneline origin/main -1

# Fetch advances origin/main to match the server right now
git fetch origin

# Now compare your local main against the freshly updated cache
git log --oneline main..origin/main
```

The range `main..origin/main` lists commits that exist on the remote but not in your local `main` — exactly the commits a `pull` would bring in. Reversing it, `origin/main..main`, shows what you would push.

Why "your branch is up to date" can be misleading

When `git status` says *"Your branch is up to date with 'origin/main'"*, it is comparing your local `main` to the remote-tracking ref `origin/main` — the cached snapshot. It is **not** contacting the server. If a teammate pushed five minutes ago and you have not fetched, the message is technically true and practically wrong.

> **Example:** You run `git status` and see "up to date," then run `git fetch` and suddenly status reports "behind by 3 commits." Nothing broke — the first message simply compared against a stale cache. Always fetch before trusting an "up to date."

Pruning stale remote-tracking refs

When someone deletes a branch on the server, your local `origin/that-branch` ref lingers until you prune it. Over time these stale refs clutter `git branch -r` and tab-completion.

```
# Fetch and remove remote-tracking refs whose branch no longer exists
git fetch --prune

# Prune without fetching anything new
git remote prune origin

# Make every fetch prune automatically (recommended, set once)
git config --global fetch.prune true
```

```
From github.com:acme/app
 - [deleted]         (none)     -> origin/feature/old-experiment
```

> **Tip:** Setting `fetch.prune true` globally means `origin/*` always mirrors reality after a fetch. Pruning the remote-tracking ref does not delete your *local* branch of the same name — you remove that separately with `git branch -d`.

Recap

- Use `git switch -c <name>` to create-and-switch; `checkout -b` still works. Rename with `git branch -m`.
- `git branch -d` is the safe delete; `-D` forces it and can orphan unmerged commits — recoverable via reflog only until `gc`.

- Delete a server branch with `git push origin --delete <name>`; inspect tracking with `git branch -vv`.
- An upstream wires a local branch to a remote one; set it with `git push -u` or `git branch --set-upstream-to`. Ahead/behind is measured against the cached `origin/*` ref.
- `git fetch` only updates remote-tracking refs and is always safe; `git pull` = fetch + integrate and changes your branch.
- `origin/main` is your local memory of the server, not the server itself — so "up to date" can be stale until you fetch. Keep it honest with `git fetch --prune` or `fetch.prune true`.

PART 03

Merge, Rebase & Cherry-Pick

Every realistic scenario for combining and moving work, with conflict resolution walkthroughs.

Merge: All the Scenarios

Merging is where branches come back together — and where a surprising amount of day-to-day Git confusion lives. This chapter walks through every flavor of merge you will meet in practice: the trivial fast-forward, the genuine three-way merge, the conflict you have to resolve by hand, and the merge you wish you had never made. For each one you get the situation, the exact commands, what Git actually does under the hood, and how to recover when it goes sideways.

Fast-forward vs true merge

When you run `git merge feature`, Git first asks a simple question: *has the current branch moved since `feature` branched off?* If it hasn't — if `main` is a direct **ancestor** of `feature` — then there is nothing to combine. Git can simply slide the `main` pointer forward to `feature`'s commit. That is a **fast-forward**.

```
# main has not moved since feature branched; merge fast-forwards
git switch main
git merge feature
```

```
Updating a1b2c3d..f6e7d89
Fast-forward
 app.py | 12 ++++++++++++
 1 file changed, 12 insertions(+)
```

No new commit is created. The history stays perfectly linear — it looks as if you had committed directly on `main` all along.

A **true merge** (also called a three-way or non-fast-forward merge) happens when *both* branches have new commits. Git cannot just move a pointer, because that would throw away one side's work. Instead it creates a **merge commit**: a special commit with *two parents*, one for each branch being joined. That commit records the combined tree.

```
# both main and feature advanced → Git must create a merge commit
git switch main
git merge feature
```

```
Merge made by the 'ort' strategy.
 app.py   | 12 ++++++++++++
 utils.py |  3 +++
 2 files changed, 15 insertions(+)
```

Seeing the shape with git log --graph

The fastest way to understand what a merge did is to look at the commit graph.

```
# draw the DAG as ASCII art
git log --oneline --graph --all
```

```
*   9d3a1f2 (HEAD -> main) Merge branch 'feature'
|\
| * 4c8e7b1 (feature) Add retry logic
| * 1a2b3c4 Add helper
* | 7f0d5e6 Fix typo in README
|/
* a1b2c3d Initial commit
```

The two diverging lines (`|\`) that rejoin at the top (`|/`) are the visual signature of a true merge. A fast-forward, by contrast, shows a single straight column with no fork.

> **Note:** The default merge strategy in modern Git (2.34+) is `ort` ("Ostensibly Recursive's Twin"), which replaced the older `recursive` strategy. It is faster and handles renames more accurately, but the conceptual model below is unchanged.

Three-way merges explained

The "three" in three-way merge refers to three commits Git looks at:

1. The tip of your current branch — **ours** (`HEAD`).
2. The tip of the branch being merged in — **theirs**.
3. The **merge base** — the most recent common ancestor of the two tips.

```
# show the merge base of two branches
git merge-base main feature
```

```
a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0
```

For every file, Git compares the merge base against both sides:

Changed on only one side
: Take that side's version. No conflict — Git knows the other side is unchanged from the base, so the change is unambiguous.

Changed identically on both sides
: Take either version; they agree.

Changed differently on both sides, in overlapping regions
: This is a **conflict**. Git cannot decide and asks you.

Because Git diffs against the common ancestor rather than blindly comparing the two tips, it can auto-merge a huge amount of work. Two people editing different functions in the same file usually merge cleanly — the merge base tells Git which lines each person *actually* touched.

> **Key idea:** A merge is not "line A vs line B." It is "how did each side change relative to where they agreed?" The merge base is what makes Git smarter than a naive diff.

--no-ff, --squash, and --ff-only

Even when a fast-forward is *possible*, you often want to control whether one happens. These three flags give you that control.

OPTION	CREATES MERGE COMMIT?	HISTORY SHAPE	KEEPS FEATURE COMMITS?	TYPICAL USE
`--ff` (default)	Only if needed	Linear when possible	Yes	Quick local integration
`--no-ff`	Always	Explicit fork & join	Yes	Preserve that a feature existed
`--squash`	No (one new commit, single parent)	Linear	No — collapsed into one	Tidy a messy branch into one commit
`--ff-only`	Never	Linear only	Yes	Refuse if a real merge is required

--no-ff: always record the merge

```
# force a merge commit even when a fast-forward was possible
git merge --no-ff feature
```

This is the standard for many teams. Every feature branch leaves a visible "merge bubble" in history, so you can see exactly which commits belonged to which piece of work — and revert the whole feature with a single command later.

--squash: collapse into one commit

```
# stage the combined result but do NOT commit or record feature as a parent
git merge --squash feature
git commit -m "Add retry logic"
```

Squash does not create a merge commit and does not list `feature` as a parent. The branch's individual commits are flattened into your staging area; you write one new commit. Git will *not* consider `feature` merged afterward — if you re-merge it, those changes look new.

--ff-only: refuse a real merge

```
# succeed only if it can fast-forward; otherwise fail and do nothing
git merge --ff-only feature
```

```
fatal: Not possible to fast-forward, aborting.
```

This is a safety guard: it guarantees you never accidentally create a merge commit. You can make it the default with config:

```
# per-repo: only fast-forward merges allowed by default
git config merge.ff only

# always create a merge commit instead (opposite policy)
git config merge.ff false
```

> **Tip:** Set `git config --global pull.ff only` to make `git pull` refuse to create surprise merge commits when your local branch has diverged from the remote.

Scenario: simple clean merge

Situation: You finished `feature/login`, it touches files that `main` has not changed, and you want it on `main`.

```
# make sure main is current, then merge
git switch main
git pull
git merge --no-ff feature/login
```

```
Merge made by the 'ort' strategy.
 auth/login.py       | 40 ++++++++++++++++++++++++++++++++++++++++
 tests/test_login.py | 18 ++++++++++++++++++
 2 files changed, 58 insertions(+)
```

What happens: Git computed the merge base, saw no overlapping edits, and produced a merge commit automatically. **Cleanup:** once merged and pushed, delete the branch.

```
git push
git branch -d feature/login   # safe delete: refuses if not merged
```

Scenario: merge with conflicts

Situation: Both `main` and `feature` edited the same lines of `config.py`. Git cannot auto-merge.

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

```
Auto-merging config.py
CONFLICT (content): Merge conflict in config.py
Automatic merge failed; fix conflicts and then commit the result.
```

Reading the markers

Git rewrites the conflicting region of `config.py` with three sections separated by conflict markers:

```
TIMEOUT = 30
<<<<<<< HEAD
RETRIES = 3
DEBUG = False
=======
RETRIES = 5
LOG_LEVEL = "info"
>>>>>>> feature
PORT = 8080
```

Everything between `<<<<<<< HEAD` and `=======` is **your** side (the branch you are on). Everything between `=======` and `>>>>>>> feature` is **their** side (the branch being merged). Your job is to produce the correct final text and delete all three marker lines.

Resolving

Edit the file to the intended result — here we keep the higher retry count and both new settings:

```
TIMEOUT = 30
RETRIES = 5
DEBUG = False
LOG_LEVEL = "info"
PORT = 8080
```

```
# mark it resolved, then complete the merge
git add config.py
git status            # confirm nothing else is unmerged
git commit            # opens editor with a prepared merge message
```

Helpers when conflicts pile up

```
# launch a configured visual merge tool
git mergetool

# take ONE whole side without hand-editing
git checkout --ours config.py     # keep HEAD's version entirely
git checkout --theirs config.py   # keep feature's version entirely
git add config.py

# give up and return to the pre-merge state
git merge --abort
```

> **Warning:** `--ours` and `--theirs` replace the *entire file* with one side. They are perfect for generated lock files and binary assets, but dangerous for hand-written code where you usually want a blend of both sides. Inspect the result before `git add`.

> **Tip:** Run `git config merge.conflictStyle zdiff3` to add a third section to each conflict showing the *merge base*. Seeing the original lines makes it far easier to understand what each side actually changed.

Scenario: merging a stale long-lived branch

Situation: `feature/big-refactor` has lived for three months. `main` moved 400 commits ahead. A direct merge produces dozens of conflicts in one overwhelming pile.

Strategy 1: merge main into the feature first, incrementally

Do the conflict resolution *on the feature branch*, in small batches, before merging back. This keeps `main` clean and lets you resolve, test, and commit repeatedly.

```
git switch feature/big-refactor
git merge main          # resolve conflicts here, test, commit
# ... time passes, main moves again ...
git merge main          # again, smaller delta each time
```

When the feature finally merges into `main`, it fast-forwards or merges trivially, because `main` is already an ancestor of the feature's latest state.

Strategy 2: enable rerere

rerere ("reuse recorded resolution") remembers how you resolved a given conflict. It replays that resolution automatically the next time the identical conflict appears — invaluable when you merge `main` in repeatedly.

```
# turn on once, globally
git config --global rerere.enabled true
```

After this, the first manual resolution is recorded; subsequent identical conflicts are resolved for you (you still `git add` to confirm).

Strategy 3: resolve incrementally with smaller scope

If even one merge is too big, attack it file by file:

```
# list only the unmerged paths
git diff --name-only --diff-filter=U

# resolve, add, and re-check after each file
git add path/to/resolved_file.py
git diff --name-only --diff-filter=U   # shrinking list
```

> **Note:** The real lesson of this scenario is to *avoid creating it*. Merge or rebase `main` into long-lived branches frequently — weekly, not quarterly — so each integration is small.

Scenario: merge then "undo the merge"

How you undo a merge depends entirely on one thing: **have you pushed it?**

Not yet pushed: reset --hard ORIG_HEAD

Whenever a merge moves `HEAD`, Git saves the previous position in `ORIG_HEAD`. If the merge is still local, throw it away cleanly:

```
# undo the merge you just made, restoring the pre-merge tip
git reset --hard ORIG_HEAD
```

What happens: `main` moves back to exactly where it was before the merge, and your working tree matches it. The merge commit is gone from this branch as if it never existed.

> **Warning:** `reset --hard` discards uncommitted changes in your working tree too. Stash or commit anything you want to keep first.

Already pushed: revert -m 1

Once others may have pulled the merge, you must *not* rewrite history. Instead, create a *new* commit that inverts the merge's changes.

```
# revert a merge commit; -m 1 means "keep parent #1, undo parent #2's changes"
git revert -m 1 9d3a1f2
```

A merge commit has two parents, so `git revert` needs to know which parent is the "mainline" to keep. `-m 1` is almost always correct: parent 1 is the branch you merged *into* (e.g. `main`), parent 2 is the branch you merged *in* (e.g. `feature`). This undoes the feature's changes while keeping main's history intact.

ASPECT	`RESET --HARD ORIG_HEAD`	`REVERT -M 1`
Rewrites history?	Yes	No
Safe after push?	No	Yes
Merge commit remains?	No — erased	Yes, plus a revert commit
Re-merging later works cleanly?	Yes	No — see warning

> **Warning:** After reverting a merge, the branch is *still considered merged* by Git. If you later fix the feature and try to merge it again, Git brings in nothing new — because the original merge is still in history. You must revert the revert (`git revert <revert-commit>`) before re-merging. This is a classic, confusing trap.

Scenario: accidental merge of the wrong branch

Situation: You meant to merge `feature/login` but typed `git merge feature/logout` and the merge succeeded, pulling in work that does not belong on `main`.

Detect it

```
# what did that last merge actually pull in?
git show --stat HEAD
git log --oneline --graph -5
```

Check the merge commit message and the second parent. If it names the wrong branch, you have your culprit.

Back out safely

```
# if NOT pushed: just reset to before the merge
git reset --hard ORIG_HEAD
```

If `ORIG_HEAD` has since been overwritten by another operation, find the pre-merge commit in the reflog instead — the reflog records every position `HEAD` has held:

```
git reflog
# locate the line just BEFORE "merge feature/logout"
```

```
9d3a1f2 (HEAD -> main) HEAD@{0}: merge feature/logout: Merge made by the 'ort'
strategy.
7f0d5e6 HEAD@{1}: commit: Fix typo in README
```

```
# reset to the commit before the bad merge
git reset --hard HEAD@{1}
```

If the bad merge was already pushed, fall back to the safe path from the previous scenario: `git revert -m 1 <merge-sha>`.

> **Tip:** The reflog is your undo history for almost everything in Git. Before any scary recovery, run `git reflog` first — the commit you think you lost is usually still there for about 90 days.

Merge commits vs linear history

Teams broadly choose between two history styles, and the choice is a real trade-off rather than a matter of taste.

CONCERN	MERGE COMMITS (`--NO-FF`)	LINEAR HISTORY (REBASE / SQUASH)
Accuracy	Records exactly what happened, when	Idealized, rewritten timeline
Readability of `git log`	Noisier; many merge bubbles	Clean, one line per change
Reverting a feature	One `revert -m 1`	May need multiple reverts
`git bisect`	Can land on merge commits	Every commit is a clean step
Grouping of work	Feature boundaries preserved	Boundaries lost after squash

Linear history reads beautifully and bisects cleanly. But it requires rewriting commits (rebasing), which is risky on shared branches and erases the true sequence of events. Merge-commit history is honest and easy to revert wholesale, at the cost of a busier graph.

> **Example policy:** Many teams settle on a hybrid — squash-merge each pull request so `main` gets one clean commit per feature (linear, easy to bisect), while developers rebase or merge freely on their own branches. The platform (GitHub/GitLab) enforces it via "Squash and merge" as the only allowed button.

Whatever you pick, **write it down and enforce it in config and CI**. The worst history is the inconsistent one, where half the branches were squashed and half were merged, and nobody can tell what the `git log` is telling them.

Recap:

- A **fast-forward** just moves a pointer; a **true merge** creates a two-parent merge commit. `git log --graph` shows which you got.
- Three-way merges diff both sides against the **merge base** — that is why Git auto-merges so much.
- `--no-ff` always records the merge, `--squash` flattens to one commit (no merge parent), `--ff-only` refuses real merges. Configure with `merge.ff`.
- Conflicts are bracketed by `<<<<<<<`, `=======`, `>>>>>>>`; edit, `git add`, `git commit`. `git merge --abort` backs out cleanly.
- Tame stale branches with frequent integration, `rerere`, and incremental resolution.
- Undo a merge with `reset --hard ORIG_HEAD` *before* pushing, and `revert -m 1` *after* — and remember a reverted merge will not re-merge until you revert the revert.
- The **reflog** is your safety net for any accidental merge.

Rebase: All the Scenarios

Rebase is the power tool of Git history editing. Used well, it produces a clean, linear, reviewable story of how your code came to be. Used carelessly, it rewrites commits other people depend on and turns a shared branch into a tangle of duplicates. This chapter walks through every rebase scenario you are likely to hit as a developer or maintainer — from a routine "catch up with main" to splitting a single commit into many, surviving conflicts, transplanting branches with `--onto`, and recovering when a rebase goes wrong.

What rebase really does (replaying commits)

A rebase takes a series of commits and *replays* them, one at a time, on top of a different base commit. The verb is literal: you are giving your commits a new **base**. For each commit on your branch that is not already on the target, Git computes the change that commit introduced (its diff). Then it applies that change on top of the new base, producing a *brand-new commit*.

The crucial consequence: **rebasing creates new commits with new hashes**. A commit's SHA-1 is derived from its content, its tree, its author and committer metadata, and — critically — its parent. Because the parent changes during a rebase, every replayed commit gets a new identity. The old commits are not modified; they are abandoned (and eventually garbage-collected), and new ones take their place.

```
                # Before rebase: feature branched off an old main
                              # main:    A---B---C
                                #               \
                             # feature:          D---E

                            # After: git rebase main
                              # main:    A---B---C
                             #                    \
          # feature:             D'---E'   (D' and E' are NEW commits)
```

Rebase moves the base of `feature` from `B` to `C`, replaying `D` and `E` as `D'` and `E'`.

Contrast with merge

A merge *preserves* history exactly as it happened and ties two lines together with a new merge commit that has two parents. A rebase *rewrites* history to look as if you had started your work from the latest base all along. The result is a straight line with no merge commit.

ASPECT	MERGE	REBASE
History shape	Non-linear; merge commits with two parents	Linear; single chain of commits
Commit hashes	Preserved	Rewritten (new hashes)
Original commits	Kept	Replaced
Records "when it happened"	Yes, faithfully	No; pretends work started from the new base
Safe on shared branches	Yes	No (see the golden rule)

> **Note:** Neither approach is "better" in the abstract. Merge is honest about history; rebase is clean about it. Many teams rebase feature branches locally to keep them tidy, then merge them into the trunk to record the integration point.

Scenario: rebase feature onto updated main

Situation: You started `feature` from `main` a few days ago. Meanwhile teammates pushed several commits to `main`. Before opening (or updating) your pull request, you want your branch to sit cleanly on top of the latest `main`. That keeps the diff minimal and conflict-free at merge time.

```
# 1. Update your local view of the remote
git fetch origin

# 2. Replay your feature commits on top of the latest main
git switch feature
git rebase origin/main

# 3. If your branch was already pushed, update the remote safely
git push --force-with-lease
```

Resolution: Your feature commits now sit on top of `origin/main`. Because rebasing rewrote their hashes, a plain `git push` would be rejected (the remote branch's tip is no longer an ancestor of yours). You must force-push — but use `--force-with-lease`, never bare `--force`. The `--force-with-lease` form checks that the remote branch still points where you last saw it. If a teammate pushed in the meantime, the push is refused instead of silently clobbering their work.

> **Tip:** Set `git config --global pull.rebase true` so `git pull` rebases your local commits onto the fetched upstream instead of creating a merge commit. Combine with `git config --global rebase.autoStash true` to auto-stash and re-apply uncommitted changes around the rebase.

Interactive rebase: reorder, squash, fixup, edit, drop

Interactive rebase (`-i`) is where rebase becomes a history editor. Instead of blindly replaying commits, Git opens an editor with a *todo list* — one line per commit, oldest at the top. It replays them according to the commands you write.

