Git In

Engineering a Version Control System

"From ancient counting stones to quantum algorithms—
every data structure tells the story of human ingenuity."

Living First Edition

Updated December 31, 2025

LICENSE & DISTRIBUTION

Git Internals: Software Engineering, Architecture, Design Patterns

A Living Architecture of Computing

Git Internals is released under the **Creative Commons Attribution-ShareAlike 4.0 International License** (CC BY-SA 4.0).

FORMAL LICENSE TERMS

You are free to:

- **Share** — copy and redistribute the material in any medium or format for any purpose, even commercially.
- **Adapt** — remix, transform, and build upon the material for any purpose, even commercially.

Under the following terms:

- **Attribution** — You must give appropriate credit to Mahdi, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
- **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

DISTRIBUTION & SOURCE ACCESS

Repository: The complete source code (LaTeX, diagrams, examples) is available at:
https://github.com/m-mdy-m/algorithms-data-structures/tree/main/books/books

Preferred Citation Format:
Mahdi. (2025). *Git Internals*. Retrieved from
https://github.com/m-mdy-m/algorithms-data-structures/tree/main/books/books

Version Control: This is a living document. Check the repository for the most current version and revision history.

WARRANTIES & DISCLAIMERS

No Warranty: This work is provided "AS IS" without warranty of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose.

Limitation of Liability: In no event shall Mahdi be liable for any direct, indirect, incidental, special, exemplary, or consequential damages arising from the use of this work.

Educational Purpose: This work is intended for educational and research purposes. Practical implementation of algorithms and techniques should be thoroughly tested and validated for production use.

TECHNICAL SPECIFICATIONS

Typeset with: LaTeX using Charter and Palatino font families
Graphics: TikZ and custom illustrations
Standards: Follows academic publishing conventions
Encoding: UTF-8 with full Unicode support
Format: Available in PDF, and LaTeX source formats

License last updated: December 31, 2025

For questions about licensing, contact: bitsgenix@gmail.com

Contents

Title Page i

Contents iii

Preface vi

Introduction ix

I Foundations 1

1 Version Control: The Problem 3

2 Evolution of Version Control 4

3 Centralized vs Distributed 5

4 Git's Design Philosophy 6

II The Codebase: Structure and Organization 7

5 Source Tree Layout 9

6 The Entry Point 10

7 Core vs Porcelain 11

8 Module Dependencies 12

9 Platform Abstraction 13

III The Data Model: Git's Core Insight 14

10 Content-Addressable Storage 16

11 The Four Object Types 17

12 The DAG: Directed Acyclic Graph 18

13 References: Branches and Tags 19

14 The Index (Staging Area) 20

15 The Working Directory 21

IV Storage and Performance 22

16 Object Storage: Loose and Packed 24

17 Pack Files 25

18 Delta Algorithms . . . 26
19 Transfer Protocols . . . 27
20 Performance Characteristics . . . 28

V Design Patterns in Git's Implementation . . . 29

21 Immutable Data Structures . . . 31
22 Content-Addressable Store Pattern . . . 32
23 Command Pattern . . . 33
24 Strategy Pattern . . . 34
25 Builder Pattern . . . 35
26 Object Pool and Caching . . . 36
27 Plugin Architecture . . . 37

VI Core Algorithms . . . 38

28 Three-Way Merge . . . 40
29 Merge Strategies . . . 41
30 Diff Algorithms . . . 42
31 Tree Comparison . . . 43
32 Object Lookup . . . 44
33 Reachability and Graph Traversal . . . 45
34 Blame and History Mining . . . 46

VII Advanced Topics . . . 47

35 Rebasing . . . 49
36 Cherry-pick and Revert . . . 50
37 Hooks . . . 51
38 Submodules and Subtrees . . . 52
39 Worktrees . . . 53
40 Garbage Collection . . . 54

VIII Git vs The World . . . 55

41 Git vs Mercurial . . . 57

42 Git vs Subversion 58
43 Git vs Perforce 59
44 Git's Influence 60
Acknowledgments 61

Preface

I STARTED READING git's source code because I was bored.

Not in a bad way. I'd just finished a project and didn't want to start another one yet. So I cloned git, opened `common-main.c`, and started reading.

That was six months ago.

What started as "let me see how this works" turned into something else. I kept finding things that made me stop and think. Not just "oh that's clever" but "wait, why did they do it THIS way and not that way?"

Like why is everything a hash? Why DAG instead of a tree? Why are branches basically just text files with a hash in them? Why does merge work the way it does?

These aren't git usage questions. These are software engineering questions.

What This Actually Is

This isn't a git tutorial. There are plenty of those already. Pro Git is excellent if you want to learn how to use git.

This is about how git is built. The code. The data structures. The algorithms. The design decisions. The patterns. The trade-offs.

I'm reading the source code and explaining what I find. Not line by line of everything—that would be insane and boring. But the interesting parts. The parts that teach you something about building software.

