

commit · branch · pull request · review · merge · ship

FROM ZERO TO TEAM WORKFLOW

Git and GitHub for Teams

Branching, Pull Requests, and CI

Commits, code review, protected branches,
and GitHub Actions — explained step by step

GIT · GITHUB · PULL REQUESTS · ACTIONS · RELEASES

THENEXBOOKS

Git and GitHub from Zero to Team Workflow

Branching, Pull Requests, and CI From Your First Commit to Protected Branches, Code Review, and GitHub Actions

First Edition

THENEXBOOKS

Git and GitHub from Zero to Team Workflow
Branching, Pull Requests, and CI

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code and configuration examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Git evolves steadily, and GitHub's features, settings, and plan availability change often; always consult the official documentation for the versions and plans you use.

Git and the Git logo are trademarks of Software Freedom Conservancy. GitHub, GitHub Actions, Dependabot, and CodeQL are trademarks of GitHub, Inc. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with or endorsed by GitHub, Inc. or the Git project.

Quillnote, the running example in this book, and its team members are fictional. Any resemblance to real projects or people is coincidental.

Preface

Git is the version control system you are most likely to meet on a software team, and GitHub is where a great many of those teams keep their code. Yet many developers learn Git one panicked search at a time: a command copied here, a conflict "fixed" there, a force push that quietly erased a colleague's work. They can commit and push, but branching feels risky, rebasing feels dangerous, and the team's pull-request and CI rules feel like obstacles rather than safety nets.

It does not have to be that way. Git is built on a small number of simple ideas — snapshots, pointers, and a history you can always inspect — and once you understand them, every command starts to make sense. GitHub then adds the collaboration layer: pull requests, code review, protected branches, and automation with GitHub Actions. Together they form the workflow that modern teams use to ship software safely every day.

This book takes you from zero to that team workflow. We start with your very first repository and build up, step by step, to branching strategies, pull requests, code review, protected branches, releases, and a complete continuous integration pipeline. Along the way we explain what Git is doing underneath, so that when something goes wrong you know how to fix it instead of starting over.

Who This Book Is For

- **Beginners** who have never used version control, or who have only used it through an editor's buttons.
- **Developers** who use Git daily but want to stop fearing merges, rebases, and "detached HEAD".
- **New team members** who need to learn pull requests, reviews, and CI quickly.
- **Tech leads and maintainers** setting up a repository, branch protections, and GitHub Actions for a team or an open-source project.

What You Will Learn

- How Git thinks: the working tree, the staging area, commits, branches, and references.
- Everyday Git: committing, exploring history, branching, merging, rebasing, and undoing mistakes safely.
- Collaboration on GitHub: remotes, SSH keys, the `gh` CLI, issues, projects, pull requests, and code review.
- Team workflows: branching strategies, protected branches and rulesets, CODEOWNERS, commit conventions, releases, and forks.
- Continuous integration with GitHub Actions: pipelines, matrices, caching, reusable workflows, secrets, environments, and deployments.
- Supply-chain security, hooks, troubleshooting, and two end-to-end case studies.

How This Book Is Organized

Part I — Git Foundations (Chapters 1–5) teaches the core model and daily commands. **Part II — Branching, Merging, and Rewriting** (Chapters 6–10) covers branches, merges, rebases, undoing, and power tools. **Part III — GitHub and Remote Collaboration** (Chapters 11–15) connects your repository to GitHub and to other people. **Part IV — Team Workflows** (Chapters 16–20) turns individual skills into a team process. **Part V — Continuous Integration with GitHub Actions** (Chapters 21–25) automates testing, releases, and security. **Part VI — Mastery and Case Studies** (Chapters 26–28) covers hooks and troubleshooting, then walks through two complete case studies. The appendices provide a cheat sheet, templates, a team workflow checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Quillnote

Throughout the book we follow **Quillnote**, a fictional open-source Markdown note-taking app written in TypeScript. It starts as a solo project on one laptop and grows into a GitHub repository with a small team of maintainers — Maya, Tomás, Aiko, and Priya — outside contributors who work through forks, a CI pipeline, and regular releases. Recurring work items such as issue #42 ("Add dark mode") and issue #57 ("Sync conflict when two devices edit the same note") let you see the same change travel from an idea to a released version.

Versions and Conventions

Examples use recent versions of Git and assume a default branch named `main`. GitHub features are described by what they do and by their setting names; where a feature depends on your plan or repository visibility, or was in preview at the time of writing, we say so.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version details, or a useful aside.

