GIT PUSH STUDIO

The Git Field Guide

Master Git Offline — From Your First Commit to Everyday Mastery

$9 · PDF + ePub

by Ajit Waghare · Version 1.0.0

Contents

The Git Field Guide

Master Git Offline — From Your First Commit to Everyday Mastery

by Ajit Waghare Git Push Studio

About This Guide

Git is the version control system that the modern software world runs on. It powers open-source projects, enterprise codebases, and everything in between. Yet for many developers it remains a tool they use by muscle memory — a handful of commands they copy from blogs, hoping they work.

This guide is designed to change that. It is a complete, offline-ready reference that walks you from Git's underlying mental model all the way to the advanced workflows used on real production teams. It is intentionally practical: every chapter is built around commands you run, output you should see, and mistakes you can recover from — not abstract theory.

You do not need the internet to use this guide. You do not need to memorize anything. You need only a terminal and the willingness to experiment in a throwaway repository. That is how you learn Git safely: break things on purpose, then use this book to fix them.

What makes this different from reading free docs

The free Git Push Studio site gives you concise command references — 28 commands with examples. That is a cheat sheet. This guide is the full course:

- **A coherent mental model.** Instead of forty disconnected pages, you get one

consistent story about what Git actually is (a content-addressable object database), which makes every command behave in a predictable way.

- **Real terminal transcripts.** You see the commands, the output, and what changed —

exactly as they would appear on your machine.

- **Recovery before memorization.** A third of this book is about undoing mistakes,

the skill that separates anxious Git users from confident ones.

- **Workflow playbooks.** Step-by-step recipes for the situations you will actually

face: shipping a feature, surviving a bad merge, fixing yesterday's terrible commit, and untangling a messy pull request.

- **Exercises and checkpoints.** Each chapter closes with a short exercise you can

complete in five minutes in a scratch repository.

What you need

- Git installed (any recent version — if you can run `git --version`, you are set).
- A terminal. Linux, macOS, and Windows (PowerShell or Git Bash) all work; this guide

uses the `$` prompt as a cross-platform convention. Windows users: Git Bash or WSL is recommended, and `weird` path quoting notes are called out where they matter.

- A scratch directory. Make one now: `mkdir ~/git-lab && cd ~/git-lab`. You will live

here during the exercises.

How to use this book

- **Sequential readers:** chapters build on each other. Read in order.
- **Busy readers:** chapters 3, 5, and 7 cover the 90% case. Read those first, then

circle back.

- **Ad-hoc readers:** the final chapter is a full command cheat sheet and glossary.

Skim it anytime.

Every code block is copy-paste safe. Terminal output that represents something *other than* what you typed is shown with a `#` prompt or in a "…" elision, and in this book it is always printed in monospace so you can tell input from output.

Let's begin.