Why does git's source look like this? Why these data structures? What problems were they solving? What did they try that didn't work? Where are the clever bits?

Who This Is For

You if you've ever wondered "how does git actually work under the hood?"

You if you want to see what a mature, complex C codebase looks like.

You if you're interested in software architecture, not just using tools.

You if you want to understand the engineering, not just memorize commands.

You don't need to be a C expert. You don't need to be a git expert. You just need to be curious about how things work.

What I'm Not Doing

I'm not teaching you git commands. I assume you already know basic git or can learn it elsewhere.

I'm not proving theorems. This isn't a CS paper. When I talk about complexity or algorithms, I keep it practical.

I'm not being comprehensive. Git has thousands of files. I'm focusing on the core stuff that matters.

I'm not being perfect. I'll probably get some things wrong. That's fine. The goal is understanding, not perfection.

How I'm Writing This

I'm trying to write the way I actually think about code. Informal. Direct. Sometimes messy.

When I say "this is clever" I mean it. When I say "I don't get why they did this" I mean that too.

I'll show you code. Real code from git. Not cleaned up pseudocode. The actual thing.

I'll explain design patterns when I see them. Not because I want to sound academic, but because patterns matter. They're how you make sense of large codebases.

The Structure

Part I: What git is, why it exists, basic concepts. Quick. Just enough context.

Part II: The architecture. How the codebase is organized. Where things are. How to navigate it.

Part III: The data model. This is the core insight that makes git work. Objects, hashes, DAG.

Part IV: Storage and performance. Pack files, delta compression, why git is fast.

Part V: Design patterns. The engineering patterns I see throughout the code.

Part VI: Core algorithms. Merge, diff, tree traversal. The complicated stuff.

Part VII: Extension points. Hooks, custom commands, how git lets you extend it.

Part VIII: Comparisons. Git vs Mercurial vs SVN. What's different and why.

A Note on the Code

Git is written in C. Old school C. Pointers everywhere. Manual memory management. Platform-specific macros.

If that scares you, don't worry. I'll explain the tricky bits. And honestly, reading real C code is a useful skill even if you never write C.

The patterns and ideas in git's code apply to any language. Content-addressable storage works in Python. DAG algorithms work in Go. Design patterns are language-agnostic.

This Is Incomplete

I'm still learning this stuff. Still reading the code. Still finding new things.

So this book is a snapshot. What I understand right now. In six months I'll probably understand more and want to rewrite parts of it.

That's okay. That's how learning works.

Why Bother?

Because git is everywhere. Every developer uses it. Most don't understand it.

Because understanding tools makes you better with them.

Because git's codebase has lessons about software engineering that apply way beyond version control.

Because it's interesting.

Let's start.

Mahdi
2025

Introduction

OPEN UP *git.c* in the git source code. First thing you see:

```
int cmd_main(int argc, const char **argv)
{
    struct strvec args = STRVEC_INIT;
    const char *cmd;
    int done_help = 0;
```

Wait. `cmd_main`? Not `main`?

That's weird. Every C program has `main`. Why is git different?

And `struct strvec args = STRVEC_INIT`—what's that macro doing? Why not just allocate normally?

And `done_help`—that's a strange name for a variable. What does it track?

These aren't just random choices. Each one solves a problem. Each one has a reason.

This book is about those reasons.

What You're Going To Read

This is a walkthrough of git's internals from a software engineering perspective.

Not how to use git. How git is built.

We're going to look at:

- The data model and why it's brilliant
- The codebase structure and how it's organized
- The algorithms and their complexity
- The design patterns used throughout
- The performance tricks and optimizations
- The extension points and plugin system
- How it compares to other version control systems

What This Book Is Not

This is not a git tutorial. I'm not teaching you commands.

This is not a complete reference. Git has over 2000 files. I'm covering the important parts.

This is not academic. I'll talk about algorithms and complexity, but I'm not writing proofs.

This is not perfect. I'm figuring this stuff out as I go. I'll make mistakes. That's fine.

Why Git?

Because it's everywhere. You probably use it every day.

Because it's well-designed. The core model is elegant even though the command interface is messy.

Because it's mature. Git has been around since 2005. The codebase has evolved, been refactored, been optimized. You can learn from that evolution.

Because it's interesting. Git does things differently than other version control systems. Understanding why teaches you about software design.

Prerequisites

You should know:

- Basic programming (any language is fine)
- Basic git usage (add, commit, push, merge)
- Basic data structures (trees, graphs, hash tables)

You don't need to know:

- C (I'll explain the tricky parts)
- Advanced git (internals, plumbing commands)
- Algorithm analysis (I'll teach what you need)

How To Read This

Start with Part I if you want context. Skip it if you already know what version control is and why git was created.

Part II (architecture) and Part III (data model) are essential. Read those.

After that, jump around. Want to understand merge algorithms? Go to Part VI. Want to see design patterns? Part V. Interested in how pack files work? Part IV.