> **WARNING**
> A common mistake or a command that can lose work or break things for your team.

Terminal sessions show commands after a `$` prompt, followed by their output. Commit hashes are abbreviated, and long output is trimmed with `...`.

Contents

Preface

PART I — GIT FOUNDATIONS

1 Why Version Control, and How Git Thinks

2 Installing and Configuring Git

3 Your First Repository: init, add, commit, status, and log

4 Inside Git: Objects, Trees, Commits, and References

5 Exploring History: log, diff, show, and blame

PART II — BRANCHING, MERGING, AND REWRITING

6 Branches Explained: Cheap, Fast, and Everywhere

7 Merging and Resolving Conflicts

8 Rebasing: Clean, Linear History Done Safely

9 Undoing Things: restore, reset, revert, and reflog

10 Power Tools: stash, cherry-pick, tags, bisect, and worktrees

PART III — GITHUB AND REMOTE COLLABORATION

11 Remotes: clone, fetch, pull, and push

12 Getting Started with GitHub: Accounts, SSH Keys, Repositories, and the gh CLI

13 Planning Work on GitHub: Issues, Labels, Milestones, and Projects

14 Pull Requests from First Commit to Merge

15 Code Review That Works: Giving and Receiving Feedback

PART IV — TEAM WORKFLOWS

16 Choosing a Branching Strategy: GitHub Flow, Git Flow, and Trunk-Based Development

17 Protecting Main: Branch Protection, Rulesets, CODEOWNERS, and Merge Queues

18 Commit Conventions, Semantic Versioning, and Release Notes

19 Forks and Open Source Contribution

20 Scaling Repositories: Monorepos, Submodules, Large Files, and Performance

PART V — CONTINUOUS INTEGRATION WITH GITHUB ACTIONS

21 GitHub Actions Fundamentals: Workflows, Jobs, Steps, and Runners

22 Building a CI Pipeline: Lint, Test, Build, and Matrix Strategies

23 Faster and Reusable Pipelines: Caching, Artifacts, Reusable Workflows, and Composite Actions

24 Secrets, Environments, OIDC, and Deployments

25 Securing the Supply Chain: Dependabot, Code Scanning, Secret Scanning, and Signed Commits

PART VI — MASTERY AND CASE STUDIES

26 Hooks, Automation, and Troubleshooting Hard Problems

27 Case Study: Quillnote from Solo Project to Team Workflow

28 Case Study: Release Day at Quillnote — From Pull Request to Production

APPENDICES

A Git and gh Cheat Sheet

B Templates and Reference

C Team Workflow Checklist

D Glossary

PART I

Git Foundations

How version control works, how to set Git up properly, how to make your first commits, and what Git is really storing underneath — the mental model that makes every later command make sense.

1 · Why Version Control
2 · Installing and Configuring Git
3 · Your First Repository
4 · Inside Git
5 · Exploring History

CHAPTER 1 · PART I — GIT FOUNDATIONS

Why Version Control, and How Git Thinks

Before you type a single Git command, it pays to understand the problem Git solves and the mental model it uses to solve it. This chapter explains why version control exists, how Git differs from the tools that came before it, what "snapshots" and "the three areas" really mean, and how Git relates to GitHub. It also introduces Quillnote, the project we will grow from a one-person experiment into a team with pull requests, reviews, and continuous integration.

The Problem Every Project Eventually Has

Almost everyone who has written a document, a spreadsheet, or a program has seen a folder that looks something like this:

```
$ ls
notes-app.zip
notes-app-backup.zip
notes-app-final.zip
notes-app-final-v2.zip
notes-app-final-v2-FIXED.zip
notes-app-final-v2-FIXED-dont-touch.zip
```

Each file is an attempt to answer questions that every project eventually asks. What did this look like before I broke it? Which version did I send to the client? What changed between Tuesday and today, and why? When two people edited the same file, whose changes won? The file-naming approach answers none of these reliably. The names drift, the "final" version is never final, and nobody remembers what the difference between `v2` and `v2-FIXED` actually was.

Things get worse as soon as a second person joins. Now you are emailing archives back and forth, or sharing a synced folder and hoping two people never save the same file at once. Someone overwrites someone else's work, and the only record of what happened is a confused chat thread.