```
# Edit the last 4 commits interactively
git rebase -i HEAD~4
```

The editor opens with something like this. Edit the leftmost keyword on each line, save, and close:

```
pick a1b2c3d Add user model
pick e4f5g6h Fix typo in user model
pick i7j8k9l Add user controller
pick m0n1o2p WIP debugging

# Rebase 9z8y7x6..m0n1o2p onto 9z8y7x6 (4 commands)
#
# Commands:
# p, pick   = use commit
# r, reword = use commit, but edit the commit message
# e, edit   = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup  = like squash, but discard this commit's log message
# d, drop   = remove commit
# (lines may be re-ordered; they are executed from top to bottom)
```

Commits execute **top to bottom**, which is the opposite of `git log` ordering. You can reorder lines freely, change keywords, and delete lines (same as `drop`).

COMMAND	SHORT	KEEPS CHANGES?	KEEPS MESSAGE?	EFFECT
pick	p	Yes	Yes	Use the commit as-is
reword	r	Yes	Edits it	Same changes, new commit message
edit	e	Yes	Yes	Stop here so you can amend or split the commit
squash	s	Yes	Combines	Meld into the previous commit; you edit the combined message
fixup	f	Yes	Discards	Meld into previous, throw this commit's message away
drop	d	No	No	Remove the commit entirely

Reordering and squashing

To fold "Fix typo in user model" into "Add user model", move its line directly below the target and change `pick` to `fixup`:

```
pick a1b2c3d Add user model
fixup e4f5g6h Fix typo in user model
pick i7j8k9l Add user controller
drop m0n1o2p WIP debugging
```

--autosquash with commit --fixup

Manually shuffling lines is error-prone. Git can do it for you. When you make a follow-up change to an earlier commit, mark it:

```
# Commit a fix targeted at an earlier commit by its hash
git commit --fixup a1b2c3d

# Or create a squash commit (lets you edit the combined message)
git commit --squash a1b2c3d

# Later, rebase with autosquash: Git pre-arranges the todo list
git rebase -i --autosquash HEAD~6
```

The fixup commit's message is prefixed with `fixup!`. Then `--autosquash` automatically moves it under its target and sets the keyword to `fixup`. Set `git config --global rebase.autosquash true` to make `--autosquash` the default for every interactive rebase.

Scenario: cleaning up messy commits before a PR

Situation: Your branch has ten commits like "wip", "fix test", "oops", "actually fix it". Reviewers should see a handful of coherent, well-described commits — not your stream-of-consciousness.

```
# See how many commits diverge from main
git log --oneline origin/main..HEAD

# Open them all for editing
git rebase -i origin/main
```

In the todo list, keep the commits that represent logical units as `pick` (or `reword` them with better messages) and collapse the noise into them with `fixup` / `squash`:

```
reword 11aa22b Implement rate limiter
fixup  33cc44d wip
fixup  55ee66f fix test
reword 77gg88h Add rate-limiter middleware
fixup  99ii00j oops
```

Resolution: Two clean commits with clear messages. Push with `git push --force-with-lease`. Reviewers now read a curated history instead of archaeology.

> **Example:** A 12-commit branch full of "wip" entries becomes `Add rate limiter` + `Wire middleware into router` + `Add rate-limiter tests`. Same code, dramatically better review experience.

Scenario: splitting one commit into many

Situation: One commit does too much — it adds a feature *and* reformats an unrelated file. You want to split it into two focused commits.

```
# Start an interactive rebase that includes the fat commit
git rebase -i HEAD~3
```

Mark the offending commit with `edit`:

```
pick aaa111 Earlier commit
edit bbb222 Add export feature and reformat config
pick ccc333 Later commit
```

The rebase stops with that commit applied. Now undo just the commit (keeping its changes in the working tree as unstaged), then re-commit in pieces:

```
# Undo the commit but keep its changes unstaged
git reset HEAD^

# Stage and commit the first logical chunk
git add src/export.js
git commit -m "Add CSV export feature"

# Stage and commit the rest
git add config/app.yml
git commit -m "Reformat app config"

# Continue the rebase
git rebase --continue
```

> **Tip:** Use `git add -p` (patch mode) to stage individual hunks within a single file. This lets you split a commit even when both changes live in the same file.

Resolution: One commit became two atomic ones. The rest of the branch replays on top of the new pair.

Scenario: rebase conflicts step by step

Situation: While replaying a commit, Git finds that the same lines were changed both on the new base and in your commit. It pauses and asks you to resolve.

```
Auto-merging src/auth.js
CONFLICT (content): Merge conflict in src/auth.js
error: could not apply 7d3f1a2... Add OAuth provider
Resolve all conflicts manually, mark them as resolved with
"git add/rm <pathspec>", then run "git rebase --continue".
```

Open the conflicted file. Git marks the clash with conflict markers — your replayed change sits between `<<<<<<<` and `=======`, and the version already on the new base sits between `=======` and `>>>>>>>`:

```
<<<<<<< HEAD
const providers = ["github", "gitlab"];
=======
const providers = ["github", "google"];
>>>>>>> 7d3f1a2 (Add OAuth provider)
```

> **Warning:** During a rebase the meaning of "HEAD" feels inverted compared to a merge. `HEAD` (the top section) is the new base — the code you are replaying *onto* — while the bottom section is *your* commit being replayed. Read the labels, not your intuition.

Edit the file to the correct final state, remove all three markers, then mark it resolved and continue:

```
# After editing src/auth.js to the desired content
git add src/auth.js

# Resume replaying the remaining commits
git rebase --continue
```

You have three controls at any pause:

COMMAND	MEANING
`git rebase --continue`	You resolved the conflict (and staged it); proceed to the next commit.
`git rebase --skip`	Drop the commit currently being applied entirely and move on.
`git rebase --abort`	Stop and restore the branch exactly as it was before the rebase started.

> **Note:** The same conflict can recur on several commits if they each touch the affected lines. Enable `git config --global rerere.enabled true` ("reuse recorded resolution"). Git then remembers how you resolved a given conflict and replays your resolution automatically next time.

`--onto` : transplanting branches

The `--onto` flag separates "which commits to replay" from "where to put them." Its full form is:

```
git rebase --onto <newbase> <upstream> <branch>
```

This means: take the commits in `<branch>` that come *after* `<upstream>`, and replay them on top of `<newbase>`. It is the tool for surgically moving a range of commits.

Concrete example: a feature built on the wrong branch

Situation: You branched `topic` off `feature`, but `feature` turned out to be a dead end. You want only *your* `topic` commits, replanted directly onto `main`.

```
# History:
# main:     A---B
#                \
# feature:        C---D
#                      \
# topic:                E---F   (E and F are yours)

# Move only E and F onto main, dropping C and D
git rebase --onto main feature topic
```

```
# Result:
# main:     A---B
#                \
# topic:          E'---F'   (cleanly on top of main; C, D gone)
```

Resolution: Read it as "replay `topic`'s commits that are after `feature`, onto `main`." The commits from `feature` (`C`, `D`) are excluded because they are at or before the `<upstream>` cutoff.

> **Example:** To drop the first commit of a branch entirely, replay everything after it onto its grandparent: `git rebase --onto branch~2 branch~1 branch`. This excises a single commit from the start of a chain.

The golden rule: never rebase shared/public history

> **Warning — the golden rule:** Never rebase commits that exist outside your local repository and that others may have based work on. Rebasing rewrites commits into new hashes; once others have your old commits, rewriting them desynchronizes everyone.

Why it breaks collaboration: suppose you push commits `D` and `E` to a shared branch, a teammate pulls them, and then you rebase and force-push `D'` and `E'`. Your teammate still has the original `D` and `E` in their history. When they next pull, Git sees two sets of commits with identical content but different hashes and tries to merge them. That produces duplicate commits and a tangled mess someone has to untangle by hand.

Practical boundaries:

- **Never** rebase `main`, `master`, `develop`, release branches, or any branch multiple people commit to.
- **Safe:** rebasing your own local commits that you have never pushed.
- **Safe exception:** rebasing and force-pushing your *own* PR/feature branch that only you work on. This is the everyday workflow for keeping a PR current. Always use `git push --force-with-lease` so you don't overwrite an unexpected push, and let collaborators know if anyone else has checked out your branch.

> **Key:** The test is simple — "Does anyone else depend on these exact commits?" If no, rebase freely. If yes, or you are unsure, prefer merge.

Scenario: recovering from a botched rebase

Situation: You ran a long interactive rebase, made wrong choices (dropped the wrong commit, resolved a conflict incorrectly, squashed too much), and already typed `git rebase --continue` to completion. The branch is now wrong, and `--abort` is no longer available because the rebase finished. Stay calm: the original commits still exist.

Recover with the reflog

Git records every position `HEAD` has occupied in the *reflog*. Your pre-rebase tip is still there:

```
git reflog
```

```
3f8a1c2 HEAD@{0}: rebase (finish): returning to refs/heads/feature
3f8a1c2 HEAD@{1}: rebase (pick): Add controller
9d2e4b7 HEAD@{2}: rebase (start): checkout origin/main
c5a7f01 HEAD@{3}: commit: Add controller
b1d9e34 HEAD@{4}: commit: Add model
a0c8f12 HEAD@{5}: commit: Initial feature work
```

The entry just before `rebase (start)` — here `HEAD@{3}`, hash `c5a7f01` — is your branch as it stood before the rebase. Reset to it:

```
# Move the branch back to its pre-rebase state, discarding the bad result
git reset --hard HEAD@{3}

# Equivalently, by hash:
git reset --hard c5a7f01
```

> **Warning:** `git reset --hard` discards uncommitted work in the working tree. Stash or commit anything you want to keep before running it.

The ORIG_HEAD shortcut

Before a rebase (or merge, or a hard reset) Git saves the previous `HEAD` into a ref called `ORIG_HEAD`. Right after a single botched rebase, this is the fastest undo:

```
# Restore the branch to where it pointed before the rebase
git reset --hard ORIG_HEAD
```

> **Tip:** `ORIG_HEAD` only remembers the *most recent* history-moving operation, so use it immediately. For anything older, walk the reflog — it keeps reachable entries for 90 days by default (30 for unreachable ones), so even "lost" commits are usually recoverable.

Resolution: Your branch is restored to its exact pre-rebase state. Nothing was truly lost — rebase abandons old commits rather than deleting them, and the reflog is your map back to them.

Recap

Key takeaways:

- Rebase *replays* commits onto a new base, producing **new hashes** — the original commits are abandoned, not edited.
- To catch up with the trunk: `git fetch`, `git rebase origin/main`, then `git push --force-with-lease`.
- Interactive rebase (`-i`) edits history via a todo list: `pick`, `reword`, `edit`, `squash`, `fixup`, `drop`. Pair `git commit --fixup` with `--autosquash` to automate cleanup.
- Split a commit with `edit` + `git reset HEAD^` + staged re-commits; combine messy commits with `fixup` / `squash`.
- On conflict: edit, `git add`, `git rebase --continue`; or `--skip` / `--abort`. Remember HEAD is the new base during a rebase.
- `git rebase --onto <newbase> <upstream> <branch>` transplants a precise range of commits.
- **Golden rule:** never rebase shared history. Force-pushing your own PR branch with `--force-with-lease` is the safe exception.
- When a rebase goes wrong, `git reflog` + `git reset --hard HEAD@{n}` (or `ORIG_HEAD`) brings you back.

Cherry-Pick: All the Scenarios

Cherry-pick is the scalpel of Git history rewriting. It lets you take exactly one commit (or a precise range of commits) from anywhere in your repository and replay it onto the branch you currently have checked out. It is indispensable for backporting fixes, rescuing work from abandoned branches, and surgically moving changes between long-lived release lines. It is also one of the easiest commands to misuse, producing duplicate commits and confusing history. This chapter walks through when cherry-pick is the right tool, the concrete scenarios developers and maintainers hit every week, the full mechanics of ranges and merge commits, conflict resolution, and how to avoid the duplicate-commit traps that bite teams who reach for it too eagerly.

When cherry-pick is the right tool

Cherry-pick copies the *change introduced by a single commit* and applies it as a new commit on your current branch. The keyword is *copies*: the result is a brand-new commit with a brand-new SHA-1 hash, even though the diff (the tree change) is identical to the source. The author, original message, and patch content are preserved. But the committer, commit date, and parentage all change, so the hash necessarily differs.

```
# Copy commit abc1234 onto whatever branch is currently checked out
git switch main
git cherry-pick abc1234
```

Reach for cherry-pick when you want **one specific commit**, not a whole line of development. Compare the three tools:

TOOL	WHAT IT MOVES	TYPICAL USE
`git merge`	An entire branch's history, joined with a merge commit	Integrating a completed feature; preserving the full branch shape
`git rebase`	A series of your commits, replayed onto a new base	Linearizing your branch on top of an updated upstream
`git cherry-pick`	One or a few hand-picked commits, copied as new commits	Backporting a hotfix; extracting a useful commit from elsewhere

> **Key:** If you want the *whole* branch, use merge or rebase. Cherry-pick is for the cases where you deliberately want *some* commits and not others, and you accept that the copies will have new hashes disconnected from the originals.

> **Note:** Because the hash changes, two branches that both contain "the same fix" via cherry-pick do not automatically know they share that change. Git tracks commits by hash, not by content. This is the root of every duplicate-commit problem later in this chapter.

Scenario: hotfix into multiple release branches

Situation. You maintain `main` plus several supported release branches: `release/2.0`, `release/2.1`, and `release/3.0`. A security fix lands on `main` as commit `f00dbabe`, and every supported release needs the same fix.

Commands. Cherry-pick the single fix onto each branch in turn. Use `-x` so each copy records which commit it came from.

```
# The fix is already on main as f00dbabe
git switch release/2.0
git cherry-pick -x f00dbabe

git switch release/2.1
git cherry-pick -x f00dbabe

git switch release/3.0
git cherry-pick -x f00dbabe
```

The `-x` flag appends a line to each new commit message:

```
(cherry picked from commit f00dbabe1234567890abcdef1234567890abcdef)
```

Resolution. Each release branch now carries its own copy of the fix with a traceable origin. Push each branch and tag a patch release as needed. If your fix spans more than one commit, give cherry-pick the list in order and Git applies them sequentially:

```
# Apply two related fix commits, oldest first, recording origins
git cherry-pick -x f00dbabe c0ffee01
```

> **Tip:** Always fix on the *oldest* still-supported branch first when possible, then merge forward (`release/2.0` → `release/2.1` → `release/3.0` → `main`). Forward-merging avoids the duplicate-commit problem entirely because each newer branch genuinely contains the older branch's commit. Cherry-pick into multiple branches is the right call only when the branches have diverged too far to merge cleanly.

Scenario: pulling one commit out of a feature branch

Situation. A teammate's `feature/dashboard` branch contains 14 commits. One of them, a small but important bug fix to a shared utility, needs to ship now. The rest of the feature is not ready and must not be merged.

Commands. First identify the commit, then copy only that one.

```
# Inspect the branch to find the commit you want
git log --oneline feature/dashboard

# Suppose the useful commit is 9a8b7c6
git switch main
git cherry-pick -x 9a8b7c6
```

```
9a8b7c6 Fix off-by-one in date formatter
4d3e2f1 WIP: dashboard layout
1122334 Add chart widget (incomplete)
```

Resolution. Only the date-formatter fix lands on `main`; the unfinished dashboard work stays behind. When `feature/dashboard` is eventually rebased or merged, Git's patch-detection (discussed below) will usually recognize that this commit is already present. It drops the commit during a rebase, avoiding a duplicate. If it merges instead, watch for the doubled change.

> **Example:** You only want part of a commit. Cherry-pick the whole commit with `-n` (no auto-commit), unstage the parts you do not want, and commit the rest:
>
> ```
> git cherry-pick -n 9a8b7c6        # stage the change, do not commit
> git restore --staged path/to/unwanted-file
> git commit -m "Backport only the formatter fix"
> ```

Cherry-picking ranges and merge commits

Cherry-pick is not limited to single commits. You can hand it a commit *range* and it will copy each commit in that range, oldest first.

Exclusive vs inclusive ranges

The two-dot syntax `A..B` means "every commit reachable from `B` but not from `A`". Crucially, this is **exclusive of `A` itself** — commit `A` is the lower bound and is *not* copied.

```
# Copy B and everything after A up to B, but NOT A
git cherry-pick A..B

# Include A as well by starting one commit earlier with A^
git cherry-pick A^..B
```

`A^` means "the parent of `A`", so `A^..B` spans from `A`'s parent (exclusive) up to `B` (inclusive) — which effectively includes `A` through `B`.

SYNTAX	INCLUDES A?	INCLUDES B?
`git cherry-pick A..B`	No	Yes
`git cherry-pick A^..B`	Yes	Yes

Useful flags

`-n` / `--no-commit`
: Apply the change to the working tree and index but do not create a commit. Combine several picks into one commit, or edit before committing.

`-x`
: Append `(cherry picked from commit <hash>)` to the message so the origin is traceable. Recommended for backports on shared branches.

`-e` / `--edit`
: Open the editor to amend the message before committing.

`-s` / `--signoff`
: Add a `Signed-off-by` trailer.

Cherry-picking a merge commit

A merge commit has two (or more) parents, so cherry-pick cannot know *which side* of the merge represents "the change". You must tell it with `-m` (mainline), giving the 1-based index of the parent to treat as the baseline.

```
# Cherry-pick merge commit deadbeef, using parent 1 as the mainline.
# The resulting patch is the diff of the merge against its first parent.
git cherry-pick -m 1 deadbeef
```

Parent 1 is normally the branch you were *on* when you merged (e.g. `main`), and parent 2 is the branch you merged *in*. So `-m 1` brings in "everything the merged branch added relative to main". Choosing the wrong parent produces a confusing or inverted patch.

> **Warning:** Cherry-picking a merge commit flattens the merged branch's combined changes into a single squashed patch — you lose the individual commits inside it. If you want those commits preserved, cherry-pick the range of underlying commits instead of the merge commit.

Scenario: cherry-pick conflicts, --continue and --abort

Situation. You cherry-pick a commit, but the target branch has diverged and the patch does not apply cleanly. Git stops mid-operation and leaves conflict markers in the affected files.

```
error: could not apply 9a8b7c6... Fix off-by-one in date formatter
hint: After resolving the conflicts, mark them with
hint: "git add/rm <pathspec>", then run "git cherry-pick --continue".
hint: You can instead skip this commit with "git cherry-pick --skip".
hint: To abort and get back to the pre-cherry-pick state, run "git cherry-pick
--abort".
```

Commands & resolution. Open the conflicted files, resolve the markers, stage the result, and continue:

```
# See which files conflicted
git status

# Edit each file to resolve the conflict markers, then stage:
git add path/to/resolved-file

# Resume the cherry-pick (reuses the original message)
git cherry-pick --continue
```

You have three other exits while a cherry-pick (or a multi-commit sequence) is paused:

COMMAND	EFFECT
`git cherry-pick --continue`	Commit the resolved change and proceed to the next commit in the sequence
`git cherry-pick --skip`	Discard the current commit and move on to the next one in a range
`git cherry-pick --abort`	Cancel the whole operation and restore the branch to its pre-cherry-pick state
`git cherry-pick --quit`	Stop the sequence but *keep* commits already made and the current index state (no rollback)

> **Tip:** Use `--abort` when you want to start over cleanly. Use `--quit` when a multi-commit pick already applied the commits you cared about and you simply want to stop without undoing them. `--skip` is for ranges where one commit is irrelevant (for example, it became a no-op on this branch).

> **Note:** If, after resolving conflicts, the staged result is empty (the change was already present), `git cherry-pick --continue` will refuse with "nothing to commit". Use `git cherry-pick --skip` to drop that now-redundant commit and keep going.

Duplicate-commit pitfalls and how to avoid them

Situation. You cherry-pick commit `X` from `feature` onto `main` to ship it early. Later, the whole `feature` branch is *merged* into `main`. Because the cherry-picked copy has a different hash than the original `X`, Git sees them as two distinct commits. History now shows the same change twice.