Code examples are real. They're from git's actual source code. Sometimes I'll simplify for clarity, but I'll tell you when I do.

What Makes Git Special?

Most version control systems think about changes. "User X changed line 47 from A to B."

Git thinks about snapshots. "Here's what the entire project looked like at this moment."

That one decision—snapshots not deltas—changes everything. It makes branching cheap. It makes merging easier. It makes git fast.

But it creates problems too. How do you store thousands of snapshots without using infinite disk space? How do you transfer data efficiently? How do you handle merge conflicts?

Git solves these problems in interesting ways. We're going to look at how.

The Core Insight

Everything in git is content-addressable.

Files aren't named "README.md". They're named by their SHA-1 hash. If two files have the same content, they have the same hash, so git only stores them once.

This sounds weird but it's powerful. It means git can tell if two things are the same by comparing hashes. It means deduplication is automatic. It means you can reference any object by its hash and git can verify it hasn't been corrupted.

The entire git data model follows from this insight.

A Note on C

Git is written in C. Not modern C++ or Rust. Old school pointer-heavy manual-memory-management C.

Why? Because git was written in 2005 by kernel developers. C was what they knew. C is portable. C is fast. C gives you control.

Reading C code is useful even if you never write C. It teaches you what's really happening. No abstractions hiding the details. Just pointers and memory and syscalls.

I'll explain the confusing parts. You'll be fine.

Starting Point

We're going to start with the basics. What is git? Why does it exist? What problems does it solve?

Then we'll look at the architecture. How is the codebase organized? Where are things?

Then the data model. This is the heart of git. Objects, hashes, the DAG.

Then we go deeper. Algorithms, patterns, performance, extensions.

By the end, you'll understand how git works. Not just "here's what this command does" but "here's how this command is implemented and why."

Let's start.

Part I

Foundations

Before diving into git's code, we need context. Why does version control exist? What problems does it solve? How did we get to git?

This part covers the basics quickly. If you already know this stuff, skim it or skip to Part II.

Chapter 1

Version Control: The Problem

Chapter 2

Evolution of Version Control

Chapter 3

Centralized vs Distributed

Chapter 4

Git's Design Philosophy

Part II

The Codebase: Structure and Organization

Before reading code, you need a map. Where are things? How is functionality divided? What are the major subsystems?

This part gives you that map.

Chapter 5

Source Tree Layout

Chapter 6

The Entry Point

Chapter 7

Core vs Porcelain

Chapter 8

Module Dependencies

Chapter 9

Platform Abstraction

Part III

The Data Model: Git's Core Insight

This is what makes git work. The data model is elegant, simple, and powerful.

Everything else in git follows from this model.

Chapter 10

Content-Addressable Storage

Chapter 11

The Four Object Types

Chapter 12

The DAG: Directed Acyclic Graph

Chapter 13

References: Branches and Tags

Chapter 14

The Index (Staging Area)

Chapter 15

The Working Directory

Part IV

Storage and Performance

Storing every file in every commit should use infinite space. It doesn't. How?

This part covers git's storage optimizations and performance characteristics.

Chapter 16

Object Storage: Loose and Packed

Chapter 17

Pack Files

Chapter 18

Delta Algorithms

Chapter 19

Transfer Protocols

Chapter 20

Performance Characteristics

Part V

Design Patterns in Git's Implementation

Git's codebase uses recognizable design patterns. Not always explicitly, but they're there.

Identifying these patterns helps you understand the code and teaches you about software architecture.

Chapter 21

Immutable Data Structures

Chapter 22

Content-Addressable Store Pattern

Chapter 23

Command Pattern

Chapter 24

Strategy Pattern

Chapter 25

Builder Pattern

Chapter 26

Object Pool and Caching

Chapter 27

Plugin Architecture

Part VI

Core Algorithms

The interesting algorithms. Merge, diff, tree traversal, conflict resolution.

This is where complexity analysis matters.

Chapter 28

Three-Way Merge

Chapter 29

Merge Strategies

Chapter 30

Diff Algorithms

Chapter 31

Tree Comparison

Chapter 32

Object Lookup

Chapter 33

Reachability and Graph Traversal

Chapter 34

Blame and History Mining

Part VII

Advanced Topics

Features that aren't core but matter: rebasing, hooks, submodules, worktrees.

How they work and why they're complicated.

Chapter 35

Rebasing

Chapter 36

Cherry-pick and Revert

Chapter 37

Hooks

Chapter 38

Submodules and Subtrees

Chapter 39

Worktrees

Chapter 40

Garbage Collection

Part VIII

Git vs The World

Git isn't the only version control system. How does it compare?

What did git do differently? What are the trade-offs?

Chapter 41

Git vs Mercurial

Chapter 42

Git vs Subversion

Chapter 43

Git vs Perforce

Chapter 44

Git's Influence

Acknowledgments

I would like to express my gratitude to everyone who supported me during the creation of this book. Special thanks to the open-source community for their invaluable resources and to all those who reviewed early drafts and provided feedback.