A **version control system** (VCS) replaces this chaos with a structured, queryable history. Instead of copying the whole project and inventing a file name, you record a *version* — Git calls it a **commit** — along with a message describing what changed and why, who made the change, and when. The system stores every version efficiently, lets you compare any two of them, and lets you return to any earlier state in seconds.

What version control actually gives you

Each of these benefits shows up again and again in this book:

- **A complete history.** Every recorded change is kept, with its author, date, and message. You can ask "what did this file look like three months ago?" and get an exact answer.
- **Safe experimentation.** Because you can always return to a known-good state, you can try risky changes without fear. If the experiment fails, you throw it away.
- **Parallel lines of work.** Branches (Chapter 6) let you develop a new feature while a colleague fixes a bug, without either of you disturbing the other, and then combine the results.
- **Collaboration without overwriting.** The system knows how to combine changes from different people and tells you clearly when two changes genuinely conflict (Chapter 7).
- **Accountability and understanding.** When a line of code looks strange, you can find the commit that introduced it and read the explanation its author left behind (Chapter 5).
- **A foundation for automation.** Once every change is a recorded, identifiable event, tools can react to it: running tests, building releases, and deploying software. That is what Part V of this book is about.

Version control is so fundamental to modern software development that most open-source projects and deployment platforms assume you already use it — and today that usually means Git.

A Short History: Local, Centralized, Distributed

Version control has gone through three broad generations. Understanding them helps explain why Git behaves the way it does, especially the parts that surprise newcomers.

Local version control

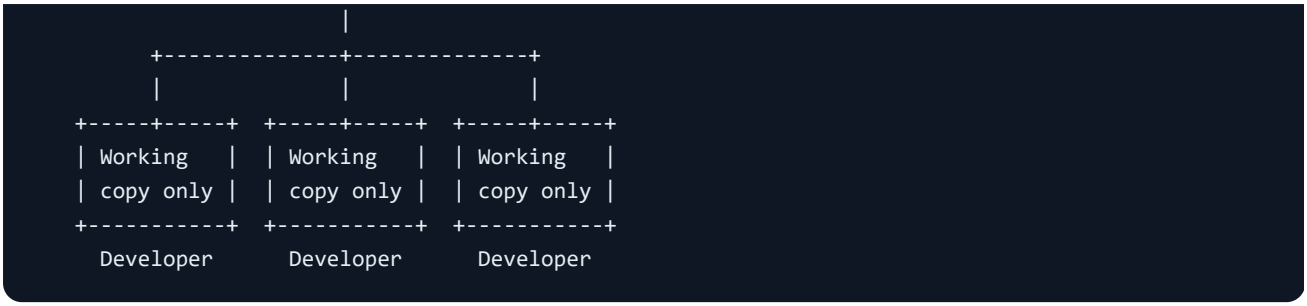
The earliest tools kept a history of individual files on a single machine. They beat copying files by hand, but did nothing to help people on different computers work together.

Centralized version control

The next generation — tools such as CVS and Subversion — introduced a single central server that holds the entire history. Each developer "checks out" a working copy of the latest files, edits them, and "commits" changes back to the server. Everyone can see what everyone else is doing, and administrators have one place to manage access.

The weakness is that the central server is the only place the full history lives. If it is unreachable, you cannot commit, browse history, or compare old versions. If it is lost without a backup, the history is lost with it. And because almost every operation needs the network, everything feels slower than it should.





Distributed version control

Distributed systems — Git and Mercurial are the best-known examples — take a different approach. When you get a copy of a project, you do not just get the latest files; you get the *entire repository*, including every commit in its history. Your laptop holds a complete, independent repository. You can commit, branch, view history, and compare versions with no network connection at all.

Collaboration happens by exchanging commits between repositories. You **push** your commits to another repository and **fetch** or **pull** commits from it (Chapter 11). In practice, teams still agree on one shared repository — usually hosted on a service such as GitHub — that acts as the official meeting point. But that is a social convention, not a technical requirement. Every clone is a full backup.



Git itself was created in 2005 by the Linux kernel community, who needed a fast, distributed tool that could handle a very large project with thousands of contributors. Speed, strong data integrity, and excellent support for many parallel branches were design goals from day one. Those goals explain a lot about Git's personality: it is fast because almost everything is local, it is strict about integrity because every piece of data is identified by a cryptographic hash, and it makes branching cheap because a large project cannot function without it.