```
a1b1c1c Fix off-by-one in date formatter   (merged original)
e2f2g2g Fix off-by-one in date formatter   (earlier cherry-pick, different
hash)
```

With a plain merge this is usually harmless on the surface (the second application is often an empty diff). But it clutters history and can cause conflicts if intervening edits touched the same lines.

Why rebase handles this better than merge

When you *rebase* instead of merge, Git computes a *patch-id* (a content hash of the diff, independent of commit hash) for each commit. It then silently drops any commit whose patch-id already exists upstream. So if `X` was cherry-picked to `main` and you later run `git rebase main feature`, Git recognizes the duplicate and omits it. The same matching powers `git cherry` and `git log --cherry-mark`, which let you preview which commits are already present:

```
# List commits on feature that are NOT yet equivalent-present on main
git cherry -v main feature   # lines prefixed '+' are missing, '-' already applied
```

How -x helps you trace and clean up

Because cherry-pick with `-x` records `(cherry picked from commit <hash>)`, you (or a reviewer) can grep history to see that a duplicate is an intentional backport rather than an accidental re-apply. From there you can decide which copy to keep:

```
# Find every commit that was cherry-picked from somewhere
git log --grep="cherry picked from" --oneline
```

How to avoid duplicates

- **Prefer merge or rebase when you want the whole branch.** Reserve cherry-pick for genuinely partial transfers.
- **Forward-merge release branches.** Fix on the oldest branch, then merge upward, so newer branches inherit the real commit rather than a copy.
- **Rebase the feature branch onto the target** after an early cherry-pick, letting patch-id dedup remove the duplicate before you merge.
- **Always use `-x`** on backports so origins are auditable.
- **If you must merge after cherry-picking, rebase first** to drop the now-redundant commit, then merge.

> **Warning:** patch-id dedup only matches commits whose diffs are *identical*. If a conflict resolution changed even one line during the cherry-pick, the patch-ids differ and the duplicate will *not* be auto-dropped on rebase. In that case you will see the change twice and must resolve it manually.

Recap

Key takeaways:

- Cherry-pick copies a commit's *change* onto your current branch as a **new commit with a new hash**; use it for one or a few hand-picked commits, not whole branches.
- Backport a hotfix across release branches by cherry-picking with `-x` to record origins; forward-merge from the oldest branch when you can to avoid copies entirely.
- Extract one commit from a feature branch with `git cherry-pick <hash>`; use `-n` plus `git restore --staged` to take only part of a commit.
- Ranges: `A..B` excludes `A`, `A^..B` includes it; merge commits need `-m 1` to pick the mainline parent; `-n` stages without committing.
- On conflict: resolve, `git add`, then `--continue`; or `--skip`, `--abort`, or `--quit` depending on whether you want to drop, roll back, or stop.
- Cherry-pick then merge can show a change twice; rebase's patch-id dedup and `git cherry` help detect duplicates, and `-x` makes them traceable.

Merge vs Rebase vs Cherry-Pick

Three commands move work between branches, but they shape history in very different ways. Choosing the right one is rarely about taste — it is about whether you want a whole branch or a single commit, whether the history is shared with others, and whether you value a clean linear log or an honest record of how work actually happened. This chapter gives you a decision flowchart, a side-by-side comparison, and the team conventions that should guide you (plus the rare moments to break them).

Decision flowchart

Before reaching for a command, ask yourself three questions: **how much** do you want to move (a whole branch or one commit?), **who else** has the commits (is the history shared?), and **what shape** do you want the result to have (linear or branched?). The flowchart below routes you to the right tool.

The comparison at a glance

COMMAND	WHAT IT DOES	HISTORY SHAPE	WHEN TO USE	RISKS
merge	Joins two branches by creating a merge commit (or fast-forwards) that ties both histories together.	Branched — preserves the original commits and the point where lines diverged and rejoined.	Integrating a finished feature branch into a shared branch; you want the context of how it was developed.	Busy, diamond-shaped history; many merge commits can obscure the main line.
rebase	Replays your commits on top of a new base, creating new commits with new hashes.	Linear — looks as if the work was written sequentially on the latest base.	Cleaning up a local feature branch before sharing; updating onto the latest `main` without a merge commit.	Rewrites history; dangerous on shared branches; can force you to resolve the same conflict repeatedly.
cherry-pick	Copies one (or a few) specific commits onto the current branch as new commits.	Selective — only the chosen commits move; the rest of the source branch stays behind.	Backporting a single fix to a release branch; grabbing one commit without the whole branch.	Duplicated commits across branches; can cause confusing conflicts if later merged.

Walk the questions

1. **Do you want the whole branch, or just one (or a few) commit(s)?**
   - *Just one or a few commits* → use **cherry-pick**. Stop here.
   - *The whole branch* → continue to question 2.

2. **Have these commits already been pushed and pulled by other people (is the history shared)?**
   - *Yes, shared* → rewriting is unsafe, so prefer **merge**. (Only rebase if your team explicitly allows a coordinated force-push; see the next section.)
   - *No, still local / only yours* → continue to question 3.

3. **Do you want a clean, linear history, or do you want to preserve the branching context?**
   - *Linear & tidy* → use **rebase** to replay onto the latest base, then fast-forward merge.

- *Preserve context (when it diverged, that it was a feature branch)* → use **merge** with `--no-ff`.

```
# Question 1 answer: just one commit → cherry-pick
git switch release-2.x
git cherry-pick a1b2c3d        # copy one fix onto the release branch

# Question 2/3 answer: whole branch, not shared, want linear → rebase then merge
git switch feature/login
git rebase main                # replay feature commits on top of main
git switch main
git merge --ff-only feature/login   # fast-forward, no merge commit

# Question 2/3 answer: whole branch, shared, preserve context → merge
git switch main
git merge --no-ff feature/login     # always create a merge commit
```

> **One sentence to remember:** cherry-pick moves *a commit*, rebase moves *your branch* onto a new base, and merge *joins* two branches together.

> The golden rule of rebase: **never rebase commits that exist outside your repository** — that is, anything others may have based work on. Rewriting shared history forces everyone downstream into a painful reconciliation.

> **Backport scenario:** a critical bug is fixed on `main` in commit `9f8e7d6`, but customers are on `release-1.4`. You do not want the dozen other unrelated commits from `main` — only the fix. Cherry-pick is the precise tool:
>
> ```
> git switch release-1.4
> git cherry-pick 9f8e7d6        # bring only the fix
> git push origin release-1.4
> ```

> Rebase shines for *cleanup*: use `git rebase -i main` to squash noisy "fix typo" commits, reword messages, and reorder before opening a pull request. A reviewer sees a coherent story instead of your trial-and-error.

Team conventions and when to break them

Which command you reach for is often not your decision alone — the repository has a policy, written or implied. Conventions exist so that history stays predictable for everyone. So the default is to **follow the convention even when another approach would be slightly more convenient for you**. Below are the three most common policies and how to choose between them.

Common policies

No rebase on shared branches
: Branches that others pull (such as `main`, `develop`, or any long-lived release branch) are never rebased or force-pushed. Integration happens through **merge** only. This is the safest universal default.

Squash-merge pull requests
: Each PR collapses into a single commit on `main`. The mainline reads as one tidy commit per feature, while messy in-progress commits stay on the (soon deleted) branch. Common on GitHub/GitLab via the merge button.

Linear history via rebase
: Contributors rebase their feature branch onto the latest `main` before merging, and merges are fast-forward only. The result is a single straight line with no merge commits. Favored by projects that value a bisectable, easy-to-read log.

How to pick one

- **Optimize for newcomers and a simple log?** Squash-merge — one commit per PR is the easiest mental model.
- **Optimize for `git bisect` and a clean straight line?** Linear-history-via-rebase, enforced with `--ff-only`.
- **Optimize for an honest record and the lowest risk?** Merge-only with `--no-ff`; the branching is visible and nothing is ever rewritten.

```
# Enforce the policy locally so you do not fight the convention
git config pull.rebase true        # rebase-on-pull for a linear-history team
git config pull.ff only            # fast-forward only; fail rather than create
merges
git config merge.ff false          # for a merge-only team: always make a merge
commit
```

> Most hosting platforms can **enforce** these rules: branch protection can require linear history, require squash merges, or block force-pushes to protected branches. When the server enforces it, your local config should match to avoid surprises at push time.

The rare cases to deviate

Conventions are guardrails, not laws of physics. There are a few legitimate moments to step outside them — always with a heads-up to your team:

- **A secret or large binary was committed.** Removing it genuinely requires rewriting history (for example with `git filter-repo`), even on a shared branch. Coordinate a force-push and have everyone re-clone or reset.
- **A long-lived feature branch only you use.** Even in a "no rebase" shop, rebasing a branch that *no one else has pulled* is safe and keeps it current with `main`.
- **An emergency hotfix mid-release.** Cherry-picking one commit onto a frozen release branch may break a "merge-only" norm, but it is exactly the surgical move the situation calls for.

> If you must rewrite a shared branch, **announce it first** and use `git push --force-with-lease` instead of `--force`. The lease variant refuses to overwrite work that arrived after your last fetch, preventing you from silently clobbering a teammate's push.

A rule of thumb: let your situation choose

When you are not sure which command to reach for, let your situation decide. Here are the three cases you will meet most often.

Working alone → rebase

You are the only person on the branch, so nothing you rewrite can hurt anyone else. Rebasing onto the latest `main` keeps your history a clean, straight line — as if you wrote it all today.

Say you spent the afternoon on `feature/profile` while `main` moved ahead. No one else has your branch, so catch up with a rebase and keep the log tidy:

```
git switch feature/profile
git rebase main                       # replay your work on top of the latest main
git switch main
git merge --ff-only feature/profile   # fast-forward, no merge commit
```

Working with a team → merge

Once other people have pulled the branch, rewriting it would force every one of them to untangle their copy. Merge instead. It never rewrites what already exists, and the merge commit records when and how the work joined.

Say three of you share `develop`, and your feature is reviewed and ready. Merge it in — safe, and honest about the history:

```
git switch develop
git merge --no-ff feature/checkout    # keep the branch's context, rewrite nothing
```

You need just one commit → cherry-pick

Sometimes you do not want a whole branch — you want one specific commit somewhere else. That is exactly what cherry-pick is for.

Say you wrote a handy helper in commit `c0ffee1` on `feature/big-redesign`, which will not ship for weeks. Another branch needs that helper now. Grab just that commit and leave the rest behind:

```
git switch main
git cherry-pick c0ffee1               # take the one useful commit, nothing else
```

> **The short version:** alone and want it clean? **rebase**. Sharing with a team? **merge**. Need one commit, not the whole branch? **cherry-pick**. When two rules pull against each other, “never rewrite shared history” wins.

Recap

- **cherry-pick** for one commit, **merge** for a whole branch when history is shared or context matters, **rebase** for a whole branch that is still private and you want linear.
- Route every decision through three questions: how much to move, whether it is shared, and what shape you want.
- Never rebase or force-push history others have already pulled — unless you are removing a secret and have coordinated it.
- Follow your team's convention by default; deviate only for secrets, truly private branches, or emergency backports — and say so out loud.

PART 04

Mistakes, Disasters & Recovery

The mistakes developers and maintainers actually make, and how to recover calmly.

Common Developer Mistakes

Most "Git disasters" are not rare corruption bugs — they are ordinary slips: a commit on the wrong branch, a typo in a message, a secret pasted into a config file, a force-push that erased a teammate's work. The good news is that Git almost never throws data away immediately. This chapter walks through the six mistakes developers hit most often. For each one, it gives you the *symptom*, the *cause*, and the exact commands to recover — safely.

> **Key idea:** Git keeps your committed work reachable through the reflog for roughly 30–90 days even after you "delete" it. Before you panic, run `git reflog`. The SHA you need is usually still there.

Committed to the wrong branch — move the commit(s)

Symptom: You meant to commit a feature on `feature/login`, but `git status` reveals you were on `main` the whole time.

The clean way: create the right branch, then rewind the wrong one

If the wrong branch has not been pushed, the simplest fix is to "carry" your current commits onto a new branch. Then reset the branch you were mistakenly on.

```
# You are on main with 1+ accidental commits
git switch -c feature/login          # new branch keeps the commits
git switch main                      # back to the branch you dirtied
git reset --hard origin/main         # rewind main to match the remote
```

> **Warning:** `git reset --hard` discards any uncommitted changes in the working tree. Make sure every change you care about is already on `feature/login` before running it. If unsure, use `git reset --keep origin/main`, which refuses to clobber uncommitted edits.

The surgical way: cherry-pick specific commits

If only some commits belong elsewhere, copy them by SHA, then drop them from the original branch.

```
git switch feature/login
git cherry-pick a1b2c3d e4f5g6h      # copy two commits over
git switch main
git reset --hard HEAD~2              # remove the two we moved
```

The stash variant: nothing committed yet

If you noticed *before* committing — your work is only staged or modified — stashing is cleanest:

```
git stash                            # park the working changes
git switch -c feature/login          # or: git switch feature/login
git stash pop                        # reapply on the correct branch
```

Bad commit message or wrong author

Symptom: The last commit has a typo, an unhelpful message, or was made with a stale `user.email` (common after switching machines).

Fixing the most recent commit

```
# Reword the latest commit message
git commit --amend -m "fix: correct off-by-one in pagination"

# Fix the author on the latest commit
git commit --amend --author="Milad Golfam <milad.golfam@gmail.com>" --no-edit
```

The `--no-edit` flag keeps the existing message while changing only the author. To stop the wrong author from happening again, set your identity locally:

```
git config user.name  "Milad Golfam"
git config user.email "milad.golfam@gmail.com"
```

Fixing many commits at once

For older commits, an interactive rebase lets you mark each one to `reword`:

```
git rebase -i HEAD~5                 # change "pick" to "reword" on each line
```

To rewrite the author across a range non-interactively, use `--exec` with an automated amend:

```
git rebase -i HEAD~5 \
  --exec 'git commit --amend --author="Milad Golfam <milad.golfam@gmail.com>" --no-edit'
```

> **Warning:** `git commit --amend` and `git rebase` both *rewrite history* — they create new commits with new SHAs. If the old commits were already pushed and pulled by others, rewriting them requires a force-push and can disrupt collaborators. Amend freely *before* sharing; coordinate *after*.

Committed secrets or huge files

Symptom: You committed an `.env` file, an API key, or a 500 MB build artifact. `git push` is rejected for size, or a scanner flags a leaked credential.

If it is still in the most recent (unpushed) commit

```
git rm --cached .env                 # stop tracking, keep the local file
echo ".env" >> .gitignore
git commit --amend --no-edit         # rewrite the last commit without it
```

> **Warning:** Removing a secret from history is **not** enough. Once a credential has been committed — especially if pushed — assume it is compromised. **Rotate it immediately:** revoke the old key and issue a new one. History rewrites do not un-leak a value someone may already have cloned.

If it is buried deep in history

Amending only fixes the tip. To purge a file from *every* commit, use `git filter-repo` — the modern, recommended rewriter, with fast, flexible path and blob filtering:

```
# Remove a file from all of history (rewrites every commit)
git filter-repo --path secrets.yml --invert-paths
```

> **Tip:** Deep-history rewriting gets a full treatment in the recovery chapter, including how to coordinate the rewrite with everyone who has a clone. Reach for it only after rotating the secret.

Going forward, prevent this by listing the patterns in `.gitignore` before the first commit:

```
# .gitignore
.env
*.key
*.pem
build/
```

---

git push --force gone wrong

Symptom: You force-pushed a rebased branch, and a teammate reports their latest commits have vanished from the remote.

Why plain --force is dangerous

A plain `git push --force` tells the remote: "replace the branch with mine, no questions asked." If a colleague pushed after your last fetch, your push silently overwrites their commits — Git does not warn you, because you explicitly demanded the overwrite.

Recover the overwritten commits

The lost commits usually still exist somewhere. If the teammate still has them locally, they simply re-push. Otherwise, check the reflog on any clone that fetched them, or the server's reflog if you host your own remote:

```
git reflog show origin/main          # local record of what origin pointed to
git reflog                           # find the SHA of the clobbered tip
git push origin <sha>:main           # restore the remote to that commit
```

> **Key habit:** Always prefer `--force-with-lease` over `--force`. It refuses the push if the remote moved since your last fetch — protecting you from overwriting work you have not seen.

```
git push --force-with-lease          # safe: fails if someone else pushed first
```

Lost commits after a hard reset

Symptom: You ran `git reset --hard HEAD~3` to "undo" something, then realized you needed those commits after all. They are gone from `git log`.

They are not gone — they are just *unreferenced*. The reflog records every position `HEAD` has occupied:

```
git reflog                          # lists HEAD@{0}, HEAD@{1}, ... with SHAs
```

Find the SHA from just before the reset (the line ending in something like `reset: moving to HEAD~3`), then recover it. You have two choices:

```
# Option A: move the current branch back onto the lost commit
git reset --hard a1b2c3d

# Option B (safer): create a rescue branch pointing at it, inspect first
git branch rescue a1b2c3d
git log rescue
```

> **Tip:** Option B is the cautious default — it never touches your current branch. You can review the rescued commits, then merge or cherry-pick what you need.

Detached HEAD confusion

Symptom: Git prints *"You are in 'detached HEAD' state"* and you are unsure whether your commits are safe.

What it means

Normally `HEAD` points at a branch name, which points at a commit. In *detached HEAD*, `HEAD` points directly at a commit with no branch attached. You get there by checking out a raw SHA or a tag:

```
git checkout v1.4.0                 # checks out a tag → detached HEAD
git checkout a1b2c3d                # checks out a SHA → detached HEAD
```

You can look around and even commit, but those commits belong to no branch. They are reachable only via `HEAD` — and the moment you switch away, nothing references them.

How to keep work made in detached HEAD

```
# Made commits while detached? Anchor them to a branch BEFORE switching away
git switch -c experiment            # creates a branch at the current commit
```

> **Warning:** If you commit in detached HEAD and then run `git switch main` without creating a branch first, those commits become unreferenced. They are not immediately deleted (recover them via `git reflog`), but they are easy to lose. Always run `git switch -c <name>` while still on the commit you want to keep.

Recap

MISTAKE	FIRST MOVE	RECOVERY COMMAND
Commit on wrong branch	Don't reset blindly	`git switch -c right; git switch wrong; git reset --hard origin/...`
Bad message / author	Only if unpushed	`git commit --amend` / `git rebase -i`
Secret committed	**Rotate the secret**	`git rm --cached` + amend, or `git filter-repo`
Bad force-push	Check reflogs	`git push origin <sha>:main`; use `--force-with-lease`
Hard reset lost commits	Don't make new commits	`git reflog` then `git branch rescue <sha>`
Detached HEAD	Anchor before leaving	`git switch -c <name>`

> **Remember:** The reflog is your safety net for almost every mistake in this chapter. A good `.gitignore` plus `--force-with-lease` prevent most of them from happening at all. When in doubt, create a rescue branch and inspect before you rewrite.

Common Maintainer Mistakes

A maintainer holds the keys to a shared repository, and that responsibility raises the cost of every mistake. A slip that would inconvenience one developer on a private branch can rewrite history for an entire team, ship broken code to production, or destroy the credit owed to a contributor. This chapter walks through the mistakes maintainers make most often. Each one is paired with its impact, the prevention to put in place ahead of time, and the recovery you need when prevention fails.

Force-pushing a shared branch

Force-pushing replaces the remote branch's history with your local version. On a private feature branch this is routine. On a branch other people are based on — `main`, `develop`, a long-lived release branch — it is one of the most destructive actions a maintainer can take.

The impact

When you run `git push --force`, the remote's old commits are no longer referenced by the branch. Every teammate who had pulled those commits now has a divergent local copy. Their next `git pull` produces confusing merge conflicts, duplicate commits, or — if they force their own state back — silently loses your changes. Commits that only existed on the remote can become unreachable and eventually be garbage-collected.

```
# The dangerous command on a shared branch
git push --force origin main   # blindly overwrites whatever is on the remote
```

Prevention

Protect shared branches at the host level so force-pushes are simply rejected, and require everyone to use a safer flag locally.

```
# GitHub: enable branch protection (requires admin + gh CLI)
gh api -X PUT repos/OWNER/REPO/branches/main/protection \
  -F enforce_admins=true \
  -F allow_force_pushes=false \
  -F required_pull_request_reviews.required_approving_review_count=1

# Locally, mandate --force-with-lease instead of --force
# It refuses the push if the remote moved since you last fetched
git push --force-with-lease origin my-feature
```

> **Tip:** Make `--force-with-lease` the team default by aliasing it: `git config --global alias.pushf 'push --force-with-lease'`. It still lets you rewrite your own branches, but stops you from clobbering work a teammate pushed in the meantime.

If it already happened

The reflog remembers where the branch pointed before the force-push. Find the lost commit and restore it.

```
# Show recent positions of the branch tip
git reflog show main

# Suppose the good commit was abc1234, before the bad push
git branch rescue abc1234          # park it on a safe branch first
git push --force-with-lease origin rescue:main   # restore the remote

# If your own reflog is empty, ask a teammate who still has the old commits;
# their clone is an intact backup
```

> **Warning:** The reflog is local and expires (90 days by default for reachable entries, 30 for unreachable). Recover quickly. Once `git gc` prunes an unreferenced commit, it is gone unless another clone still holds it.

Merging unreviewed or broken code

Merging straight to a protected branch without review or without passing tests is how regressions reach everyone. The mistake is treating merge authority as permission to skip the gates.

Prevention: gate the branch

CONTROL	WHAT IT ENFORCES
Required reviews	At least N approvals before merge is allowed
Required status checks	CI must be green; a "merge only green" policy
Dismiss stale approvals	New pushes invalidate old approvals
Linear history / no direct push	All changes arrive through reviewed PRs

```
# Require a passing CI check named "build" before any merge
gh api -X PATCH repos/OWNER/REPO/branches/main/protection/required_status_checks \
  -F strict=true \
  -f 'contexts[]=build'
```

Fix: reverting a bad merge

If a broken merge slips through, do not force-push it away on a shared branch. Use `git revert` with `-m 1` to create a new commit that undoes the merge while keeping history intact.