How to Read the Terminal Transcripts

Throughout this guide you will see blocks like this:

```
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

The line starting with `$` is a command you type. Lines that follow are the output. When output has been trimmed to keep the example short, you will see `…` on its own line. Anything in `[brackets]` inside output, such as `[main 3f9a11c]`, is a value Git generates on your machine and will differ from the book — that is expected.

A note on safety. Everything in chapters 1–9 can be run safely in `~/git-lab`. Where a command is destructive (`git reset --hard`, `git rebase`, `filter-branch`), the chapter that covers it says so out loud before you run it, and gives you the escape hatch first.

1. Git's Mental Model

What Git actually is, and why knowing it makes every command obvious

Before you learn any more syntax, build the picture once. Almost every confusing Git error message is only confusing because the model in your head is wrong. Fix the model and the errors become self-explanatory.

Git is a content-addressable object store

At its core, Git is a database of *snapshots*, keyed by the SHA-1 hash of their contents. When you commit, Git stores:

- **blobs** — file contents, nothing else. Same content always produces the same blob

hash, regardless of filename.

- **trees** — directory listings: filename → blob/tree + permissions. A tree is the

state of the folder at one moment.

- **commits** — a snapshot object that records: one tree, zero or more parent commits,

an author, a committer, a message, and a timestamp.

That is the entire world. `git commit`, `git push`, `git merge` — they are all just machinery for creating, storing, and relocating these three

kinds of objects.

The insight that unlocks everything: Git does not store *diffs*. It stores complete snapshots. A diff is something Git *computes on demand* when you ask it to compare two snapshots. This is why Git can branch, merge, and rewind so cheaply — nothing is ever "partially updated".

The commit graph is a DAG

Commits form a **directed acyclic graph** (DAG): each commit points back at its parent(s). A normal commit has one parent; a merge commit has two (or more).

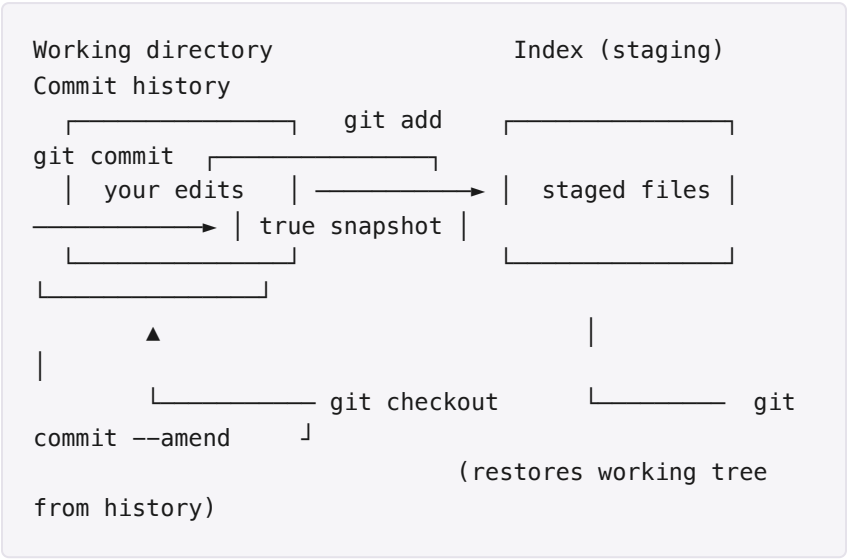
```
o---o---o---o---o        main
         \
          o---o          feature
```

This orientation — *backwards time* — is the single most useful fact in this book: **commit messages describe changes, but commits point to what came before.**

Because every commit records its parent hashes, any two histories have a unique **merge base** (the nearest common ancestor). Merges, rebases, and "is this branch behind?" are all implemented by walking this graph. When you graduate from typing commands to *reading* history, you are always reading a DAG.

The three places your work lives

You are always moving content between three areas. Its a classic diagram, and chapter after chapter will refer back to it:



- **Working directory** — the files you edit. Git knows what's here only by

comparison: "is this file different from the index?"

- **Index (staging area)** — a proposed next snapshot. `git add` places file contents

here. `git status` shows you the difference between working dir and index.

- **History** — the committed DAG. `git commit` freezes the index into a permanent,

immutable snapshot.

Notice what is *not* in that diagram: you. That's by design. Git is not a server that coordinates people; it is a tool you run. Collaboration happens through **remotes**, which are just copies of a repository that you push to and fetch from (chapter 5).

Refs: names for commits

A 40-character hash is a great address and a terrible name. **Refs** (references) are human-readable labels stored under `.git/refs/`:

- `refs/heads/main` — a **branch**: a pointer that moves forward as you commit.
- `refs/remotes/origin/main` — a **remote-tracking branch**: your local memory of what

the remote looked like last time you fetched.

- `refs/tags/v1.0` — a **tag**: a pointer that *never* moves (usually).

Branches are cheap. A branch is not a collection of commits; it is just a moving label naming the latest commit, and "everything reachable from it" is the branch. Creating a branch is a one-line write to a file. That mental flip — *branch = movable label, not a folder of work* — removes most merge anxiety you will ever feel.

Two very useful facts about refs:

1. `HEAD` is a special ref meaning "where I am now". Usually `HEAD` points at a branch,

which points at a commit. In a *detached HEAD* state it points directly at a commit.

1. Refs are what you *normally* manipulate, but nothing stops you from using a raw hash:

`git log 3f9a11c`, `git show main~2` — refs and hashes are interchangeable arguments in almost every command. That flexibility is why `~` and `^` notation matters (below).

Relative revision notation

You will constantly want "the commit before this one". Git gives you two operators:

- `HEAD~1` — the first-parent ancestor; `HEAD~n` climbs n commits up the main line.
- `HEAD^1` — the first parent; `HEAD^2` — the second parent (only meaningful for merge

commits).

For a linear history they are the same: `HEAD~2` == `HEAD^^`. For a merge `M` with parents `A` and `B`: `M^1` is `A`, `M^2` is `B`, and `M~1` is `A`.

Bonus precision: `git show HEAD~1:path/to/file.txt` retrieves a file's content as it was one commit ago — no checkout required. You will use this pattern constantly.

What the hash really is

Every object's hash is, by construction, a test of whether it is *identical* to another object. Two files with the same content and metadata produce the same blob hash, so Git can store the same

content once and share it across repositories. This is also why Git verifies downloads: if a transfer corrupts a byte, the hash won't match and Git refuses the object.

Checkpoint exercise

In `~/git-lab`, initialize a repo and create your first real objects:

```
$ cd ~/git-lab
$ git init git-lab-1 && cd git-lab-1
$ echo "hello git" > hello.txt
$ git add hello.txt
$ git commit -m "first commit"
```

Now look behind the curtain:

```
$ git cat-file -t HEAD            # what kind of object
is HEAD?
commit
$ git cat-file -p HEAD            # show the commit's
internals
tree 4b825dc642cb6eb9a060e54bf8d69288fbee4904
author Ajit <ajit@gitpushstudio.in> 1730000000 +0000
committer Ajit <ajit@gitpushstudio.in> 1730000000 +0000

first commit
```

`git cat-file -t` (type) and `-p` (pretty print) are your X-ray goggles. Use them a few times tonight and the object model will stop being abstract. You have just touched blobs, trees, and commits — the entire universe of Git.