Aspect	Centralized	Distributed (Git)
Where history lives	One server	Every clone
Commit offline	No	Yes

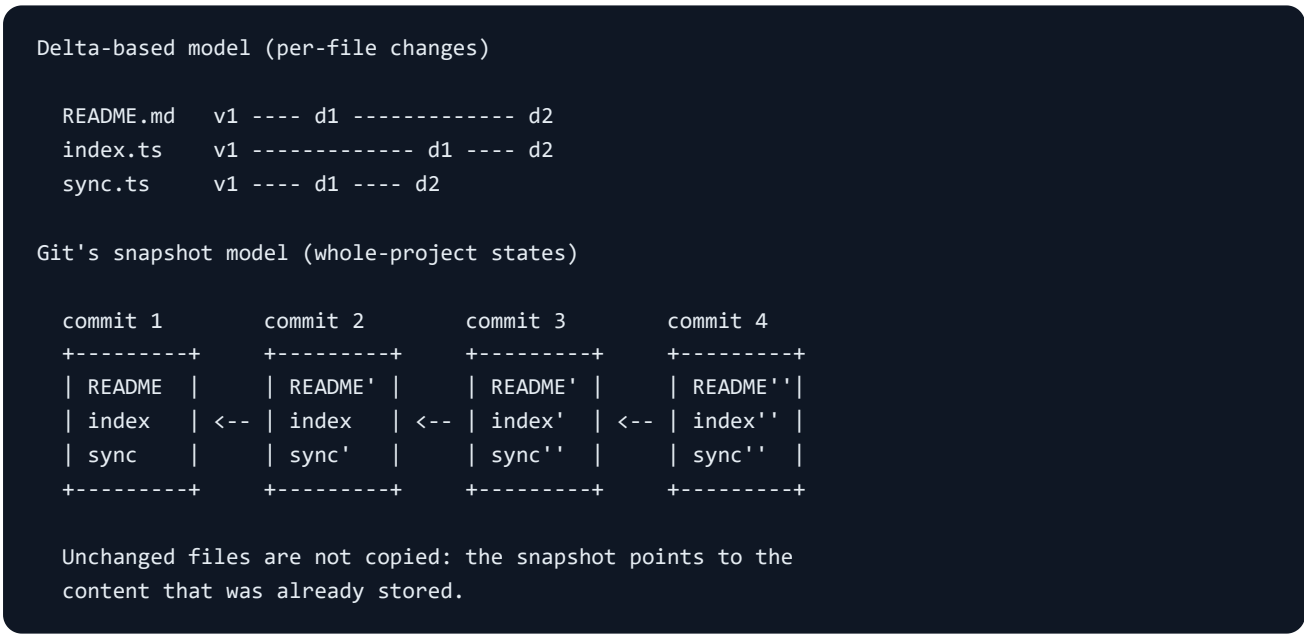
View history offline	Usually no	Yes
Speed of common tasks	Network-bound	Mostly local, fast
Branching	Often heavy	Cheap and routine
Single point of failure	The server	None by design
Access control	Built into server	Provided by host

> **NOTE**
> "Distributed" does not mean "no central repository." Nearly every team designates one repository — on GitHub, in our case — as the source of truth. The difference is that your local repository is a full peer, not a thin client. You can do real work on a plane, and a lost server does not mean lost history.

How Git Thinks: Snapshots, Not Differences

Many earlier systems stored history as a list of changes to files: "version 1 of `README.md`, then this diff, then that diff." To reconstruct a version, the system replays the diffs in order. This is sometimes called *delta-based* version control.

Git's model is conceptually different. Each commit records a **snapshot** of what every tracked file in the project looked like at that moment. If a file did not change between two commits, Git does not store it again; the new snapshot simply points to the identical content already stored. The mental picture is a sequence of complete project states, each linked to the one before it.



Why does this matter to you as a user? Because it shapes how commands behave. When you ask Git for the difference between two commits, it is not reading stored diffs; it compares two snapshots and computes the difference on the spot. When you switch to a branch, Git replaces the files in your folder with the snapshot that branch points to. When you think about "what is in this commit," the right answer is "the entire project as it was," not "a few changed lines."

The snapshot model is a *conceptual* model. Internally, Git is still careful about disk space: it compresses content and, when it packs objects together, it stores similar content as deltas against each other. Chapter 4 opens the hood and shows exactly how snapshots are built from blobs and trees, and how packfiles keep the repository small. For everyday work, though, "a commit is a snapshot" is the model to keep in your head.