```
# Identify the merge commit
git log --oneline --merges -5

# Revert it, keeping the first parent (mainline) as the base
git revert -m 1 <merge-sha>     # -m 1 = "the side we want to keep"
git push origin main
```

> **Key idea:** Reverting a merge records that the merged changes were undone. If you later want to merge that branch again, you must first revert the revert (or rebase the branch), because Git considers the changes "already merged." Document this for the contributor so they are not surprised.

Tangled history from inconsistent strategies

When some PRs are merged with merge commits, others squashed, and others rebased ad hoc, the resulting graph becomes unreadable. Bisecting gets harder, `git log --graph` gets tangled, and reverting becomes guesswork.

Pick one strategy and enforce it

STRATEGY	HISTORY SHAPE	BEST WHEN
Merge commit	Full branch topology preserved	You value an accurate record of how work was integrated
Squash merge	One commit per PR on mainline	You want a clean, linear, one-change-per-line log
Rebase & merge	Linear, every commit replayed	Contributors craft clean commit series and you want no merge bubbles

```
# Enforce a single merge method on GitHub (here: squash only)
gh api -X PATCH repos/OWNER/REPO \
  -F allow_merge_commit=false \
  -F allow_squash_merge=true \
  -F allow_rebase_merge=false
```

> **Tip:** Whatever you choose, write it in `CONTRIBUTING.md` and disable the other buttons in the host settings. Consistency is worth more than the theoretically "best" strategy — a predictable history is one you can reason about.

Mishandling contributor PRs and forks

Contributors usually work from forks. Maintainers stumble when they cannot fetch a PR locally to test it, or when their merge tooling strips the original author's name off the commit.

Fetching a PR or a fork

```
# Fetch a specific pull request by its number into a local branch
git fetch origin pull/42/head:pr-42
git switch pr-42                # test, build, review locally

# Or add the contributor's fork as a named remote
git remote add contributor https://github.com/THEIR_USER/REPO.git
git fetch contributor
git switch -c review-feature contributor/feature-branch
```

Preserving author attribution

The mistake here is committing a contributor's work under your own name, or squashing a multi-author PR into a single commit that erases co-authors.

```
# Applying a patch keeps the original author by default — don't override it
git am < contribution.patch     # preserves the Author: field from the patch

# When squashing, credit every author with trailers in the commit body:
#   Co-authored-by: Jane Dev <jane@example.com>
#   Co-authored-by: Sam Fixer <sam@example.com>
```

> **Warning:** A squash merge collapses every commit in a PR into one. If you do not carry forward `Co-authored-by:` trailers, contributors lose their commit credit entirely. Most hosts add these automatically for squash merges — verify yours does, and never hand-edit them away.

Broken tags and releases

Tags are the public, immutable names of your releases. Moving or deleting a tag that people have already pulled is far more disruptive than rewriting a branch. Build systems, package managers, and downstream users pin to those exact names.

Why moving a published tag hurts

Deleting a remote tag and re-creating it at a new commit causes a problem. Anyone who already fetched the old tag keeps the old commit, while new clones get the new one. Two users at "v1.2.0" now have different code — a silent, hard-to-debug split.

```
# The dangerous "fix" — re-pointing a released tag
git tag -d v1.2.0                  # delete locally
git push origin --delete v1.2.0    # delete on the remote
git tag v1.2.0 <new-sha>           # re-create at a different commit
git push origin v1.2.0            # downstream users now disagree about v1.2.0
```

> **Key idea:** Never reuse a tag name that has been released. If a release is wrong, ship a new tag — `v1.2.1` — and leave the broken one in place (optionally yanked at the package-registry level, with a note). Immutability is the whole point of a release tag.

Annotated vs lightweight, and signing

A lightweight tag is just a pointer with no metadata. An annotated tag is a real object carrying a tagger, date, and message — and it can be GPG-signed. Releases should always be annotated and, ideally, signed.

```
# Lightweight — avoid for releases
git tag v1.2.0

# Annotated — the correct choice for a release
git tag -a v1.2.0 -m "Release 1.2.0"

# Signed annotated tag, so users can verify authenticity
git tag -s v1.2.0 -m "Release 1.2.0"
git push origin v1.2.0

# Anyone can verify the signature
git tag -v v1.2.0
```

Tip: Push tags explicitly (`git push origin v1.2.0`) rather than relying on `git push --tags` sweeping up every local tag. Sweeping can publish half-baked or experimental tags you never meant to release.

Recap

Key takeaways for maintainers:

- Protect shared branches: forbid force-push at the host, mandate `--force-with-lease` locally, and recover lost commits through the reflog or a teammate's clone.
- Gate merges with required reviews and green CI; undo a bad merge with `git revert -m 1` , not a force-push.
- Choose one merge strategy and enforce it in host settings and `CONTRIBUTING.md` for a readable history.
- Fetch PRs with `git fetch origin pull/ID/head` , keep forks as named remotes, and preserve author and `Co-authored-by` credit.
- Treat released tags as immutable: annotate and sign them, push them explicitly, and never reuse a published tag — ship a new version instead.

The Recovery Toolkit

Almost every "I just destroyed my work" moment in Git is recoverable. Git is a content-addressable database that rarely throws data away immediately — instead it keeps unreferenced objects around for weeks. This chapter assembles the core recovery tools — `reflog`, `reset`, `revert`, `restore`, `fsck`, and `bisect` — into one toolkit. It then closes with a calm, repeatable disaster-response procedure you can follow when your pulse is racing.

> **Key idea:** A commit you can no longer see is not the same as a commit that is gone. As long as the commit object still exists in the object database — and it usually does for ~30 days — you can point a branch back at it and bring it back to life.

reflog: your time machine

The reflog records where `HEAD` and each branch tip *have been*, even after operations that rewrite history. Every commit, checkout, reset, rebase, merge, and amend appends an entry. This is your single most important recovery tool: if you know where a ref pointed before the mistake, you can move it back.

```
# Show the movement history of HEAD (most recent first)
git reflog

# Equivalent, more explicit form
git reflog show HEAD
```

```
a1b2c3d HEAD@{0}: reset: moving to HEAD~3
9f8e7d6 HEAD@{1}: commit: add billing report
1234abc HEAD@{2}: commit: fix rounding error
5678def HEAD@{3}: checkout: moving from main to feature/x
```

The HEAD@{n} syntax

Each entry is addressable as `HEAD@{n}`, where `n` counts backwards from the current position. `HEAD@{0}` is where you are now; `HEAD@{1}` is where you were one move ago. You can also use time-based selectors like `HEAD@{2.hours.ago}` or `HEAD@{yesterday}`.

```
# Inspect what HEAD@{1} actually pointed at
git show HEAD@{1}

# A branch has its own reflog too
git reflog show feature/x
git show feature/x@{1}
```

Recovering from almost anything

Suppose a botched `git reset --hard` erased three good commits. The reflog still holds the old tip. Point a branch (or your current `HEAD`) back at it:

```
# Find the entry from before the bad reset, e.g. HEAD@{1}
git reflog

# Option A: move the current branch back to it
git reset --hard HEAD@{1}

# Option B (safer): recover onto a new branch first, then inspect
git branch recovered HEAD@{1}
```

> **Warning:** The reflog is local and not pushed. It expires: reachable entries default to 90 days (`gc.reflogExpire`) and unreachable ones to 30 days (`gc.reflogExpireUnreachable`). Recover sooner rather than later, and never assume a teammate's clone shares your reflog — it does not.

Undo with reset (soft/mixed/hard) vs revert

`git reset` moves the current branch pointer and, depending on the mode, optionally updates the index (staging area) and working tree. It is the tool for *local, not-yet-shared* history. The three modes differ only in how much they touch.

MODE	MOVES HEAD	RESETS INDEX	RESETS WORKING TREE	TYPICAL USE
`--soft`	Yes	No	No	Uncommit but keep changes staged (e.g. to re-commit differently)
`--mixed` (default)	Yes	Yes	No	Uncommit and unstage; changes remain in working tree
`--hard`	Yes	Yes	Yes	Discard commits, staging, and working changes entirely

```
# Undo the last commit but keep its changes staged
git reset --soft HEAD~1

# Undo the last commit, keep changes in the working tree (unstaged)
git reset --mixed HEAD~1   # or simply: git reset HEAD~1

# Throw away the last commit AND all local changes (destructive)
git reset --hard HEAD~1
```

> **Warning:** `git reset --hard` discards uncommitted working-tree changes that exist in no commit and therefore cannot be recovered from the reflog. Run `git stash` or commit first if there is any doubt.

reset rewrites history; revert is additive

`git reset` rewrites history by moving the branch pointer — fine locally, dangerous on a branch others have pulled. `git revert` instead creates a *new* commit that applies the inverse of a target commit, leaving the original history intact. That makes it the safe choice on shared branches.

```
# Safely undo a pushed commit by adding an inverse commit
git revert <sha>

# Revert a range without committing each one separately
git revert --no-commit <old-sha>..<new-sha>
git commit -m "Revert the broken feature batch"
```

TOOL	WHAT IT DOES	HISTORY	SAFE ON SHARED BRANCHES?
`git reset`	Moves the branch pointer; optionally edits index/working tree	Rewrites	No — requires a force-push
`git revert`	Adds a new commit that inverts a previous one	Preserves and appends	Yes
`git restore`	Restores file contents in the working tree and/or index	Untouched	Yes (does not move commits)

restore and checkout for files

`git restore` (introduced in Git 2.23) splits the file-level half of the overloaded `git checkout` into a focused, predictable command. It operates on file contents only and never moves `HEAD`.

Discarding working-tree changes

```
# Discard unstaged edits to a file, reverting to the index version
git restore path/to/file.js

# Discard ALL unstaged changes in the tree (destructive)
git restore .
```

> **Warning:** Plain `git restore <file>` overwrites uncommitted edits with no undo. Those changes are not in any commit, so the reflog cannot bring them back.

Unstaging and pulling from a specific revision

```
# Unstage a file (keep the edits in the working tree)
git restore --staged path/to/file.js

# Restore a file's content from a specific commit/branch into the working tree
git restore --source=HEAD~2 path/to/file.js
git restore --source=main config/settings.yml

# Restore into BOTH the index and working tree at once
git restore --source=<rev> --staged --worktree path/to/file.js
```

The legacy checkout form

Older muscle memory and pre-2.23 environments use `git checkout` for the same job. It still works, but the `--` separator is important to separate file names from branch names.

```
# Legacy: discard working-tree changes to a file
git checkout -- path/to/file.js

# Legacy: pull a file from another revision
git checkout <rev> -- path/to/file.js
```

> **Tip:** Prefer `git restore` and `git switch` in new work. Splitting checkout's two jobs (changing branches vs. restoring files) into separate verbs prevents a whole class of accidental file overwrites.

Recovering deleted branches and dangling commits

Deleting a branch only removes a pointer. The commits it referenced remain as *dangling* (unreachable) objects until garbage collection prunes them. Recovery means finding the SHA and pointing a fresh branch at it.

Find the SHA via reflog

```
# HEAD's reflog often still shows the deleted branch's last tip
git reflog
# Look for: "checkout: moving from feature/lost to main"

# Recreate the branch at that commit
git branch feature/lost <sha>
```

When the reflog entry is gone: fsck

If the branch reflog was also removed, scan the object database directly for unreachable commits.

```
# List dangling commits, blobs, and trees
git fsck --dangling

# Same, but materialize results under .git/lost-found for browsing
git fsck --lost-found
```

```
dangling commit 7e1c9a4f6b2d8e3a1c0f9b8a7d6e5c4b3a2f1e0d
dangling blob   3f2a1b0c9d8e7f6a5b4c3d2e1f0a9b8c7d6e5f4a
```

```
# Inspect a candidate before committing to it
git show 7e1c9a4

# Resurrect it as a branch
git branch recovered 7e1c9a4
```

> **Tip:** A surprising amount of "lost" work — a stash you dropped, a commit that lost its only ref — surfaces in `git fsck --dangling`. Pipe candidate SHAs through `git log -1 --format='%ci %s'` to triage by date and message quickly.

bisect: finding the commit that broke it

When a bug appeared somewhere across hundreds of commits, `git bisect` performs a binary search through history to pinpoint the exact commit that introduced it. You mark one known-bad and one known-good commit; Git checks out the midpoint repeatedly, narrowing the range with each answer.

Manual bisect

```
# Begin a bisect session
git bisect start

# Mark the current (broken) commit as bad
git bisect bad

# Mark a commit you know was working as good
git bisect good v1.4.0

# Git checks out the midpoint. Test it, then report:
git bisect good   # ... if this revision works
git bisect bad    # ... if it is broken
# Repeat until Git names the first bad commit
```

```
b9d3f2a is the first bad commit
commit b9d3f2a...
    Author: ...
    refactor: replace currency parser
```

Automated bisect

If you have a script or test that exits `0` for good and non-zero for bad, let Git drive the whole search unattended.

```
# Git runs the command at each step and classifies automatically
git bisect start HEAD v1.4.0
git bisect run ./run-tests.sh

# After you have the answer, return to your original HEAD
git bisect reset
```

> **Note:** An exit code of `125` from a `git bisect run` command means "skip this commit, it cannot be tested" (e.g. it does not build) — distinct from a genuine pass or fail. Always finish with `git bisect reset` to restore your branch.

Purging secrets from history

A committed API key, password, or private file is not removed by deleting it in a later commit — it remains in every snapshot of history and in every clone. Removing it requires *rewriting history* to strip the content from past commits.

git filter-repo (preferred)

`git filter-repo` is the modern, fast, officially recommended tool (the old `git filter-branch` is deprecated for being slow and error-prone).

```
# Remove a file from all of history
git filter-repo --path config/secrets.yml --invert-paths

# Or replace a leaked literal string everywhere it appears
echo 'AKIA1234567890ABCD==>REMOVED' > replacements.txt
git filter-repo --replace-text replacements.txt
```

Rewrite, force-push, and rotate

```
# After rewriting, publish the new history (coordinate first!)
git push --force-with-lease --all
git push --force-with-lease --tags
```

> **Warning:** Rotate the secret regardless of how thoroughly you scrub history. Once a credential has been pushed, assume it is compromised — it may live in forks, clones, CI logs, and provider caches you cannot reach. Revoke and reissue the key; the history rewrite is damage control, not a fix on its own.

> **Key idea:** History rewrites change every downstream SHA. Coordinate with the whole team before force-pushing: collaborators must re-clone or carefully rebase their work, or they will reintroduce the old (secret-bearing) commits on the next push.

A calm disaster-response checklist

When something goes wrong, the worst move is a fast, panicked second action that compounds the first. Slow down and work the list. The guiding principle: Git almost never deletes committed data immediately, so you usually have more time than your adrenaline suggests.

> **Tip:** Before any risky operation, make a throwaway backup branch — `git branch backup/$(date +%s)` — or a mirror clone. A free undo point costs seconds and turns most "disasters" into non-events.

1. **Stop.** Do not run another Git command reflexively. Take your hands off the keyboard and read the error and the current state first.

2. **Do not force-push.** Pushing — especially with `--force` — can spread the problem to everyone and overwrite the very history you need to recover. Keep the damage local while you assess.
3. **Note the current SHAs.** Record where every relevant ref points right now: `git rev-parse HEAD`, `git branch -v`, and `git reflog`. You may need to return to this exact state.
4. **Check the reflog.** Run `git reflog` (and the branch's reflog) to locate the last good commit before the mistake. This solves most cases.
5. **Work on a copy.** Create a backup or recovery branch — `git branch recovered <sha>` — or clone the repo, and do your surgery there. Never experiment on the only copy of fragile state.
6. **Recover.** Apply the right tool: `git reset --hard HEAD@{n}` to move a tip back, `git restore` for files, `git fsck --lost-found` for dangling commits, `git revert` for shared branches.
7. **Verify.** Inspect the result before trusting it: `git log --oneline --graph`, `git status`, run the build and tests. Confirm the recovered state is genuinely correct.
8. **Clean up and communicate.** Only now delete temporary branches, and — if a shared rewrite was needed — coordinate the force-push and tell the team what changed and what they must do.

PART 05

Commit Quality & Conventions

What a good commit looks like and how often to make one.

Writing Beautiful Commits

A commit is more than a snapshot of code; it is a message to the future. Months from now, when a colleague (or you) runs `git blame` on a puzzling line, the commit message is the only explanation that survives. This chapter shows how to write commits that read well, machine-parse cleanly, and tell a coherent story of *why* the code changed.

Anatomy of a Great Commit Message

Every good commit message follows the same three-part shape: a short **subject line**, a blank line, and an optional **body**, sometimes closed by a **footer**. Git itself treats the first line specially, so the structure is not just stylistic.

- **Subject line** — a concise summary (~50 characters) of what the commit does. This is what appears in `git log --oneline`, in pull-request lists, and in tools like `git shortlog`.
- **Blank line** — mandatory separation. Without it, Git cannot tell the subject from the body, and many tools will mangle the output.
- **Body** — explains *what* changed and especially *why*. Do not describe *how*; the diff already shows that.
- **Footer** — structured metadata: issue references, co-authors, sign-offs, and breaking-change notices.

Here is an annotated example. The arrows are annotations, not part of the real message:

```
Add retry logic to the payment gateway client        ← subject (~50 chars, imperative)
                                                      ← blank line (required)
The gateway occasionally returns 503 during peak     ← body: the WHY
load. Without retries a single transient failure
aborts the whole checkout, which users experience
as a lost cart.

Retry up to three times with exponential backoff,    ← body: the WHAT (not line-by-line how)
capping total wait at five seconds so we never
block the request thread indefinitely.
                                                      ← blank line before footer
Closes #482                                           ← footer: issue reference
Co-authored-by: Dana Lee <dana@example.com>          ← footer: attribution
```

> The single most valuable sentence in any commit body answers the question *"Why was this change necessary?"* The diff already records the *how*; only the human can record the *why*.

The 50/72 Rule and Imperative Mood

Two conventions make commit history readable across nearly every tool: the **50/72 rule** and the **imperative mood**.

50/72

Keep the subject line to roughly **50 characters** and never more than about 72. Short subjects stay readable in `git log --oneline`, in GitHub lists, and in terminal columns. Wrap the body at **72 characters**. That way `git log` (which indents output by four spaces) still fits in an 80-column terminal without ugly wrapping.

> Most editors can show a ruler at column 50 and 72. You can also let Git pick your editor and configure `git config --global core.editor` so commit editing always happens with your wrapping rules in place.

Imperative mood

Write the subject as a command: **"Add X"**, not "Added X" or "Adds X". The trick to remember it: a good subject completes the sentence *"If applied, this commit will ___"*.

```
Good:  Fix race condition in session cache
Bad:   Fixed race condition in session cache
Bad:   Fixes race condition in session cache
Bad:   Fixing race condition in session cache
```

Why imperative? Git's own generated messages already use it — *"Merge branch..."*, *"Revert..."* — so your commits match the tooling. It is also the shortest grammatical form and reads as an instruction the patch carries out.

Conventional Commits

Conventional Commits is a lightweight convention that adds machine-readable structure to the subject line. The format is:

```
<type>(<scope>): <subject>
```

The `type` is required. The `(<scope>)` is optional and names the area of the codebase affected, and the `subject` follows the usual imperative rules. For example: `feat(auth): add OAuth2 login`.

The standard types

TYPE	MEANING	TYPICAL SEMVER IMPACT
`feat`	A new feature for the user	MINOR
`fix`	A bug fix for the user	PATCH
`docs`	Documentation only changes	none
`style`	Formatting, whitespace, semicolons; no code-meaning change	none
`refactor`	Code change that neither fixes a bug nor adds a feature	none
`perf`	A change that improves performance	PATCH
`test`	Adding or correcting tests	none
`build`	Changes to the build system or dependencies	none
`ci`	Changes to CI configuration and scripts	none
`chore`	Other changes that do not modify src or test files	none

A few well-formed Conventional Commit subjects:

```
feat(api): add pagination to the search endpoint
fix(parser): handle empty input without throwing
docs(readme): clarify the install steps for Windows
refactor(cache): extract eviction policy into its own module
perf(render): memoize the layout calculation
```

How it powers changelogs and SemVer

Because the type is machine-readable, tools like `semantic-release`, `standard-version`, and `commitizen` can read your history and act on it automatically. A run of commits since the last release is parsed: a `feat` bumps the MINOR version, a `fix` bumps PATCH, and a breaking change bumps MAJOR. The same parse generates a grouped **CHANGELOG** (Features, Bug Fixes, and so on) without anyone writing it by hand.

Body, Footers, Issue Refs, and BREAKING CHANGE

The footer block carries structured metadata. Each entry sits on its own line, after a blank line that separates it from the body.

Referencing issues

Hosting platforms recognize keywords that link — and even auto-close — issues when the commit lands on the default branch:

```
Closes #123
Fixes #456, #457
Refs #789
```

Use `Closes` / `Fixes` when the commit fully resolves the issue, and `Refs` (or `Re #123`) when it only relates to one.

Attribution and provenance

Two footers are widely understood by Git hosts:

`Co-authored-by`
: Credits additional authors (for pair or mob programming). GitHub renders each co-author's avatar on the commit.

`Signed-off-by`
: Asserts the Developer Certificate of Origin. Generated with `git commit -s`; required by projects like the Linux kernel.

```
git commit -s -m "fix(net): close socket on early return"
```

```
Co-authored-by: Sam Rivera <sam@example.com>
Signed-off-by: Milad Golfam <milad.golfam@gmail.com>
```

BREAKING CHANGE and the "!" shorthand

A commit that breaks backward compatibility must announce it. There are two equivalent ways. The footer form is explicit:

```
feat(config): rename the timeout option to timeoutMs

BREAKING CHANGE: the timeout key is removed. Update configs
to use timeoutMs, which is now expressed in milliseconds.
```

The shorthand adds a `!` before the colon. It can stand alone or accompany the footer:

```
feat(config)!: rename the timeout option to timeoutMs
```

> Any commit containing a `BREAKING CHANGE:` footer or a `!` marker triggers a **MAJOR** version bump under SemVer — regardless of whether its type is `feat` or `fix`. Use it deliberately, and always explain the migration path in the body.

Atomic Commits: One Logical Change Each

An **atomic commit** captures exactly one logical change — and the whole of that change. It builds, it passes tests on its own, and it can be reverted in isolation without dragging unrelated work along.

Why bother

- **Reviewable** — a reviewer can reason about one idea at a time.
- **Revertible** — `git revert` undoes one feature cleanly, not a tangle of half-related edits.
- **Bisectable** — `git bisect` can pinpoint the exact commit that introduced a bug, but only if each commit is a coherent, working unit.
- **Cherry-pickable** — backporting a single fix to a release branch is trivial.

How to achieve it

When your working tree has accumulated several unrelated changes, stage them selectively instead of using `git add .`. Interactive staging lets you pick individual hunks:

```
git add -p          # choose hunks interactively, one logical change at a time
git commit -m "fix(auth): reject expired tokens"

git add -p          # stage the next, unrelated change separately
git commit -m "docs(auth): document token lifetime"
```

If you have already made a messy commit, reshape history before sharing it. An interactive rebase lets you split, reorder, squash, and reword:

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

> Mixing a refactor and a feature in one commit is a classic anti-pattern. If the feature later proves buggy and must be reverted, the unrelated refactor is reverted with it. Keep "moving the furniture" and "adding a new room" in separate commits.

Good vs Bad Commit Examples Side by Side

The difference between a weak commit and a strong one is almost always the presence of context. Compare:

WEAK	STRONG
`fix stuff`	`fix(cart): prevent negative quantities at checkout`
`wip`	`feat(search): add fuzzy matching to product lookup`
`updated code`	`refactor(db): replace raw SQL with query builder`
`asdf`	`test(auth): cover token-expiry edge cases`
`changes`	`perf(list): avoid re-sorting on every keystroke`

Now the full form. A weak commit forces the next reader to open the diff and guess intent:

```
fix stuff

(no body, no reason, no link to the bug it fixes)
```

A strong commit hands the reader everything they need without opening a single file:

```
fix(cart): prevent negative quantities at checkout

A user could decrement quantity below zero, which produced a
negative line total and a credit on the final invoice. Clamp
the quantity to a minimum of one in the reducer, and guard the
API so a crafted request cannot bypass the UI.

Closes #931
```

> If you genuinely need a checkpoint commit while you work, that is fine — just squash or reword the `wip` commits before pushing or opening a pull request, so the shared history stays clean.

Recap

- Structure every message as **subject · blank line · body · footer**; the body explains *why*, not how.
- Keep the subject near **50 characters**, wrap the body at **72**, and write in the **imperative mood** ("Add", not "Added").
- Adopt **Conventional Commits** (`<type>(<scope>): <subject>`) to power automated changelogs and SemVer.
- Use footers for `Closes #123` , `Co-authored-by` , and `Signed-off-by` ; mark incompatibilities with `BREAKING CHANGE:` or `!` .
- Keep commits **atomic** — one logical change each — using `git add -p` and `git rebase -i` to shape them.

How Often Should You Commit

There is no universal number of commits per hour, but there is a reliable instinct you can build: commit whenever one small, coherent piece of work reaches a stable point. This chapter explains why frequent commits pay off, how to separate scratch local history from the polished history you publish, how to size commits around features and tasks, and how to tidy everything up before it reaches a shared branch.

Commit early, commit often — the why

The single most useful habit in Git is to commit small and commit frequently. A commit is not a ceremony or a release; it is a cheap, local snapshot. The more of them you have, the more options Git can give you later.

Small commits buy you four concrete advantages:

- **Easy review.** A reviewer reading a 40-line commit with a clear message understands it in seconds. A 2,000-line commit forces them to reverse-engineer your intent.
- **Easy revert.** If one change turns out to be wrong, `git revert` on a focused commit removes exactly that change and nothing else.
- **Easy bisect.** `git bisect` finds the commit that introduced a bug by binary search. When each commit changes one thing, the commit it lands on tells you precisely what broke.
- **Less lost work.** Anything committed is recoverable, even after a bad `reset`, through the reflog. Uncommitted work in your editor is one stray command away from gone.

> **Rule of thumb:** commit when *one logical thing works*. A function passes its test, a bug is fixed, a refactor compiles cleanly — that is a commit. If you cannot describe the change in a single short sentence, it is probably two commits.

```
# Stage and commit one coherent change
git add src/auth/login.js
git commit -m "Fix off-by-one in session expiry check"
```

Notice the commit is scoped to one file and one idea. The opposite anti-pattern — working all day and running `git commit -am "stuff"` at 6 p.m. — throws away every benefit above.

Local save points vs publishable history

Frequent committing sometimes worries people: "Won't my history be a mess of half-finished commits?" Locally, that is completely fine. Git distinguishes between **private history** (commits only on your machine) and **shared history** (commits you have pushed and others may have built on).

Private history is yours to rewrite

Before you push, your commits are scratch paper. Make them as messy as you like — checkpoint after every experiment — then reshape them into something readable. This is the core idea: *commit often for safety, rewrite before sharing for clarity.*

```
# Rapid local checkpoints while exploring
git commit -am "WIP: trying approach A"
git commit -am "WIP: approach A doesn't work, trying B"
git commit -am "WIP: B works, needs cleanup"

# Later, collapse these into clean commits before pushing
git rebase -i origin/main
```

> The golden rule of rewriting: never rewrite history that has been pushed to a shared branch and that teammates may have pulled. Rewriting private, unpushed commits is safe and encouraged; rewriting shared commits forces painful re-syncs on everyone else.

Per-feature, per-task, and WIP commits

Granularity is a spectrum. A useful mental model is three sizes of commit, each serving a different purpose.

Per-task commits

The ideal published commit corresponds to one task: "Add rate limiting to the login endpoint." It is small enough to review, large enough to stand on its own, and it usually maps to a ticket or issue. Tie commits to your tracker by referencing the issue ID so the history and the backlog stay connected.

```
git commit -m "Add rate limiting to login endpoint

Limits to 5 attempts per minute per IP.

Closes #482"
```

On GitHub, GitLab, and similar hosts, keywords like `Closes #482` or `Fixes #482` automatically close the linked issue when the commit lands on the default branch.

Per-feature grouping

A feature is usually several tasks. Keep each task its own commit on a feature branch rather than one giant commit. That way each step is reviewable and bisectable. The branch name carries the feature; the commits carry the steps.

WIP commits and fixups

Work-in-progress commits are temporary save points you fully intend to clean up. When you spot a small mistake in an earlier commit, do not write "fix typo" as a new commit. Instead, record it as a `fixup` targeting the original, then let an autosquash rebase fold it in automatically.

```
# Make a correction that belongs to an earlier commit
git add src/parser.js
git commit --fixup=a1b2c3d        # marks it for commit a1b2c3d

# Autosquash folds every fixup into its target
git rebase -i --autosquash origin/main
```

> If you are interrupted mid-task and your work does not even compile, commit anyway with a clear `WIP:` prefix. It is safer than a stash, survives branch switches, and you will squash it out before pushing. A messy committed state always beats lost uncommitted work.

Squashing before merge to keep history clean

The tension between "commit often" and "keep history clean" is resolved by rewriting at the boundary between private and shared history. You commit freely while working, then squash and reorder just before the branch merges.

Interactive rebase

Run `git rebase -i` against the branch point to reorder, reword, combine, or drop commits. Mark commits with `pick` to keep, `squash` to merge into the previous one and edit the message, or `fixup` to merge silently.

```
git rebase -i origin/main

# In the editor, e.g.:
#   pick   a1b2c3d Add rate limiting to login endpoint
#   fixup  d4e5f6a WIP: tweak limit
#   squash 7g8h9i0 WIP: tests for rate limiting
```

Squash-merge on pull requests

Most hosting platforms offer a "Squash and merge" button. It collapses an entire PR into one commit on the target branch, using the PR title and description as the message. This keeps the main branch tidy — one commit per merged PR — while preserving all the intermediate detail inside the PR for anyone who wants it.

The balance to strike: commits should be **atomic** (each does one complete thing and leaves the build green) but not **fragmented** (no trail of "fix", "fix again", "actually fix" noise). Squash the noise; keep the meaningful steps.

> Squash-merge is great for many small PRs, but it discards the internal commit structure of a large PR. For a substantial feature where each commit is genuinely atomic and well-described, a plain merge or rebase-merge preserves that valuable history. Match the strategy to the PR.

Cadence by project type

The right rhythm depends on who else touches the repository and how much process surrounds it. The advice below assumes the same core habit everywhere — frequent local commits — and varies only in how that work is grouped, reviewed, and published.

PROJECT TYPE	LOCAL COMMIT CADENCE	PUBLISH / PR CADENCE	CONVENTIONS
Solo hobby	Whenever something works; very frequent, even WIP	Push directly to main as often as you like	Minimal process; reasonable messages still help future-you
Small team	Frequent on a feature branch	One PR per task or small feature, reviewed within a day	Short-lived branches, squash-merge, issue references
Large team / monorepo	Frequent locally; rebase often to stay current	Small, single-purpose PRs; merge multiple times per day	Strict atomic commits, conventional commit messages, required CI & reviews
Open source	Frequent on a fork branch	One focused PR per change; cadence set by maintainer review	Clean, signed-off, well-documented commits; follow CONTRIBUTING guidelines

Reading the table

As you move down the table, the *local* habit barely changes — commit early and often — but the published unit gets smaller and the polish gets stricter. A solo developer can push raw history. A contributor to a large project or an open-source maintainer is expected to hand over commits that read like a clean narrative, because many strangers will review, bisect, and revert them for years.

> When joining any project, read its `CONTRIBUTING.md` and skim `git log` first. Matching the existing commit style — message format, granularity, squash policy — gets your changes accepted faster than imposing your own.

> **Recap:** Commit early and often locally — small commits make review, revert, bisect, and recovery easy, and the rule is "commit when one logical thing works." Treat unpushed commits as private scratch history you can freely rewrite, and reserve clean, atomic commits for shared branches. Use WIP and `--fixup` commits as temporary save points, then collapse them with `git rebase -i --autosquash` or a squash-merge before publishing. Keep the same frequent-commit habit across solo, team, and open-source work; only the size and polish of what you publish should change.

PART 06

Workflows & Modern Practice

Branching models, AI-assisted Git, and collaborating through pull requests.

Git Flow & Branching Models

A branching model is the social contract your team writes in Git. It decides where work begins, how features are integrated, when releases are cut, and how emergency fixes reach production. There is no universally correct model — only models that fit a given team size, release cadence, and deployment style. This chapter walks through the four dominant approaches: the classic Git Flow, the lightweight GitHub Flow, the environment-aware GitLab Flow, and trunk-based development. By the end you should be able to pick one deliberately rather than inheriting whatever the last project used.

> Every model below is just a convention layered on top of plain Git branches and merges. Git itself has no opinion about what `main` or `develop` mean — the meaning lives in your team's discipline and tooling (CI, branch protection, review rules).

Git Flow: main, develop, feature, release, hotfix

Git Flow, introduced by Vincent Driessen in 2010, is the most structured of the common models. It defines two long-lived branches and three categories of short-lived support branches.

The branch roles

`main`
: The production branch. Every commit on `main` is a released, tagged version. You never commit to it directly.

`develop`
: The integration branch where completed features accumulate between releases. It always reflects the "next release in progress."

`feature/<name>`
: Branched from `develop`, merged back into `develop`. One per unit of work.

`release/<version>`
: Branched from `develop` when the next release is feature-complete. Only bug fixes, version bumps, and release prep happen here. Merged into both `main` and `develop`.

`hotfix/<version>`
: Branched from `main` to patch a production emergency. Merged into both `main` and `develop` so the fix is not lost in the next release.

A typical feature cycle

```
# start a feature off develop
git switch develop
git switch -c feature/payment-retry

# ... work, commit ...
git add .
git commit -m "Add exponential backoff to payment retries"

# integrate it back into develop
git switch develop
git merge --no-ff feature/payment-retry
git branch -d feature/payment-retry
```

The `--no-ff` flag forces a merge commit even when a fast-forward is possible, preserving the feature branch as a visible unit in history.

Cutting a release and shipping a hotfix

```
# freeze the next release
git switch develop
git switch -c release/1.4.0
# bump version, fix last-minute bugs, then:
git switch main
git merge --no-ff release/1.4.0
git tag -a v1.4.0 -m "Release 1.4.0"
git switch develop
git merge --no-ff release/1.4.0   # carry fixes back
git branch -d release/1.4.0

# emergency production fix
git switch main
git switch -c hotfix/1.4.1
git commit -am "Fix null pointer in checkout"
git switch main
git merge --no-ff hotfix/1.4.1
git tag -a v1.4.1 -m "Hotfix 1.4.1"
git switch develop
git merge --no-ff hotfix/1.4.1
```

ASCII lifecycle

```
main     o─────────────────o──────────o   (v1.4.0)   (v1.4.1)
          \               / \          \
release    \      r───r   \          \
            \    /          \          \
develop  o──o──o─────o──────o──────────o
              \        /      (hotfix back-merge)
feature        f──f──f
```

When it fits — and its downsides

Git Flow shines when you ship **versioned software on a schedule**. Think desktop apps, mobile apps behind store review, libraries with semantic versioning, or any product where several versions are in use at once.

> Git Flow is a poor fit for continuous deployment. The `develop` / `main` split, long-lived branches, and merge ceremony create friction and large, infrequent integrations — exactly what modern web teams try to avoid. Many teams adopt it by reflex and suffer for it.

GitHub Flow (lightweight, PR-based)

GitHub Flow strips the model down to one rule: `main` is always deployable. Everything else is a short-lived branch that becomes a pull request.

1. Branch off `main` with a descriptive name.
2. Commit and push regularly.
3. Open a pull request to start discussion and run CI.
4. Review, iterate, and get approval.
5. Merge to `main` and deploy.

```
git switch main
git pull
git switch -c add-csv-export
# ... commits ...
git push -u origin add-csv-export
# open a PR, get review, then merge via the platform
# deploy happens from main, often automatically
```

> Keep branches alive for hours or days, not weeks. The shorter the branch, the smaller the diff, the easier the review, and the rarer the merge conflict. If a branch lingers, rebase it onto `main` frequently to stay current.

GitHub Flow is ideal for web apps and services with **continuous deployment** and a strong test suite. Its weakness is that it has no release branches: if you must support multiple released versions at once, a single `main` is not enough.

GitLab Flow and trunk-based development

GitLab Flow

GitLab Flow keeps the simplicity of GitHub Flow but adds **environment** or **release** branches to model how code promotes through stages. Changes flow in one direction: `main` → `staging` → `production`.

```
# feature merges into main as usual
git switch main
git merge --no-ff feature/search-filters

# promote main into the next environment
git switch production
git merge --no-ff main      # deploy to production from here
```

For versioned products, GitLab Flow uses `release/<version>` branches instead. The rule: fixes are *cherry-picked from `main` into release branches*, never the reverse — so `main` always has every fix.

```
# fix lands on main first
git switch main
git commit -am "Fix race in cache eviction"
# then back-port the single fix to a maintained release
git switch release/2.1
git cherry-pick <sha>
```

Trunk-based development

Trunk-based development pushes the short-branch idea to its limit: developers integrate into a single trunk (`main`) at least once a day. Branches live for a few hours at most. Incomplete work is hidden behind **feature flags** rather than isolated on long branches.

```
git switch -c quick-fix
# small change committed and merged the same day
git switch main
git merge quick-fix
git push
```

The trick is shipping unfinished features safely: code reaches `main` while inactive, gated by a runtime flag.

```
# pseudocode inside the application
if featureEnabled("new-dashboard"):
    render_new_dashboard()
else:
    render_old_dashboard()
```

> Trunk-based development is the model behind most high-performing continuous-delivery teams. It demands strong automated testing, fast CI, and feature-flag tooling — but in return it eliminates merge hell and keeps the team continuously integrated.

Choosing a model for your team and release cycle

Pick by answering three questions: how big is the team, how often do you release, and how do you deploy.

- **Solo or small team, ship continuously:** GitHub Flow. Minimal ceremony, fast feedback.
- **Web team with staging/prod environments:** GitLab Flow with environment branches makes promotion explicit and auditable.
- **Versioned/packaged software with multiple supported releases:** Git Flow, or GitLab Flow with release branches and cherry-picks.
- **Large team, mature CI, true continuous delivery:** trunk-based development with feature flags scales best and avoids long-lived divergence.

> Models are not all-or-nothing. A common hybrid is trunk-based development on `main` plus short-lived `release/<version>` branches only when an actual release must be maintained. Start as simple as your release cycle allows and add structure only when a real problem demands it.

> The most expensive mistake is mismatching model to cadence: running Git Flow on a team that deploys ten times a day, or trunk-based development without the test coverage and flags to make it safe. The model must serve the release cycle, not fight it.

Branch lifecycles compared

MODEL	LONG-LIVED BRANCHES	RELEASE CADENCE	COMPLEXITY	BEST FOR
Git Flow	`main`, `develop` (+ release, hotfix, feature)	Scheduled, versioned	High	Packaged/desktop/mobile software, semantic versioning
GitHub Flow	`main` only	Continuous, deploy from `main`	Low	Web apps/services with strong CI
GitLab Flow	`main` + environment or release branches	Continuous with promotion, or maintained releases	Medium	Teams needing staging/prod stages or multiple versions
Trunk-based	`main` (trunk) only	Continuous, many times per day	Low branching, high tooling	Large teams doing true continuous delivery with feature flags

Read the table by lifecycle, not just branch count. In Git Flow a change passes through feature → develop → release → main, touching four branches. In GitHub Flow it is feature → main. In trunk-based it is, often, a single same-day commit to `main` behind a flag. The fewer hops a change makes to production, the faster your feedback loop — at the cost of relying more heavily on automated safety nets.

Recap

- **Git Flow** gives maximum structure (`main` + `develop` + release/hotfix/feature) and suits scheduled, versioned releases — but is heavy for continuous deployment.
- **GitHub Flow** is one rule — `main` stays deployable — with short-lived PR branches; simple and popular for web services.
- **GitLab Flow** adds environment or release branches and promotes changes forward via merge or cherry-pick.
- **Trunk-based development** integrates into one trunk daily, hiding unfinished work behind feature flags; the foundation of high-performing CD teams.
- Choose by **team size, release frequency, and deployment style** — and prefer the simplest model your release cycle allows.

Git in the AI Era

Large language models have quietly become part of the everyday Git workflow. They draft commit messages from your staged diff, summarize a pull request, explain a gnarly merge conflict, and propose a rebase plan in seconds. None of this changes what Git *is* — the object model, the refs, the reflog are all unchanged — but it changes the experience of the human sitting in front of it. This chapter is a practical, tool-neutral tour. You'll see where AI genuinely helps a developer or maintainer, how to wire it into the commands you already run, and — just as important — where it still gets things wrong and why you remain accountable for every commit that carries your name.

> **The one rule that governs this whole chapter:** AI drafts, you decide. Every command below produces a *suggestion*. The diff you commit, the resolution you keep, the description you publish — those are yours. Treat AI output as a fast first draft from a confident junior colleague who has never seen your production incident history.

AI-generated commit messages from your diff

The single most popular AI-assisted Git task is also the simplest to reason about: turn a staged diff into a well-formed commit message. The idea is a clean pipeline. Git already knows exactly what changed, so you feed that change to a model and ask it to describe it in your house style (usually Conventional Commits: `type(scope): summary`, imperative mood, a wrapped body).

The conceptual pipeline

At its core the flow is three stages: **capture the staged diff → send it to a model with your conventions → review and commit the message**. The capture step is plain Git:

```
# What the model needs to see: only what is staged for this commit
git diff --staged

# Conceptual pipeline — pipe the staged diff into a CLI that drafts the message.
# (Replace `commit-msg-tool` with whichever tool you use; examples follow.)
git diff --staged | commit-msg-tool > /tmp/msg.txt

# Review the draft, edit if needed, then commit with it
git commit --edit --file=/tmp/msg.txt
```

Notice the deliberate use of `git diff --staged` (equivalently `git diff --cached`) rather than `git diff`. You want the model to describe the snapshot you are about to record, not your unstaged scratch work. Always land in an editor (`--edit`) before the commit is final — that is your review gate, not a formality.

> **Never pipe secrets into a model.** A staged diff can contain an accidentally-committed `.env` file, an API key, a private hostname, or customer data. Whatever you send to a hosted model leaves your machine. Stage deliberately, scan the diff yourself first, and keep a `.gitignore` plus a secret-scanning pre-commit hook (covered below) between you and an embarrassing leak.

Tools and integrations

The ecosystem ranges from full agentic CLIs that can run Git for you, to tiny single-purpose scripts that do nothing but draft a message. They are not mutually exclusive — many developers use a heavyweight assistant for review and a lightweight script for commits.

TOOL	WHAT IT DOES	SHAPE	NOTES
Claude Code	Agentic terminal assistant. Reads your repo, drafts commit messages, summarizes diffs, explains conflicts, proposes rebase plans, and can run Git commands on request.	Interactive CLI / agent	Broadest scope; keeps a human approval step for actions. A good default for AI-assisted Git on modern Claude models (e.g. `claude-opus-4-8`).
GitHub Copilot	Inline code completion plus commit-message suggestions in supported editors/IDEs and a "generate commit message" action that reads the staged diff.	Editor / IDE integration	Tightly bound to the editor and GitHub PR flow; convenient where you already live in VS Code.
aicommits	Tiny CLI focused on one job: read `git diff --staged` and produce a commit message, optionally in Conventional Commit format.	Single-purpose CLI	Minimal, scriptable, fast. Wraps the `git commit` step directly.
opencommit	CLI that generates commit messages from the staged diff, supports Conventional Commits and emoji conventions, and can be installed as a Git hook.	Single-purpose CLI	Configurable conventions; can hook into `prepare-commit-msg` to pre-fill the editor.

A pattern worth remembering: the *diff-to-message* tools (aicommits, opencommit) are deterministic in scope — they only ever describe a diff, so the blast radius of a bad output is one editable commit message. The agentic tools (Claude Code, Copilot Chat) have wider scope and therefore deserve wider scrutiny.

Writing prompts that produce good messages

If you build your own helper, or steer a chat-based assistant, the quality of the output depends almost entirely on two inputs: **the diff** and **your conventions**. Give the model the actual staged change and a clear statement of the format you want. Vague prompts produce vague, generic messages ("update code"); specific prompts produce reviewable ones.

An example prompt

Here is a prompt template that encodes the common rules — Conventional Commits, imperative mood, a 50-character subject and a body wrapped at 72 columns:

```
You are writing a git commit message. Follow these rules exactly:

- Format: Conventional Commits — type(scope): subject
  Allowed types: feat, fix, docs, style, refactor, test, chore, perf, build, ci
- Subject line: imperative mood ("add", not "added"/"adds"),
  no trailing period, ⩽ 50 characters.
- Leave one blank line, then a body that explains WHY the change
  was made, not just what. Wrap the body at 72 columns.
- Do not invent changes that are not in the diff.
- Output only the commit message. No backticks, no commentary.