Every snapshot is identified by its content

Git gives every piece of data it stores an identifier computed from the content itself: a cryptographic hash. A commit's ID, such as `3f9c2a1e7b4d...`, is a hash of the commit's contents, which include the hash of its snapshot and the hash of its parent commit. Change a single byte anywhere in the history and every hash from that point onward changes.

This gives Git a strong integrity guarantee. You cannot silently corrupt or tamper with a file, a snapshot, or a piece of history without Git noticing, because the IDs would no longer match their contents. It also means commit IDs are globally meaningful: if you and a teammate both have a commit with the same ID, you both have exactly the same snapshot and exactly the same history leading up to it.

By default Git uses SHA-1 hashes, displayed as 40 hexadecimal characters. Recent versions of Git can also create repositories that use SHA-256 instead, although at the time of writing SHA-1 repositories remain the norm and hosting services generally expect them. You will almost always see hashes abbreviated to the first seven or so characters, which is enough to be unique in most repositories. Chapter 4 explores hashes in depth.

Nearly every operation is local

Because your clone contains the complete history, the vast majority of Git commands never touch the network. Viewing the log, comparing versions, creating branches, committing, and switching between branches all read and write files in a hidden folder inside your project. Only a small set of commands — mainly `clone`, `fetch`, `pull`, and `push` — talk to other repositories. When something feels instantaneous in Git, this is why.

It also means there is a clear moment when your work becomes visible to others: when you push it. Until then, your commits are private to your machine. This is liberating — you can commit messy work-in-progress, reorganize it, and only share it when it is ready — but it also means an unpushed commit exists in exactly one place. A laptop that dies with a week of unpushed work takes that week with it.

> **WARNING**
>
> A commit is not a backup until it has been pushed somewhere else. Get into the habit of pushing your branches regularly, even unfinished ones, once you start working with a remote in Chapter 11. "It is committed" and "it is safe" are not the same statement.

The Three Areas: Working Tree, Staging Area, Repository

If you remember only one diagram from this chapter, make it this one. Git manages your project in three distinct places, and almost every command moves content from one of them to another.



The working tree

The **working tree** (also called the working directory) is the ordinary folder of files you open in your editor. It holds one particular version of the project — usually the latest commit on the branch you are on — plus whatever edits you have made since. Git watches it but does not record anything in it automatically. You can edit, create, and delete files freely; nothing is saved into history until you ask.

The staging area

The **staging area**, which Git's own documentation usually calls **the index**, is where you assemble the next commit. Think of it as a draft of the next snapshot. When you run `git add`, you copy the current content of a file from the working tree into the staging area. When you run `git commit`, Git takes whatever is in the staging area — exactly that, and nothing else — and records it as a new snapshot.

Newcomers often ask why this middle step exists. Why not just commit whatever is in the folder? The answer is that the staging area lets you craft commits deliberately. Suppose you fixed a bug and, while you were there, also reformatted an unrelated file and started a new feature. With a staging area, you

can stage only the bug fix, commit it with a clear message, then stage and commit the other changes separately. You can even stage part of a file, as Chapter 3 shows with `git add -p`. Clean, focused commits make history easier to read, review, and undo.

The repository

The **repository** is the database of every snapshot you have committed, along with branches, tags, and configuration. It lives in a hidden directory named `.git` at the top of your project. Everything that makes a folder a Git repository is inside `.git`; delete that directory and you are left with ordinary files and no history. Chapter 4 takes a guided tour of its contents.

> **WARNING**
>
> Never delete, move, or hand-edit files inside `.git` unless you know exactly what you are doing. It is the entire history of your project. If you want a fresh start, move it aside rather than deleting it, and only once you are sure nothing in it matters.

The states of a file

From the three areas follow the states a file can be in. Git first divides files into **untracked** (Git has never been told about them) and **tracked** (they were in the last snapshot or have been staged). Tracked files are then in one of three states:

State	Meaning	How it gets there
Unmodified	Same as last commit	After a commit
Modified	Changed, not staged	You edit the file
Staged	Change is in the index	`git add`
Committed	Stored in history	`git commit`

A single file can be in more than one state at once. If you edit `README.md`, stage it, and then edit it again, the staged version and the working-tree version differ. `git status` will list the file under both "Changes to be committed" and "Changes not staged for commit." That is not a bug; it is the staging area doing exactly its job. The commit will contain the staged version, not your latest edit, until you run `git add` again.