Here is the staged diff:

<diff>
{paste the output of: git diff --staged}
</diff>
```

Two details earn their keep. "Explain WHY, not just what" pushes the model past restating the diff line-by-line — the diff already shows *what*; the message should record intent. "Do not invent changes that are not in the diff" is a direct guard against hallucination (see below). A model handed a small diff will sometimes invent a grander story; this instruction reins it in.

> For repositories with strong conventions, commit a short `COMMIT_CONVENTIONS.md` and feed it to the model alongside the diff. The model follows a written spec far more reliably than it guesses your team's unwritten habits — and the spec doubles as onboarding documentation for new humans.

AI-assisted code review and PR summaries

Beyond commits, the highest-leverage use of AI in Git is summarizing change. A pull request that touches forty files is hard for a reviewer to load into their head. A model can read the full diff and produce a structured description — what changed, why, what to watch for — that gives the human reviewer a map before they dive in.

Generating a PR description from the branch diff

```
# Capture everything this branch adds over the base (e.g. main)
git diff main...HEAD > /tmp/pr.diff

# Or just the list of commits, for a higher-level summary
git log --oneline main..HEAD

# Pipe the diff to your assistant to draft a PR body, then review it
git diff main...HEAD | pr-summary-tool > /tmp/pr-body.md

# With the GitHub CLI, open the PR using the drafted (and reviewed) body
gh pr create --title "Add rate limiting to the API gateway" --body-file /tmp/pr-
body.md
```

The three-dot `main...HEAD` form diffs against the merge base, so the summary reflects what your branch *introduces*, not unrelated changes that landed on `main` in the meantime. The model can also surface review suggestions — potential bugs, missing tests, edge cases — which are genuinely useful as a checklist.

> AI review is a **second pair of eyes, never the only pair**. A model can flag a plausible-looking null-pointer risk that is actually unreachable, and it can confidently miss a real concurrency bug. Keep human approval mandatory before merge. Use the AI summary to make the human review faster and more thorough — not to skip it. A generated PR description should also be read by its author before publishing; it is your name on the PR.

AI for conflict resolution and history cleanup

Merge conflicts and history rewrites are where Git frightens people, and where a model's ability to *explain* pays off most. The win here is understanding, not blind automation: the model translates conflict markers and rebase mechanics into plain language so you can make the call.

Explaining a conflict

```
# See exactly which files are conflicted
git diff --diff-filter=U --name-only

# Show both sides plus the common ancestor for a conflicted file
git checkout --conflict=diff3 -- path/to/file
git diff path/to/file | conflict-explainer-tool
```

The `diff3` view is the key input: it shows *theirs*, *ours*, and the common ancestor. Now the model (and you) can reason about what each side intended rather than guessing from two competing snapshots. A good assistant will explain the divergence and propose a resolution — but you apply it, run the tests, and confirm the result compiles and behaves.

Drafting a rebase plan

```
# Ask the model to draft an interactive-rebase plan, then YOU run the rebase
git log --oneline -n 10 origin/main..HEAD

# Start the interactive rebase and adapt the proposed pick/squash/reword plan
git rebase --interactive origin/main
```

> History rewriting (`rebase`, `commit --amend`, `filter-repo`) is destructive. **Always verify before and after.** Before you start, note your current commit so you can recover: `git rev-parse HEAD` — and remember the reflog (`git reflog`) holds your prior tips for a while. After a model-suggested resolution or rebase, run the test suite and re-read the diff (`git diff`) yourself. A conflict resolution that *looks* right and compiles can still silently drop one side's intent.

Guardrails: reviewing what the AI commits

The discipline that keeps AI assistance safe is a small set of mechanical checks that run regardless of who — or what — wrote the change. The principle is simple: **nothing reaches a commit unreviewed**, and some classes of mistake are caught by automation rather than by attention.

- **Read the diff before committing.** Whether a human or a model staged it, run `git diff --staged` and read it. AI-staged changes can include edits you did not ask for.
- **Watch for secrets.** Models occasionally surface, or fail to remove, credentials in a diff. Scan for them.

- **Watch for hallucinated APIs.** AI-written code can call functions, flags, or libraries that do not exist. The code must actually run and the tests must pass — a clean-looking diff is not a passing build.
- **Watch for license issues.** Generated code can resemble licensed source. For anything substantial, confirm provenance and that your project's license permits it.

A pre-commit hook as a safety net

```
# .git/hooks/pre-commit  (make it executable: chmod +x .git/hooks/pre-commit)
#!/usr/bin/env bash
set -euo pipefail

# 1. Block obvious secrets staged for commit
if git diff --cached | grep -nE '(API_KEY|SECRET|PASSWORD|BEGIN [A-Z]+ PRIVATE
KEY)'; then
  echo "Refusing to commit: possible secret in staged diff." >&2
  exit 1
fi

# 2. Block leftover conflict markers (whitespace/marker check)
if ! git diff --cached --check; then
  echo "Refusing to commit: conflict markers or whitespace errors." >&2
  exit 1
fi

# 3. Run tests / linters so hallucinated APIs fail loudly
make test
```

> For team-wide enforcement, manage hooks with a framework such as `pre-commit` (and set `core.hooksPath` so everyone runs the same checks) plus a dedicated secret scanner. A hook that runs on every machine is worth more than a code-review convention that depends on memory — especially once AI is generating large diffs quickly.

What AI still gets wrong — keep a human in the loop

For all its usefulness, an LLM has structural blind spots that no prompt fully removes. Understanding them is what separates productive use from misplaced trust.

- **It does not know your intent.** The model sees the diff, not the bug report, the customer call, or the architectural decision behind the change. It can describe *what* changed and guess at *why*, but the real "why" lives in your head and your tracker.

- **It lacks context the repo does not contain.** Deployment constraints, an incident that shaped a workaround, a teammate's in-flight branch — none of it is in the diff, so none of it informs the output.
- **It can be confidently wrong.** A hallucinated function name, an invented commit rationale, or a conflict resolution that drops one side's logic arrives in the same fluent, assured tone as a correct answer. Fluency is not correctness.
- **You own the commit.** Your name goes on the author line. `git blame` points at you, not the model. Accountability does not delegate.

> The right mental model: AI is an accelerator for the parts of Git that are tedious — drafting messages, summarizing diffs, explaining conflicts — and a research assistant for the parts that are confusing. It is not a replacement for judgment. Keep a human in the loop on every action that writes to history, and you get most of the speed with none of the surrendered control.

> **Recap**
>
> - The core pattern is a pipeline: `git diff --staged` → a model that knows your conventions → a message you review and commit.
> - Tools span a spectrum: agentic assistants (Claude Code, GitHub Copilot) for review and explanation; focused CLIs (aicommits, opencommit) for diff-to-message. Modern Claude models such as `claude-opus-4-8` are a solid default for AI-assisted Git, with lighter models for cost-sensitive automation.
> - Good output needs the real diff plus an explicit spec: Conventional Commits, imperative mood, 50/72 wrapping, and "don't invent changes not in the diff."
> - AI shines at PR summaries, conflict explanation (use `diff3`), and rebase planning — but a human approves the merge and verifies every rewrite.
> - Guardrails are mechanical: read the staged diff, scan for secrets, run the tests so hallucinated APIs fail, and enforce it all with a pre-commit hook.
> - The model never knows your full intent and can be confidently wrong. You own the commit — keep yourself in the loop.

Collaboration & Pull Requests

Pull requests are where individual work becomes shared history. Whether you are contributing to an open-source project you have no write access to, or pushing branches into a company monorepo, the same core skills apply. You keep your branch current, present changes for review, respond to feedback without rewriting other people's understanding of your work, and merge in a way that leaves the main branch clean. This chapter walks through the two dominant collaboration models, the rebase-versus-merge question for updating a branch, review and merge mechanics, and the guardrails that keep a busy repository sane.

Fork-and-PR vs shared-repo workflows

There are two structural ways teams collaborate through pull requests, and the choice is usually decided by trust and access rather than preference.

When each model is used

Fork-and-PR
: Used in open-source and across organizational boundaries. You do not have write access to the upstream repository, so you create your own copy (a *fork*), push branches there, and open a pull request from your fork into upstream. Maintainers review and merge; you never touch their branches directly.

Shared-repo
: Used inside a team where everyone has write access to one repository. You push feature branches straight into the shared repo and open PRs branch-to-branch. There is no fork; the only thing protecting `main` is branch protection (covered below).

> The git mechanics are nearly identical. The only real difference is *where* your branch lives: in a fork (a separate remote) or in the shared origin. Everything about commits, rebasing, and review is the same.

Adding the upstream remote

When you fork, your fork becomes `origin`. The original project is conventionally named `upstream`. You add it once so you can pull in changes that landed after you forked.

```
# origin is your fork; add the original repo as upstream
git remote add upstream https://github.com/original-org/project.git

# confirm both remotes
git remote -v
# origin    https://github.com/you/project.git (fetch/push)
# upstream  https://github.com/original-org/project.git (fetch)
```

Keeping a fork in sync

Your fork does not update itself. Before starting new work, sync your local `main` with upstream, then push it to your fork so the fork's `main` matches the project's.

```
# fetch everything upstream has done
git fetch upstream

# fast-forward your local main to match upstream main
git switch main
git merge --ff-only upstream/main

# push the refreshed main up to your fork (origin)
git push origin main
```

The `--ff-only` flag is deliberate. If your local `main` has diverged, the merge refuses rather than creating a stray merge commit on a branch that should mirror upstream. With the GitHub CLI you can do the sync server-side in one step:

```
# sync the fork's default branch from upstream without cloning
gh repo sync you/project --source original-org/project
```

Keeping branches up to date

While your PR sits in review, `main` keeps moving. Eventually you need to bring those changes in so your branch merges cleanly and CI runs against current code. You have two tools: **rebase onto main** or **merge main in**.

```
# start from current main
git fetch origin

# option A: rebase your branch onto the latest main
git switch feature/login
git rebase origin/main

# option B: merge the latest main into your branch
git switch feature/login
git merge origin/main
```

Rebase vs merge for updating a branch

ASPECT	REBASE ONTO MAIN	MERGE MAIN IN
Resulting history	Linear; your commits replay on top of main	Preserves a merge commit and true timeline
Commit hashes	Rewritten (new hashes)	Unchanged
Conflict handling	Resolve per commit as it replays	Resolve once in a single merge commit
Push after	Requires `--force-with-lease`	Plain `git push`
Safe on shared branch?	No — rewrites others' commits	Yes
Review diff noise	Clean; reviewers see only your work	Adds merge commits into the PR

> Rule of thumb: rebase a branch only *you* work on, for a clean linear PR. If others share the branch, or it is already widely pushed, merge `main` in instead.

Force-pushing after a rebase

A rebase rewrites your commits, so the remote branch no longer fast-forwards. A normal `git push` is rejected. The safe way to overwrite is `--force-with-lease`, which refuses if someone else pushed in the meantime — unlike a blind `--force`, which would silently discard their work.

```
# after a successful rebase, update the PR branch safely
git push --force-with-lease origin feature/login
```

> Never use plain `git push --force` on a branch others might have pulled. Always prefer `--force-with-lease`: it checks that the remote is where you last saw it before overwriting, so you cannot clobber a teammate's commits.

Reviewing, requesting changes, and merging cleanly

Review etiquette

- Review the change, not the person; phrase comments as questions or suggestions.
- Distinguish blocking issues from nits — prefix optional comments with `nit:`.
- Approve when the change is good enough to ship, not only when it is perfect.
- For authors: keep PRs small and focused so reviewers can hold the whole change in their head.

```
# review a PR from the terminal
gh pr checkout 482          # check out the PR branch locally
gh pr diff 482              # read the diff
gh pr review 482 --approve
gh pr review 482 --request-changes --body "Please add a test for the empty case"
```

Addressing feedback with fixup commits

When a reviewer asks for changes, do not silently amend or force-push over everything — that makes their re-review harder. Instead, add small `--fixup` commits that point at the commit they belong to. Reviewers can see exactly what changed, and you can collapse them automatically just before merge.

```
# make a fix that belongs to an earlier commit
git commit --fixup=a1b2c3d

# later, fold all fixups into their targets automatically
git rebase -i --autosquash origin/main
git push --force-with-lease
```

> During active review, push fixup commits as plain commits so the diff is additive. Only autosquash-and-force-push once review is essentially done, so reviewers never lose their place mid-discussion.

The three merge buttons

Most platforms offer three ways to land a PR. They differ entirely in what they do to `main`'s history.

BUTTON	WHAT LANDS ON MAIN	BEST FOR
Merge commit	All branch commits plus one merge commit joining the lines	Preserving full per-commit history and the branch shape
Squash and merge	One new commit containing the whole PR's diff	Messy work-in-progress history; one logical change per PR
Rebase and merge	Each branch commit replayed onto main, no merge commit	Clean linear history where every commit is meaningful

```
# merge a PR with each strategy via the CLI
gh pr merge 482 --merge     # merge commit
gh pr merge 482 --squash    # squash into one commit
gh pr merge 482 --rebase    # rebase commits onto base, linear
```

> Pick one strategy per repository and enforce it in settings, rather than letting each author choose. Squash-merge is the most common default because it keeps `main` readable regardless of how chaotic the branch was.

Protecting branches and enforcing conventions

Write access plus human error is how `main` breaks. Branch protection turns conventions into rules the server enforces.

Branch protection, required reviews and checks

- Disallow direct pushes to `main` — all changes go through a PR.
- Require one or more approving reviews before merge.
- Require status checks (CI, lint, tests) to pass, and require the branch to be up to date with base first.
- Optionally require a linear history, signed commits, or that conversations be resolved.

```
# enable protection on main with the GitHub API via gh
gh api -X PUT repos/your-org/project/branches/main/protection \
  --input protection.json
# protection.json sets required_status_checks, required_pull_request_reviews, etc.
```

CODEOWNERS

A `CODEOWNERS` file maps paths to the people or teams who must review changes there. Combined with "require review from code owners," it routes PRs to the right reviewers automatically.

```
# .github/CODEOWNERS
*               @your-org/maintainers
/api/           @your-org/backend
/web/           @your-org/frontend
*.sql           @your-org/dba
```

Commit and PR templates, status checks

Templates standardize what authors write. A pull request template pre-fills the PR description with a checklist; a commit message template nudges toward a convention like Conventional Commits.

```
# point git at a local commit-message template
git config commit.template .gitmessage.txt

# PR template lives here and is loaded automatically by GitHub
# .github/pull_request_template.md
```

> Status checks are reported by CI providers via the commit status / checks API. Branch protection's "required checks" list references those check names exactly, so a typo means the gate silently never applies — verify the names match a real run.

Recap

- Fork-and-PR is for outside contributors; shared-repo is for trusted teams — the git mechanics are the same, only the remote differs.
- Add `upstream` and sync regularly with `git fetch upstream` plus a fast-forward merge.
- Rebase a private branch for a clean linear PR; merge `main` in when the branch is shared. After rebasing, push with `--force-with-lease`, never plain `--force`.
- Address review feedback with `--fixup` commits, then autosquash before merge.
- Merge commit preserves history, squash collapses to one commit, rebase keeps it linear — choose one per repo.
- Branch protection, required reviews and checks, CODEOWNERS, and templates turn conventions into enforced guardrails.

PART 07

Power Tools & Automation

Hooks, aliases, tags, and CI to make Git work for you.

Hooks, Aliases & Productivity

Git ships with sensible defaults, but its real power for daily work comes from the small layers you build on top. These are shorthand aliases for the commands you type a hundred times a day, configuration tweaks that smooth out rough edges, and hooks that automate checks so a broken commit never leaves your machine. This chapter covers the productivity tooling that separates a casual user from someone who uses Git fluently.

Useful aliases and config tweaks

An **alias** is a custom name for a Git command (or a chain of commands). They live in your Git configuration and are expanded whenever you invoke `git <alias>`. Define them once globally and they follow you to every repository on the machine.

Defining aliases

You can set aliases from the command line. Each writes a key under the `[alias]` section of `~/.gitconfig`.

```
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.unstage 'reset HEAD --'
git config --global alias.last 'log -1 HEAD --stat'
git config --global alias.lg "log --graph --abbrev-commit --decorate --all \
  --format=format:'%C(bold blue)%h%C(reset) - %C(green)(%ar)%C(reset) %s %C(dim
white)- %an%C(reset)'"
```

After this, `git st` runs `git status`, `git unstage <file>` removes a file from the index without touching the working tree, and `git lg` prints a compact, colorized commit graph spanning all branches.

> **Tip:** An alias that begins with `!` runs an arbitrary shell command rather than a Git subcommand. For example, `git config --global alias.cleanup '!git branch --merged | grep -v "\*" | xargs -n 1 git branch -d'` deletes branches already merged into the current one.

A starter alias table

ALIAS	EXPANDS TO	WHAT IT DOES
`st`	`status`	Show working tree status
`co`	`checkout`	Switch branches or restore files
`br`	`branch`	List or manage branches
`ci`	`commit`	Record changes
`unstage`	`reset HEAD --`	Remove a file from the index
`last`	`log -1 HEAD --stat`	Show the most recent commit
`lg`	`log --graph ...`	Pretty commit graph
`amend`	`commit --amend --no-edit`	Fold staged changes into the last commit
`wip`	`!git add -A && git commit -m WIP`	Quick checkpoint commit

Config tweaks worth setting

Beyond aliases, a handful of configuration options remove recurring friction:

```
# Always rebase instead of merge when pulling, keeping history linear
git config --global pull.rebase true

# On first push of a new branch, create the upstream automatically
git config --global push.autoSetupRemote true

# Remember how you resolved conflicts and replay them automatically
git config --global rerere.enabled true

# Use your preferred editor for commit messages and interactive rebases
git config --global core.editor "vim"        # or "code --wait", "nano"

# Saner default branch name for new repositories
git config --global init.defaultBranch main
```

`pull.rebase`
: Replays your local commits on top of the fetched upstream instead of creating a merge commit, avoiding noisy "Merge branch" entries.

`push.autoSetupRemote`
: Lets a bare `git push` on a brand-new branch work without the usual `--set-upstream` dance.

`rerere.enabled`
: "Reuse recorded resolution" — Git memorizes conflict resolutions so repeated rebases or merges resolve the same conflicts for you.

> **Note:** Configuration is layered. Values in a repository's `.git/config` override the user-level `~/.gitconfig`, which in turn overrides the system file. Inspect the effective value and its origin with `git config --show-origin --get pull.rebase`.

Pre-commit and commit-msg hooks

Client-side hooks are scripts Git runs at specific points in your workflow. They live in `.git/hooks` inside each repository. Git installs sample files there with a `.sample` suffix; rename one (dropping the suffix) and make it executable to activate it. Any executable language works — the shebang line decides the interpreter.

A sample pre-commit hook

The `pre-commit` hook runs before the commit message is even requested. If it exits with a non-zero status, the commit is aborted — making it the ideal place to run a linter or a fast test suite.

```
#!/bin/sh
# .git/hooks/pre-commit

# Collect staged files that are added, copied, or modified
staged=$(git diff --cached --name-only --diff-filter=ACM | grep '\.js$')

if [ -n "$staged" ]; then
  echo "Running linter on staged files..."
  if ! npx eslint $staged; then
    echo "Lint failed. Fix the issues above or commit with --no-verify." 1>&2
    exit 1
  fi
fi

# Run the test suite, discarding stdout but keeping errors visible
echo "Running tests..."
npm test >/dev/null 2>&1 || { echo "Tests failed; commit aborted." 1>&2; exit 1; }

exit 0
```

Make it run with `chmod +x .git/hooks/pre-commit`. The redirect `2>&1` merges standard error into standard output, and `>/dev/null` silences ordinary test chatter while still surfacing failures.

A commit-msg hook for Conventional Commits

The `commit-msg` hook receives the path to a temporary file holding the proposed message as its first argument, `$1`. You can read that file and reject messages that do not follow your team's format — here, the Conventional Commits specification.

```
#!/bin/sh
# .git/hooks/commit-msg

msg_file=$1
first_line=$(head -n 1 "$msg_file")

pattern='^(feat|fix|docs|style|refactor|perf|test|build|ci|chore|revert)(\
(.+\))?!?: .+'

if ! echo "$first_line" | grep -Eq "$pattern"; then
  echo "Commit message does not follow Conventional Commits." 1>&2
  echo "Expected:  type(scope): subject" 1>&2
  echo "Example:   feat(auth): add token refresh" 1>&2
  exit 1
fi

exit 0
```

> **Warning:** Hooks live under `.git/hooks`, which is *not* part of the repository and is never cloned or pushed. By default your teammates get none of your hooks. Anyone can also bypass a client-side hook with `git commit --no-verify`. So treat hooks as a convenience for catching mistakes early, never as a security boundary — enforce hard rules on the server (CI or branch protection).

Managing hooks across a team

Because hooks are not shared automatically, teams use a manager that stores hook definitions in a committed config file and installs the real hooks on setup:

- **pre-commit** (the framework, `pre-commit.com`) — declare checks in `.pre-commit-config.yaml`; run `pre-commit install` to wire them in. Language-agnostic and widely used in Python-heavy projects.
- **Husky** — popular in the JavaScript ecosystem; hook scripts live under a committed `.husky/` directory and are installed via a `prepare` npm script.

> **Tip:** Without a framework, you can still share hooks. Commit them to a `.githooks/` directory and point Git at it for the whole repo with `git config core.hooksPath .githooks`. Each developer runs that command once after cloning.

---

stash, worktree, and submodule essentials

git stash — shelve work in progress

A stash records your dirty working tree and index, then reverts to a clean `HEAD` so you can switch context — fixing an urgent bug, say — without committing half-finished work.

```
git stash push -m "half-done login form"   # shelve tracked changes
git stash push -u                          # -u also stashes untracked files
git stash list                             # show the stash stack
git stash apply stash@{0}                  # restore but keep the stash entry
git stash pop                              # restore the latest and drop it
git stash drop stash@{1}                   # delete one entry
```

Stashes form a stack; `stash@{0}` is the most recent. Use `apply` when you want to reuse the same changes in several branches, and `pop` when you are done with the entry.

git worktree — multiple checkouts, one repository

Normally a repository has a single working directory. `git worktree` lets you check out additional branches into separate directories that share the same object store — no second clone, no duplicated history. This is perfect for reviewing a colleague's branch while leaving your current work untouched.

```
# Create a new working directory for an existing branch
git worktree add ../project-hotfix hotfix/login

# Create a directory and a brand-new branch in one step
git worktree add -b feature/export ../project-export

git worktree list      # show all linked worktrees
git worktree remove ../project-hotfix   # clean up when finished
```

> **Note:** Each branch can be checked out in only one worktree at a time. Attempting to check out a branch already active elsewhere is rejected, which prevents two directories from racing on the same ref.

git submodule — repositories within repositories

A **submodule** embeds another Git repository inside yours at a fixed commit, recorded in a `.gitmodules` file. It is useful for vendoring a shared library while keeping its history separate.

```
# Add a submodule at a given path
git submodule add https://github.com/example/libfoo.git vendor/libfoo

# After cloning a project that has submodules, fetch and check them out
git clone https://github.com/example/app.git
cd app
git submodule update --init --recursive

# Pull the latest commits for every submodule
git submodule update --remote
```

The `--recursive` flag descends into nested submodules, and `--init` populates any that have not been initialized yet. The parent repository tracks a specific submodule commit, so updating a submodule means staging that new pointer in the parent and committing it.

> **Warning:** A fresh clone leaves submodule directories empty until you run `git submodule update --init --recursive` (or clone with `--recurse-submodules`). Forgetting this is the most common cause of "missing files" confusion with submodules.

Key takeaways:

- Aliases turn frequent commands into muscle memory; set `st`, `lg`, `unstage`, and friends globally once.
- Config tweaks like `pull.rebase`, `push.autoSetupRemote`, and `rerere.enabled` remove daily friction.
- Client-side hooks (`pre-commit`, `commit-msg`) catch problems before they land, but live in `.git/hooks`, are not shared, and can be bypassed — use a manager like pre-commit or Husky to distribute them, and enforce real rules in CI.
- `stash` shelves work in progress, `worktree` gives you parallel checkouts without a second clone, and `submodule` embeds other repos at pinned commits — remember `--init --recursive`.

Recap

Productivity in Git is cumulative. Each alias saves a few keystrokes, each config flag removes a recurring annoyance, and each hook stops a class of mistakes from ever reaching the remote. Layer them deliberately. Put aliases and config in your global file so they follow you everywhere, manage hooks through a shared framework so the whole team benefits, and keep `stash`, `worktree`, and `submodule` in your toolkit for the moments when a single linear working directory is not enough.

Git in CI/CD & Tagging

Tags are how Git marks a single commit as a meaningful milestone — usually a release. Combined with a CI/CD pipeline, a single `git push` of a tag can compile artifacts, generate a changelog, and publish a GitHub Release without anyone touching a dashboard. This chapter covers semantic versioning conventions, automated changelog generation from your commit history, and the pipeline plumbing that turns a tag into a shipped release.

Semantic versioning and tags

Semantic Versioning (SemVer) gives version numbers a contract rather than an arbitrary count. A version looks like `MAJOR.MINOR.PATCH` — for example `v1.4.2` — and each segment communicates the *kind* of change since the previous release.

SEGMENT	INCREMENT WHEN…	EXAMPLE
MAJOR	You make incompatible API changes that break existing consumers.	`1.4.2` → `2.0.0`
MINOR	You add functionality in a backward-compatible way.	`1.4.2` → `1.5.0`
PATCH	You make backward-compatible bug fixes only.	`1.4.2` → `1.4.3`

Note the reset rule: bumping MINOR zeroes PATCH, and bumping MAJOR zeroes both MINOR and PATCH. The full format is `v<major>.<minor>.<patch>`, optionally followed by a pre-release suffix such as `-rc.1` or `-beta.2` (e.g. `v2.0.0-rc.1`).

> The leading `v` (as in `v1.4.2`) is a widely used convention but not part of the SemVer spec itself. Pick one style and apply it consistently — many tools let you configure the prefix.

Annotated vs lightweight tags

Git has two kinds of tags. A **lightweight** tag is just a named pointer to a commit. An **annotated** tag is a full object in the Git database, carrying a tagger name, email, date, a message, and an optional GPG signature.

	LIGHTWEIGHT	ANNOTATED
Command	`git tag v1.2.0`	`git tag -a v1.2.0 -m "msg"`
Stores author/date	No	Yes
Can be signed	No	Yes (`-s`)
Has a message	No	Yes
Recommended for releases	No	Yes

```
# Lightweight tag on the current commit
git tag v1.2.0

# Annotated tag (preferred for releases)
git tag -a v1.2.0 -m "Release 1.2.0"

# Annotated + signed tag
git tag -s v1.2.0 -m "Release 1.2.0"

# Tag a specific past commit
git tag -a v1.2.0 9fceb02 -m "Release 1.2.0"

# Inspect a tag
git show v1.2.0
```

> Always use annotated tags for releases. Tools like `git describe` default to annotated tags, and the embedded date and author make the release auditable later.

Pushing tags

Tags are **not** sent to the remote by a normal `git push` — you must push them explicitly.

```
# Push a single tag (preferred — explicit and CI-friendly)
git push origin v1.2.0

# Push every local tag the remote does not yet have
git push origin --tags

# Push only annotated tags reachable from pushed commits
git push --follow-tags

# Delete a tag locally, then on the remote
git tag -d v1.2.0
git push origin --delete v1.2.0
```

> Avoid moving or re-pushing a tag that has already been published. Other clones and CI caches will keep the old target, producing confusing "two different v1.2.0" situations. If a release was wrong, cut a new version (e.g. `v1.2.1`) instead of changing the existing tag.

Tags trigger CI releases

Most CI systems can listen specifically for tag events. Pushing `v1.2.0` becomes the signal that says "this exact commit is a release — go build and publish it." The next sections show how that signal flows through changelog generation and release automation.

Automated changelogs from Conventional Commits

Conventional Commits is a lightweight convention for commit messages that makes them machine-readable. The first line follows the shape `<type>(<scope>): <description>`, for example `feat(api): add pagination to search`. Because the *type* is structured, tooling can derive both the next version number and the changelog automatically.

COMMIT TYPE	VERSION BUMP	CHANGELOG SECTION
`fix:`	PATCH	Bug Fixes
`feat:`	MINOR	Features
`feat!:` or `BREAKING CHANGE:`	MAJOR	BREAKING CHANGES
`perf:`	PATCH	Performance
`docs:`, `chore:`, `refactor:`, `test:`	none (usually)	often omitted

A breaking change is flagged either with a `!` after the type/scope, or with a `BREAKING CHANGE:` footer in the commit body:

```
git commit -m "feat(auth)!: drop support for legacy tokens" \
           -m "BREAKING CHANGE: clients must migrate to OAuth2 before upgrading."
```

Tooling

- **semantic-release** — fully automated: analyzes commits, decides the next version, generates the changelog, tags, and publishes. No manual version bump at all.
- **standard-version** (and its successor **commit-and-tag-version**) — bumps the version and writes `CHANGELOG.md` locally; you push the tag yourself.
- **git-cliff** — a fast, configurable changelog generator (written in Rust) driven by a TOML config; language-agnostic.
- **release-please** — opens and maintains a "release PR" that accumulates changes; merging it cuts the release. Popular in GitHub-centric workflows.

```
# standard-version: bump version + update CHANGELOG.md, then push
npx standard-version
git push --follow-tags origin main

# git-cliff: generate a changelog for the whole history
git cliff --output CHANGELOG.md

# git-cliff: only the unreleased range since the latest tag
git cliff --unreleased --tag v1.3.0
```

A quick changelog with plain git log

When you do not need full tooling, `git log` alone produces a usable list of changes since the last tag:

```
# Range from the most recent tag to HEAD, one bullet per commit
git log "$(git describe --tags --abbrev=0)"..HEAD \
        --pretty=format:'- %s (%h)' --no-merges
```

> Conventional Commits turn your history into an API. The discipline of typing each commit is what lets a machine answer "what is the next version, and what changed?" with zero guesswork.

Release automation

The payoff is a pipeline that reacts to a tag push, builds artifacts, and publishes a release. Below is a minimal GitHub Actions workflow that fires only when a tag matching `v*` is pushed.

```
name: Release

on:
  push:
    tags:
      - 'v*'

permissions:
  contents: write   # required to create a GitHub Release

jobs:
  release:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0      # full history for changelog tooling
          fetch-tags: true    # ensure tags are present

      - name: Build artifacts
        run: |
          make build
          tar -czf app-${{ github.ref_name }}.tar.gz ./dist

      - name: Create GitHub Release
        uses: softprops/action-gh-release@v2
        with:
          files: app-${{ github.ref_name }}.tar.gz
          generate_release_notes: true
```

Here `${{ github.ref_name }}` resolves to the tag (for example `v1.2.0`), so the artifact is named after the version automatically. The `generate_release_notes` option asks GitHub to assemble notes from merged PRs since the previous tag.

Building artifacts and creating the release

The build step is whatever your project needs — compiling binaries, bundling a container image, producing a wheel or npm package. The release step uploads those files and publishes the GitHub Release. You can also create a release from the command line with the GitHub CLI, which is handy for scripts or other CI systems:

```
# Create a release for an existing tag and attach a built artifact
gh release create v1.2.0 ./dist/app-v1.2.0.tar.gz \
   --title "v1.2.0" \
   --notes-file CHANGELOG-latest.md
```

Shallow clones and fetching tags in CI

CI runners clone shallowly by default to save time and bandwidth — GitHub Actions `checkout` uses `fetch-depth: 1`, fetching only the tip commit and **no tags**. That breaks any tool that walks history or runs `git describe`.

NEED	SETTING
Full history (changelog tools, `git describe`)	`fetch-depth: 0`
Tags available in the working copy	`fetch-tags: true`
Default (fast, history-blind)	`fetch-depth: 1`

If you are deepening a checkout manually rather than via the action options:

```
# Turn a shallow clone into a full one and pull all tags
git fetch --unshallow --tags

# Or just fetch tags onto an existing clone
git fetch origin --tags
```

> The single most common CI release bug is a tool reporting "no names found, cannot describe anything" or producing an empty changelog. It is almost always a shallow clone missing history or tags. Set `fetch-depth: 0` and `fetch-tags: true` before reaching for anything more exotic.

> Keep the release pipeline idempotent: if a job re-runs for the same tag, it should overwrite or skip rather than fail. Tools like `gh release create` error if the release already exists — guard with `gh release view v1.2.0 || gh release create …`.

Recap

- **SemVer** encodes intent: PATCH for fixes, MINOR for backward-compatible features, MAJOR for breaking changes — and a bump resets the segments to its right.
- Use **annotated tags** (`git tag -a`) for releases, and push them explicitly with `git push origin v1.2.0` or `--follow-tags` .
- **Conventional Commits** let tools like semantic-release, standard-version, git-cliff, and release-please derive both the version bump and the changelog automatically.
- A **tag-triggered pipeline** turns one push into a built, published GitHub Release — but only if CI fetches enough history and tags (`fetch-depth: 0` , `fetch-tags: true`).

Conclusion

If you have worked through this book, Git is no longer a black box that occasionally swallows your work. You have built a mental model of how it actually thinks. You have also practised the moves that turn a tense, late-night "what just happened" into a calm, deliberate "here is exactly how I fix this."

What you should now be able to do

Look back at where you started and notice how much ground you have covered. You now carry with you:

- **A solid mental model.** Commits, trees, blobs, refs, the index, and the working tree are no longer mysterious. You understand that a branch is just a movable pointer and that almost everything in Git is an object addressed by its hash.
- **A fluent everyday workflow.** Staging, committing, branching, fetching, pulling, and pushing are muscle memory. You read `git status` and `git log` like a map rather than a riddle.
- **Confident integration.** Merge, rebase, and cherry-pick are tools you reach for on purpose, knowing the trade-offs of each and how to resolve the conflicts they surface.
- **Calm recovery.** A bad reset, a lost commit, a botched rebase, or a force-push gone wrong no longer means panic. You know to reach for `git reflog`, `git restore`, and `git revert`, and you trust that Git rarely throws anything away.
- **Clean commits and a sensible team workflow.** You write focused, well-described commits, keep history readable, and collaborate through a branching strategy your teammates can actually follow.
- **AI-assisted productivity.** You can lean on AI tools to draft commit messages, explain unfamiliar history, and suggest recovery paths, while keeping your own judgement firmly in charge.

The principles worth keeping

Tools change and commands accumulate flags, but a handful of principles will serve you for as long as you use version control:

1. **Understand before you run.** Especially for history-rewriting commands, know what a command does before you press Enter.
2. **Prefer safe, reversible operations.** Favour `git revert` over a hard `git reset` on anything you might regret.
3. **Never rewrite shared history.** Rebase and amend freely on your own branch; leave published commits alone.
4. **Commit small and often.** Small commits are easier to review, revert, and reason about.

5. **Write commits for your future self.** The person reading the log in six months will thank you.
6. **Automate the boring parts.** Aliases and hooks remove friction and prevent mistakes.
7. **Remember the reflog.** When something seems lost, it usually is not.

> If you remember only one sentence from this book, make it this: in Git, the work is almost never gone — you simply have not yet found the ref that still points to it.

Where to go next

Reading takes you only so far; fluency comes from doing. To keep growing:

- **Practise on real projects.** Use a throwaway repository to deliberately break things, then recover them.
- **Contribute to open source.** Nothing sharpens your workflow faster than collaborating with strangers across pull requests and rebases.
- **Read the official docs and man pages.** Try `git help <command>` and the free *Pro Git* book to deepen each topic.
- **Set up your aliases and hooks.** Tailor Git to the way you actually work.

```
git config --global alias.lg "log --oneline --graph --decorate --all"
git config --global alias.last "log -1 HEAD"
git help reflog
```

A closing word

Thank you for spending this time with the book. Git rewards patience and curiosity, and you have shown both. The next time a colleague stares at a tangled repository, you will be the calm one who quietly untangles it.

> This book is shared freely. If it saved you time or spared you a headache, please consider supporting it on the support page — every contribution helps keep it free and growing for the next developer.

Go build something, break it safely, and recover it with confidence. Happy committing.

Appendix

A dense, copy-paste-ready reference for the working developer. When you are mid-incident you do not want prose — you want the command. This appendix collects the commands, the conflict workflow, the "I broke it" recovery map, and sane starter configuration. With them you can fix the situation and get back to work.

> Throughout this appendix, replace anything written as `<branch>`, `<remote>`, `<sha>`, or `<file>` with your real values. Commands that rewrite history (`reset --hard`, `push --force`, `rebase`) are flagged — read the warning before running them on shared branches.

A. Master Command Reference

Grouped by area, then ordered roughly alphabetically within each group. The *Common example* column shows a realistic command, not just the bare verb.

Setup & Configuration

COMMAND	WHAT IT DOES	COMMON EXAMPLE
`git init`	Create a new repository in the current directory	`git init`
`git clone`	Copy a remote repository locally	`git clone git@github.com:org/repo.git`
`git config`	Read or set configuration values	`git config --global user.email you@example.com`
`git remote add`	Register a named remote URL	`git remote add origin <url>`
`git remote -v`	List configured remotes and URLs	`git remote -v`

Staging & Committing

COMMAND	WHAT IT DOES	COMMON EXAMPLE
`git add`	Stage changes for the next commit	`git add src/ README.md`
`git add -p`	Stage selected hunks interactively	`git add -p`
`git commit`	Record staged changes	`git commit -m "Fix login redirect"`
`git commit --amend`	Replace the last commit (message and/or content)	`git commit --amend --no-edit`
`git restore --staged`	Unstage a file, keeping working-tree edits	`git restore --staged <file>`
`git rm`	Remove a tracked file and stage the deletion	`git rm --cached secret.env`

Branching & Switching

COMMAND	WHAT IT DOES	COMMON EXAMPLE
`git branch`	List, create, or delete branches	`git branch -d <branch>`
`git switch`	Change the current branch (modern verb)	`git switch main`
`git switch -c`	Create and switch to a new branch	`git switch -c feature/api`
`git checkout`	Switch branches or restore files (legacy verb)	`git checkout <branch>`
`git merge`	Join another branch into the current one	`git merge feature/api`
`git rebase`	Replay commits onto a new base **(rewrites history)**	`git rebase main`

Syncing with Remotes

COMMAND	WHAT IT DOES	COMMON EXAMPLE
`git fetch`	Download remote objects without merging	`git fetch origin`
`git pull`	Fetch and integrate the remote branch	`git pull --rebase origin main`
`git push`	Upload local commits to a remote	`git push -u origin <branch>`
`git push --force-with-lease`	Force-push safely, aborting if the remote moved **(rewrites remote)**	`git push --force-with-lease`

History & Recovery

COMMAND	WHAT IT DOES	COMMON EXAMPLE
`git log`	Show commit history	`git log --oneline --graph --all`
`git reflog`	Show where HEAD has been (the safety net)	`git reflog`
`git reset --soft`	Move HEAD, keep index and working tree	`git reset --soft HEAD~1`
`git reset --mixed`	Move HEAD, reset index, keep working tree (default)	`git reset HEAD~1`
`git reset --hard`	Move HEAD and discard all changes **(destructive)**	`git reset --hard origin/main`
`git revert`	Create a new commit that undoes a prior one (safe)	`git revert <sha>`
`git cherry-pick`	Apply a specific commit onto the current branch	`git cherry-pick <sha>`
`git stash`	Shelve uncommitted changes	`git stash push -m "wip"`

Inspection & Debugging

COMMAND	WHAT IT DOES	COMMON EXAMPLE
`git status`	Show working-tree and staging state	`git status -sb`
`git diff`	Show changes between states	`git diff --staged`
`git show`	Display a commit or object	`git show <sha>`
`git blame`	Show who last changed each line	`git blame -L 10,20 <file>`
`git bisect`	Binary-search history for a bad commit	`git bisect start`

B. Conflict-Resolution Quick Guide

A conflict happens when two changes touch the same lines and Git cannot decide which to keep. It pauses the operation and asks you to choose. Stay calm — nothing is lost, and you can always abort.

The five-step workflow

1. Run `git status` to list the files marked **both modified**.
2. Open each conflicted file and find the marker blocks.
3. Edit the file so it contains the final, correct content and delete every marker line.
4. `git add <file>` to mark the conflict resolved.
5. Continue the operation with `--continue` (or commit, for a plain merge).

Reading the markers