A first look at the three areas in action

You will install Git in Chapter 2 and start using it properly in Chapter 3, but a short preview makes the three areas concrete. Here a brand-new repository gains a `README.md` file, and `git status` reports where it stands at each step:

```
$ git init quillnote
Initialized empty Git repository in /home/dev/quillnote/.git/
$ cd quillnote
```

```
$ echo "# Quillnote" > README.md
$ git status
On branch main

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)
```

The file exists only in the working tree. Git sees it but is not tracking it. Now we stage it:

```
$ git add README.md
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   README.md
```

The file's content is now in the staging area, ready to become part of the first snapshot. Finally we commit:

```
$ git commit -m "Add README with project name"
[main (root-commit) 3f9c2a1] Add README with project name
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
$ git status
On branch main
nothing to commit, working tree clean
```

The snapshot is stored in the repository with the abbreviated ID `3f9c2a1`. All three areas now agree, which is what "working tree clean" means. Your exact output may differ slightly — for example, the branch might be called `master` if you have not yet configured `init.defaultBranch`, which Chapter 2 covers.

> **TIP**
>
> Run `git status` constantly, especially while you are learning. It is read-only, fast, and its output even suggests the commands you are likely to need next. Experienced developers run it dozens of times a day.

Git Is Not GitHub

One of the most common sources of confusion for newcomers is the relationship between Git and GitHub. They are related, but they are not the same thing, and mixing them up leads to muddled thinking about where your work lives.

Git is the version control tool: free, open-source software that runs on your computer. It manages the working tree, the staging area, and the repository. It works entirely without an internet connection and without any account anywhere. You could use Git for your whole career without ever touching GitHub.

GitHub is a hosting service built around Git. It stores copies of Git repositories on its servers and adds a web interface and collaboration features on top: pull requests for proposing and reviewing changes, issues for tracking work, project boards, access control for organizations and teams, and GitHub Actions for running automated workflows. Other services such as GitLab and Bitbucket play a similar role, and many companies run their own Git servers. The Git skills you learn in Parts I and II transfer directly to all of them.

Question	Git	GitHub
What is it?	Version control tool	Hosting and collaboration
Where does it run?	Your computer	Web service
Needs an account?	No	Yes
Works offline?	Yes	No
Commits, branches	Yes (core)	Stores and displays them
Pull requests	No	Yes
Issues, projects	No	Yes
CI (Actions)	No	Yes

A useful way to think about it: Git is the engine, and GitHub is one of several garages where teams park their engines and work on them together. A "pull request" is a GitHub concept layered on top of ordinary Git branches; under the hood, merging a pull request creates ordinary Git commits that you can inspect with ordinary Git commands. When something goes wrong in a pull request, the fix is almost always a Git operation on your machine followed by a push.

This book deliberately teaches Git first. Parts I and II are about Git alone, on your own machine. Part III introduces remotes and GitHub, Part IV covers how teams organize their work on GitHub, and Part V builds continuous integration with GitHub Actions. By the time you open your first pull request in Chapter 14, you will understand exactly what is happening to the commits inside it.

> **NOTE**
>
> GitHub also offers a desktop application and many editors include Git integration with buttons for staging and committing. These are front ends that run Git commands for you. Everything in this book uses the command line because it works the same everywhere, shows you exactly what is happening, and makes the graphical tools easy to understand later.

Common Misconceptions to Unlearn Early

A few beliefs trip up nearly every Git beginner. Clearing them up now will save you confusion later.

"Saving the file saves it in Git"

Saving in your editor only changes the working tree. Git records nothing until you stage and commit. If you delete an uncommitted file, Git cannot bring it back, because it never had a copy. Commit early and often; small commits cost nothing.

"A commit stores my changes"

As we saw, a commit stores a snapshot of the whole project, plus a pointer to its parent. The "changes" you see in `git show` are computed by comparing that snapshot to its parent. This distinction becomes important when you start rebasing and cherry-picking in Part II: those commands create new snapshots, which is why the resulting commits get new IDs.

"Git and GitHub are the same thing"

We covered this above, but it bears repeating because it causes real mistakes: people assume a commit is "on GitHub" as soon as they make it, or that deleting a repository on GitHub deletes their local work. Neither is true. Local and remote repositories are separate copies that you synchronize deliberately.

"Branches are expensive, so I should avoid them"

In Git, a branch is a tiny file containing a commit ID, so creating one is essentially instantaneous. Chapter 6 shows why teams create and delete branches constantly.