```
<<<<<<< HEAD
your current branch's version of the lines
=======
the incoming branch's version of the lines
>>>>>>> feature/api
```

Everything between `<<<<<<<` and `=======` is **ours** (the branch you are on). Everything between `=======` and `>>>>>>>` is **theirs** (the branch being merged in). Keep one side, keep both, or write something new — then remove all three marker lines.

Taking one side wholesale

```
# Keep our version of a file
git checkout --ours <file>

# Keep their version of a file
git checkout --theirs <file>

# Then stage it
git add <file>

# Launch a configured visual merge tool instead
git mergetool
```

> During a **rebase**, the meaning of `--ours` and `--theirs` is inverted relative to a merge: "ours" is the branch you are replaying *onto*, and "theirs" is the commit being replayed. When in doubt, check with `git diff` before choosing a side.

Abort or continue, by operation

OPERATION	CONTINUE AFTER RESOLVING	ABORT AND UNDO EVERYTHING
Merge	`git merge --continue`	`git merge --abort`
Rebase	`git rebase --continue`	`git rebase --abort`
Cherry-pick	`git cherry-pick --continue`	`git cherry-pick --abort`
Revert	`git revert --continue`	`git revert --abort`

> To skip a single conflicting commit during a rebase or cherry-pick instead of resolving it, use `--skip`. Enable `rerere` (see section D) so Git remembers how you resolved a conflict and replays it automatically next time.

C. "I Broke It" Fix Lookup

Find the symptom on the left; run the command on the right. Before any destructive fix, `git reflog` shows every recent position of HEAD so you can recover a lost commit by its SHA.

I DID THIS BY MISTAKE	RECOVERY COMMAND
Committed to the wrong branch	`git switch <right-branch> && git cherry-pick <sha>`, then on the wrong branch `git reset --hard HEAD~1`
Want to undo the last commit but **keep** the changes staged	`git reset --soft HEAD~1`
Want to undo the last commit and unstage the changes	`git reset HEAD~1`
Amended a commit and lost the original	`git reflog`, find the pre-amend SHA, then `git reset --hard <sha>`
Ran `git reset --hard` and lost work	`git reflog`, then `git reset --hard <sha>` (or `git stash apply` if it was stashed)
Deleted a branch I still needed	`git reflog` to find its tip, then `git branch <name> <sha>`
Pushed a bad merge to a shared branch	`git revert -m 1 <merge-sha>` then push (safe; do not force-push shared history)
Committed a secret (password, key, token)	Rotate the secret immediately, then scrub history with `git filter-repo` and force-push
Made commits in detached HEAD and want to keep them	`git switch -c <new-branch>` before switching away
Discarded working changes I wanted back	`git stash list` / `git fsck --lost-found` (recovery is best-effort)

> A committed secret is compromised the moment it is pushed. Removing it from history does **not** un-leak it. Always rotate the credential first; scrub history second.

D. Recommended .gitconfig

A sane baseline for `~/.gitconfig`. Adjust the user block, then keep the rest.

```
[user]
    name = Your Name
    email = you@example.com

[core]
    editor = vim
    autocrlf = input          # normalize line endings on commit
    excludesfile = ~/.gitignore_global

[init]
    defaultBranch = main

[pull]
    rebase = true             # rebase instead of merge on pull

[push]
    default = simple
    autoSetupRemote = true    # auto-create upstream on first push

[rebase]
    autosquash = true         # honor fixup!/squash! commits
    autostash = true          # stash and reapply local changes

[rerere]
    enabled = true            # reuse recorded conflict resolutions

[merge]
    conflictstyle = zdiff3    # show common ancestor in conflicts

[alias]
    co = checkout
    sw = switch
    st = status -sb
    last = log -1 HEAD
    unstage = restore --staged
    lg = log --oneline --graph --all --decorate
    amend = commit --amend --no-edit
```

autoSetupRemote

Lets a bare `git push` create the upstream branch, so you can skip `-u origin <branch>`.

rerere.enabled

Records how you resolved a conflict and replays it the next time the same conflict appears — invaluable during long rebases.

merge.conflictstyle

`zdiff3` adds a base section so you can see the original lines both sides changed.

E. Recommended .gitignore Starters

Per-project `.gitignore` files live at the repository root and are committed so the whole team shares them. Below are minimal starters; expand per stack.

Node

```
node_modules/
npm-debug.log*
dist/
build/
.env
.env.local
coverage/
```

Python

```
__pycache__/
*.py[cod]
.venv/
venv/
*.egg-info/
.pytest_cache/
.mypy_cache/
.env
```

OS & editor noise

```
# macOS
.DS_Store
# Windows
Thumbs.db
# Editors
.vscode/
.idea/
*.swp
```

> OS and editor files belong in a **global** ignore, not in every project. Point Git at one with `git config --global core.excludesfile ~/.gitignore_global` and put the `.DS_Store`, `Thumbs.db`, and editor entries there. Keep the project `.gitignore` focused on build artifacts and dependencies specific to that codebase.

> Ignore rules do **not** apply to files that Git already tracks. If you added a file before ignoring it, stop tracking it with `git rm --cached <file>` and commit the removal — the working copy stays on disk.

Glossary

This glossary defines the essential Git terms used throughout the book. Each entry is short and practical. It focuses on what the term means when you are recovering from a problem or reasoning about a repository's state. Terms are grouped alphabetically; commands and identifiers appear in `monospace`.

B

Bare repository
A repository that has no working tree, containing only the contents of the `.git` directory. Bare repositories are typically used on servers as a shared push/fetch target, since there is no checkout to conflict with incoming changes. Created with `git init --bare`.

Blob
The Git object type that stores the raw contents of a file, with no filename or permissions attached. Identical file contents always produce the same blob, no matter where the file lives. That is how Git deduplicates data.

Branch
A movable named pointer to a commit, stored as a `ref` under `refs/heads/`. Committing on a branch advances that pointer to the new commit. Branches are cheap because they are just a file holding a 40-character object ID.

C

Checkout

The act of switching `HEAD` to a different branch or commit and updating the working tree to match it. The classic command is `git checkout`; modern Git splits its roles into `git switch` (branches) and `git restore` (files).

Cherry-pick

Applying the change introduced by one specific commit onto the current branch, creating a new commit with a new object ID. Run with `git cherry-pick <commit>`; useful for porting a single fix between branches without merging everything.

Clone

A full copy of a repository, including its complete history and all refs, created with `git clone`. The clone records where it came from as a remote named `origin`.

Commit

A snapshot of the entire project at a point in time, plus metadata: author, committer, message, and the parent commit(s). Each commit points to a `tree` and is identified by its `SHA-1` /object ID, forming the nodes of the history.

Conflict

A situation where Git cannot automatically reconcile two sets of changes to the same region of a file during a merge, rebase, cherry-pick, or revert. Conflicts are marked in the file with `<<<<<<<`, `=======`, and `>>>>>>>` markers that you must resolve by hand and then stage.

D

DAG (Directed Acyclic Graph)
The mathematical shape of Git history: commits are nodes, parent links are directed edges, and no commit can be its own ancestor (no cycles). Understanding history as a DAG explains how merges, branches, and the `merge base` work.

Detached HEAD
The state in which `HEAD` points directly at a commit instead of at a branch. New commits made here belong to no branch. They can be lost once you switch away, unless you create a branch or recover them via the `reflog`.

Diff
A line-by-line representation of the differences between two snapshots, such as two commits, the index and the working tree, or a commit and its parent. Produced by `git diff` and central to reviewing and applying changes.

F

Fast-forward
A merge in which the target branch is a direct ancestor of the branch being merged in, so Git simply moves the branch pointer forward rather than creating a merge commit. Disable it with `--no-ff` when you want an explicit merge commit.

Fetch
Downloading new commits, objects, and refs from a remote into your local repository. It updates your remote-tracking branches without touching your working tree or local branches. Run with `git fetch`.

Fork
A server-side copy of a repository owned by a different account. On hosting platforms, you use it to propose changes to a project you cannot push to directly. A fork is distinct from a local clone, though you typically clone your fork to work on it.

H

HEAD
A symbolic ref that names your current position in history, usually pointing at the currently checked-out branch (and through it, a commit). It is the starting point for new commits and the reference point for expressions like `HEAD~1` and `HEAD^`.

Hook
An executable script that Git runs automatically at a defined point in an operation, such as `pre-commit` or `pre-push`. Hooks live in `.git/hooks/` and let you enforce policies or automate checks.

I

Index (staging area)
The intermediate area, also called the *cache*, that holds the snapshot you are preparing for your next commit. `git add` moves changes from the working tree into the index, and `git commit` turns the index into a new commit.

M

Merge
Combining the histories of two branches into one, typically producing a merge commit with two parents. Run with `git merge`; if the changes overlap you may have to resolve a `conflict`.

Merge base
The most recent common ancestor of two commits in the DAG. Git uses it as the reference point for a three-way merge and for computing which changes each side introduced; find it with `git merge-base`.

O

Object
Any of the four content-addressed items Git stores in its database: `blob`, `tree`, `commit`, and `tag`. Every object is named by the `SHA-1` hash of its contents, making the store immutable and verifiable.

Origin
The conventional default name for the remote a repository was cloned from. It is just a name; `origin/main` is the remote-tracking branch recording where `main` was on that remote at the last fetch.

P

Packfile
A single compressed file that bundles many objects together with delta compression, used to store history efficiently and to transfer it over the network. Loose objects are periodically rolled into packfiles by `git gc`.

Pull
A combined operation that performs a `fetch` followed by an integration step, either `merge` (default) or `rebase` (with `--rebase`). Run with `git pull`; it updates your current branch with new upstream work.

Push
Sending your local commits and refs to a remote so others can see them, with `git push`. Pushes are rejected if they would not fast-forward the remote branch, unless you force the update.

R

Rebase

Reapplying a series of commits onto a new base commit, rewriting them as new commits with new object IDs to produce a linear history. Run with `git rebase`; interactive mode (`-i`) also lets you reorder, edit, or `squash` commits.

Ref

A human-readable name that points to a commit (or another ref), such as a branch, tag, or `HEAD`. Refs are stored under `.git/refs/` or in the packed-refs file.

Reflog

A local log recording every change to the tip of a ref, including `HEAD`, even after rebases, resets, or branch deletions. The reflog (`git reflog`) is your primary tool for recovering commits that seem lost.

Remote

A named reference to another copy of the repository, typically hosted elsewhere, that you fetch from and push to. Manage remotes with `git remote`; `origin` is the usual default.

Remote-tracking branch

A local, read-only pointer such as `origin/main` that records the last known position of a branch on a remote. It updates only when you fetch or push, and serves as the comparison point for your local branch.

Repository

The complete set of objects, refs, and configuration that make up a project's history, stored in the `.git` directory. A repository may or may not have an associated working tree (see *bare repository*).

Reset

Moving the current branch to point at a different commit and, depending on the mode, updating the index and working tree. `git reset --soft` moves only the branch, `--mixed` also resets the index, and `--hard` resets the working tree too.

Restore

The modern command (`git restore`) for discarding changes in the working tree or unstaging files from the index, without moving any branch. It separates file-restoration concerns from branch-switching.

Revert

Creating a new commit that undoes the changes of an earlier commit while preserving history, with `git revert`. Unlike `reset`, it is safe to use on commits that have already been pushed and shared.

S

SHA-1 / object ID
The 40-character hexadecimal hash that uniquely names every Git object based on its contents. Because the ID is derived from the data, any change produces a different ID, which is what makes history tamper-evident. Newer repositories may use SHA-256 instead.

Squash
Combining several commits into one, usually during an interactive rebase or a squash merge, to present a cleaner history. The original commit messages can be merged into a single message.

Stash
A temporary holding place for uncommitted changes so you can switch contexts with a clean working tree, managed by `git stash`. Restore the changes later with `git stash pop` or `git stash apply`.

Submodule
A repository embedded inside another repository at a fixed commit, recorded in a `.gitmodules` file. Submodules let you include external projects while pinning the exact version you depend on.

T

Tag (annotated vs lightweight)
A ref that marks a specific commit, commonly used for releases. A **lightweight** tag is just a name pointing at a commit, while an **annotated** tag is a full object storing a tagger, date, message, and an optional signature. Create one with `git tag -a`.

Tree
The Git object representing a directory: it lists filenames, modes, and the object IDs of the `blobs` and sub-`trees` it contains. Each commit points to exactly one top-level tree describing the whole snapshot.

U

Upstream
The remote-tracking branch that a local branch is configured to compare against and integrate from, set with `git branch --set-upstream-to` or implied by `git push -u`. It is what `git status` reports as "ahead" or "behind" and the default for a bare `git pull`.

W

Working tree / working directory
The checked-out files on disk that you actually edit, reflecting the snapshot of the currently checked-out commit plus any uncommitted changes. It is distinct from the `index` and the repository's object store.

Worktree
An additional working tree linked to the same repository, allowing you to check out different branches in separate directories simultaneously. Manage them with `git worktree add` and `git worktree list`.

Throughout the book, terms set in `monospace` within a definition cross-reference other entries in this glossary. When recovering from a problem, the most valuable concepts to learn are `HEAD`, the `index`, the `reflog`, and the difference between `reset` and `revert`.

References

No single book can hold the whole of Git. This page collects the resources that informed the scenarios in this book and that you should keep within reach as a working developer or maintainer: the official manuals you will consult daily, the deeper books that explain Git's internals, hands-on sites for building intuition, the conventions that keep a project's history readable, the workflows teams argue about, and the tools that make day-to-day Git safer and faster. Every entry includes a one-line note so you know why it is worth your time. URLs are given as plain text so you can copy them.

Official documentation

- **The Git reference manual / man pages** — run `git help` for an overview and `git help <command>` (for example `git help rebase`) for the authoritative description of any command, including every flag. Also online at git-scm.com/docs.
- **Pro Git, 2nd edition**, by Scott Chacon & Ben Straub — the canonical free book, read it in full at git-scm.com/book. Chapters 2, 3, and 7 (basics, branching, and tools) map closely to the recovery scenarios in this book.
- **gittutorial** — `git help tutorial`, a gentle first walk through the core commands for newcomers.
- **giteveryday** — `git help everyday`, a curated set of the ~20 commands that cover most real-world use.
- **gitglossary** — `git help glossary`, precise definitions of terms like *ref*, *tree-ish*, *detached HEAD*, and *fast-forward* that recur throughout this book.
- **gitworkflows** and **gitrevisions** — `git help workflows` and `git help revisions`, the official notes on recommended workflows and the full revision-naming syntax (such as `HEAD~3` and `main@{yesterday}`).

> The man pages ship with Git itself, so they always match your installed version. When this book and a web tutorial disagree, trust `git help <command>` for your machine.

Books

- **Pro Git**, Scott Chacon & Ben Straub (Apress) — free online and in print; the single most useful book to own alongside this one.
- **Version Control with Git**, Prem Kumar Ponuthorai & Jon Loeliger (O'Reilly) — a thorough, mechanism-first treatment that explains *why* Git behaves as it does, not just how.
- **Git Pocket Guide**, Richard E. Silverman (O'Reilly) — a compact, task-oriented reference for quick lookups at the desk.

- **Building Git**, James Coglan — reimplements Git from scratch in code; the best way to truly understand objects, refs, and the index.

Interactive learning

- **Learn Git Branching** (learngitbranching.js.org) — a visual, in-browser sandbox where commits animate as you type real commands; unmatched for building a mental model of branching, rebasing, and cherry-picking.
- **GitHub Skills** (skills.github.com) — short, guided courses run inside real repositories, covering branching, pull requests, and resolving merge conflicts.
- **Atlassian Git Tutorials** (atlassian.com/git/tutorials) — clear diagrams and step-by-step explanations of common workflows and commands.
- **git-scm.com/videos** — short official screencasts introducing the core concepts for those who learn better by watching.

> Pair Learn Git Branching with this book's recovery chapters: reproduce a "lost commit" scenario in the sandbox first, then practise the `git reflog` rescue with no fear of breaking anything real.

Conventions & specs

- **Conventional Commits** (conventionalcommits.org) — a lightweight specification for commit messages such as `feat:`, `fix:`, and `BREAKING CHANGE:` that let you automate changelogs and version bumps.
- **Semantic Versioning (SemVer)** (semver.org) — the MAJOR.MINOR.PATCH scheme that gives version numbers a shared, predictable meaning across the ecosystem.
- **"How to Write a Git Commit Message"**, Chris Beams (chris.beams.io/posts/git-commit) — the essay that popularised the 50/72 rule: a concise imperative subject under ~50 characters, a blank line, then a body wrapped at ~72.
- **Keep a Changelog** (keepachangelog.com) — a simple convention for human-readable CHANGELOG files, organised by version with *Added*, *Changed*, *Fixed*, and *Removed* sections.

Workflows

- **"A successful Git branching model" (Git Flow)**, Vincent Driessen (nvie.com/posts/a-successful-git-branching-model) — the original article describing the `develop` / `release` / `hotfix` branching strategy; note Driessen's own later caveat that it suits versioned releases more than continuous delivery.

- **GitHub Flow** (docs.github.com/get-started/quickstart/github-flow) — a deliberately minimal model: branch off `main`, open a pull request, merge, deploy.
- **GitLab Flow** (about.gitlab.com, search "GitLab Flow") — extends GitHub Flow with environment and release branches for teams that need staged deployments.
- **Trunk-Based Development** (trunkbaseddevelopment.com) — Paul Hammant's site advocating short-lived branches and frequent integration into a single trunk, the foundation of continuous integration at scale.

Tools

- **git-filter-repo** (github.com/newren/git-filter-repo) — the Git project's recommended tool for rewriting history, for example to remove a leaked secret or a large file across every commit; far faster and safer than `git filter-branch`.
- **pre-commit** (pre-commit.com) — a framework for managing Git hooks across languages, so linters and formatters run automatically before each commit.
- **husky** (typicode.github.io/husky) — the popular way to wire Git hooks into JavaScript and Node.js projects via package.json.
- **semantic-release** (semantic-release.gitbook.io), **standard-version**, and **git-cliff** (git-cliff.org) — automate versioning and changelog generation from Conventional Commits; git-cliff is a fast, highly configurable changelog generator written in Rust.
- **GitHub CLI (gh)** (cli.github.com) — brings pull requests, issues, and releases to the terminal so you rarely need to leave it.
- **AI-assisted tooling** — **GitHub Copilot** (github.com/features/copilot) and **Claude Code** (Anthropic's official CLI, docs.claude.com) assist with code and commands directly in your workflow, while **aicommits** (github.com/Nutlope/aicommits) and **opencommit** (github.com/di-sukharev/opencommit) generate commit messages from your staged diff.

> Treat history-rewriting tools with respect: `git-filter-repo` changes commit hashes for everyone. Always work on a fresh clone, coordinate with collaborators, and keep a backup ref before you run it.

Software moves quickly, so prefer the official sites above to mirrored copies, and re-check tool documentation against the version you have installed. When in doubt about any command in this book, the fastest authoritative answer is always one terminal away: `git help <command>`.

Index

N

O

P

R

S

T

U

W

Writing prompts that produce good messages 6.2

About the Author

Milad Golfam

Milad Golfam is a senior software developer and engineering leader with over a decade of experience building scalable, high-availability backend systems. As CTO of Safarmarket.com, he has architected microservices that handle tens of thousands of concurrent requests per second at sub-200ms response times, across Java, Golang, Python, and Node.js.

His work sits where software engineering meets mathematics. He holds a Master's in Computer Engineering, an MBA, and a Master's in Computational and Applied Mathematics — and it is that blend, production systems on one side and applied math on the other, that shapes his practical, example-first way of writing.

He writes and publishes open, freely available books and tools for people who learn by doing — with prose, formulas, runnable code, and real-world usage in every chapter. You can find more of his work, and reach out to say hello, at his website below.

🌐 **migolfam.com**