"If I mess up, the history is ruined"

Git is remarkably forgiving. Once something has been committed, it is very hard to lose, even if you reset, rebase, or delete a branch by mistake. Chapter 9 covers the tools — `git restore`, `git reset`, `git revert`, and the reflog — that let you recover from almost anything. The main thing Git cannot recover is work you never committed.

What Goes Wrong on Real Teams

Many team problems with Git trace back to the fundamentals in this chapter:

- **Giant, mixed commits.** Someone works for a week and commits everything at once with the message "updates." Nobody can review it, nobody can understand it later, and undoing one part means undoing all of it. The staging area exists precisely to prevent this.
- **Work that only exists on one laptop.** A developer commits diligently but never pushes. Then the machine is lost, or they go on holiday and a colleague needs the branch. Local commits are real, but they are not shared or backed up.
- **Treating the hosting service as the tool.** Teams that only ever click buttons in a web interface are stuck the moment something unusual happens, such as a conflict the web editor cannot resolve. Understanding Git itself removes that ceiling.
- **Committing things that should never be in history.** Passwords, API keys, large generated files, and personal editor settings end up in commits and then in every clone. Because history is shared and durable, removing them later is painful. Chapter 3 introduces `.gitignore` to prevent this, and Chapter 25 covers secret scanning.

Each of these is avoidable with habits you will build in the next few chapters.

Meet Quillnote

Abstract explanations only go so far. To make every chapter concrete, this book follows one project from its first commit to a mature, team-run open-source application with automated releases. That project is **Quillnote**.

Quillnote is a fictional, open-source, cross-platform note-taking app that stores notes as Markdown files. It is written in TypeScript and runs on Node.js. It has an editor component, a sync engine that keeps notes consistent across devices, and a small command-line interface. On GitHub, it lives in a fictional organization as the repository `quillnote/quillnote`. When it is fully grown, its layout looks like this:

```
quillnote/
├── .github/
│   └── workflows/        CI and release automation (Part V)
├── docs/                 User and contributor documentation
├── src/
│   ├── cli/              Command-line interface
│   ├── editor/           Markdown editor and UI
│   └── sync/             Device synchronization engine
├── tests/                Automated tests
├── CHANGELOG.md
├── README.md
└── package.json
```

You do not need to know TypeScript to follow along. The code in Quillnote's files is deliberately small, and what matters is how the project's history is created, shaped, shared, and automated. Any language would work the same way.

How Quillnote grows through the book

The project evolves alongside your skills:

- **Part I (Chapters 1–5):** Quillnote is a solo project. You create the repository, make its first commits, look inside the `.git` directory, and learn to explore history.
- **Part II (Chapters 6–10):** still working alone, you use branches such as `feature/dark-mode`, merge and rebase them, resolve conflicts, and learn to undo mistakes safely.
- **Parts III and IV (Chapters 11–20):** Quillnote moves to GitHub and gains a small team. Work is tracked in issues, proposed in pull requests, reviewed, and protected by branch rules. Outside contributors arrive through forks.
- **Parts V and VI (Chapters 21–28):** the project adopts continuous integration with GitHub Actions, tightens its supply-chain security, and ships tagged releases such as `v1.0.0` and `v1.1.0`.

The Quillnote team

From Part III onward, four recurring (and entirely fictional) people work on Quillnote. You will see their names in commit authors, pull request reviews, and examples:

Name	Role	Typical work
Maya	Maintainer	Reviews, merges, releases
Tomás	Backend developer	The sync engine
Aiko	Frontend developer	Editor and UI
Priya	Docs contributor	Docs; later a maintainer

Two pieces of work will follow us through many chapters. Issue **#42, "Add dark mode,"** is a feature Aiko builds on the branch `feature/dark-mode`. Issue **#57, "Sync conflict when two devices edit the same note,"** is a bug Tomás fixes on `fix/sync-conflict`. Example email addresses always use the reserved `example.com` domain, such as `maya@example.com`.

In Part I, though, it is just you. The commits you make in Chapter 3 become the foundation that the whole team builds on later, and the habits you form now — small commits, clear messages, a tidy `.gitignore` — are the habits the team will depend on.

How to Read the Examples in This Book

A few conventions will make the examples easier to follow:

- Terminal sessions show commands prefixed with `$` and their output below. Do not type the dollar sign.