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

Preface

I HAD JUST FINISHED a project and wanted to start something new. I was working on gix—a Git wrapper I'd been thinking about—and realized I didn't actually understand Git well enough.

Sure, I could use it. I knew the commands. But how does it work? Why is it designed this way? What makes it different from other version control systems?

I read some articles. Looked at Pro Git's internals chapter. Found a small booklet on Git internals. But I wanted more. I wanted to see the actual code. The decisions. The trade-offs.

So I cloned the Git repository and started reading.

Why I'm Writing This

At first, I was just making notes for myself. Trying to understand the codebase. Mapping out how things connect.

Then I thought maybe I could write an article about it. Share what I learned.

But the more I read, the more I found. This wasn't article material. There was too much to say.

So here we are. A book about Git's internals from an engineering perspective.

What This Book Actually Is

This is me reading Git's source code and explaining what I find.

Not line by line—that would be boring and pointless. But the important parts. The clever solutions. The patterns that keep showing up. The reasons things are built the way they are.

I care about software engineering. How systems are designed. Why certain approaches work. What problems they solve. Git is interesting because it does things differently and those differences matter.

Who This Is For

You if you've used Git but want to understand how it works underneath.
You if you're interested in system design and want to see a real, mature codebase.
You if you like reading code and learning from it.
You don't need to be a C expert. I'll explain the tricky bits. You don't need to know Git internals already. That's what this book is about.

What This Isn't

This isn't a Git tutorial. I'm not teaching you how to use Git. There are better resources for that.
This isn't comprehensive. Git's codebase is massive. I'm focusing on the core ideas and the interesting engineering choices.
This isn't perfect. I'm still learning this stuff. I'll probably get some things wrong. That's okay.

How I'm Approaching This

I started with Git's first commit. The original implementation by Linus Torvalds. It's small and shows the core model clearly.
Then I worked forward, looking at how the codebase evolved. What got added. What got changed. Why.
I'm trying to explain not just what the code does but why it's designed that way. What problem was being solved? What alternatives existed? What trade-offs were made?

This Is Ongoing

I'm still reading the code. Still learning. This book reflects what I understand right now.
Maybe in six months I'll understand more and want to revise some sections. That's how learning works.
Consider this a snapshot of the journey, not the final destination.

Why Bother?

Because Git is everywhere and most people don't understand it.
Because understanding your tools makes you better at using them.
Because Git's design has lessons that apply beyond version control.

Because it's interesting.
Let's start.

Mahdi
2025

Introduction

EVERY DESIGN decision has a reason. Every data structure solves a problem. Every algorithm makes a trade-off.
When you look at mature software, you're seeing years of evolution. Hundreds of developers. Thousands of decisions. Some good, some less good, all made for specific reasons at specific times.
Git is like that. It's been around since 2005. It's used by millions. It's been optimized, refactored, extended. And understanding it teaches you about software engineering.

What You're Going To Learn

This book walks through Git's internals from an engineering perspective.
Not how to use Git. How Git is built.
We'll look at the data model and why it's clever. The codebase structure and how it evolved. The algorithms and their complexity. The design patterns that keep appearing. The performance optimizations. How it all fits together.
And more importantly—why. Why these choices? What problems do they solve? What are the trade-offs?

What This Book Isn't

This is not a Git tutorial. If you need to learn Git commands, read Pro Git or the official documentation.
This is not complete reference. Git has over 2000 files and hundreds of commands. I'm covering the important parts, the core insights, the interesting engineering.
This is not academic. I'll talk about algorithms and complexity, but I'm not writing proofs or formal specifications. This is practical engineering.
This is not flawless. I'll make mistakes. I'll miss things. That's fine. The goal is understanding, not perfection.

Why Git?

Because you probably use it every day. Most developers do.
Because it's well-designed at its core. The data model is elegant even if the command interface is messy.
Because it's mature. Twenty years of evolution. You can learn from that evolution—what worked, what didn't, what got changed and why.
Because it does things differently. Most version control systems work one way. Git works another way. Understanding why teaches you about design choices.

Prerequisites

You should know basic programming. Any language is fine. Concepts like variables, functions, data structures.
You should know basic Git usage. Add, commit, push, merge. Not expert level, just the basics.
You should know basic data structures. Trees, graphs, hash tables. Nothing fancy.
You don't need to know C deeply. I'll explain the important parts.
You don't need to know advanced Git internals. That's what this book teaches.
You don't need algorithm analysis background. I'll teach what matters.

How To Read This

Start with Part I if you want historical context. Skip it if you already know about version control evolution and why Git was created.
Parts II and III are essential. They cover architecture and the data model. Read those.
After that, jump around based on your interests. Want to understand merging? Go to Part VI. Curious about performance? Part IV. Interested in patterns? Part V.
Code examples are simplified for clarity but accurate in concept. When I show code, I'll tell you if I've simplified it.

The Core Ideas

Most version control systems think in terms of changes. They track what changed between versions. Git doesn't.
Git thinks in terms of snapshots. It stores the complete state of your project at each commit. Not the difference—the whole thing.

That one decision changes everything. It makes branching cheap. It makes merging easier. It makes Git fast. But it also creates challenges. How do you store thousands of snapshots efficiently? How do you transfer them over networks? How do you handle conflicts?
Git solves these problems in specific ways. We're going to look at how and why.

Content-Addressable Storage

Everything in Git is identified by its content hash. Files aren't named by their path. They're named by their SHA-1 hash.
If two files have the same content, they have the same hash. Git only stores them once. Automatic deduplication.
This seems weird at first but it's powerful. Git can tell if two things are identical by comparing hashes. It can detect corruption by verifying hashes. It can reference any object by its hash.
The entire system builds on this idea.

Starting Point

We'll start with foundations. What is version control? Why does Git exist? What problems does it solve?
Then architecture. How is the codebase organized? Where are things located?
Then the data model. This is the heart of Git. Objects, hashes, the DAG structure.
Then we go deeper. Storage, algorithms, patterns, performance.
By the end, you'll understand how Git works. Not just what commands do, but how they're implemented and why.

A Word on Engineering

Good engineering isn't about being clever. It's about solving problems effectively. Sometimes the best solution is simple. Sometimes it's complex because the problem is complex. Sometimes it's ugly because reality is messy.
Git has all of these. Simple elegant parts. Complex necessary parts. Ugly historical parts.
Learning to distinguish between them—to see which complexity is essential and which is accidental—that's what studying real systems teaches you.
Let's start.

Part I

Context

Before diving into Git's engineering, you need to understand why it exists and what problems it solves.

This part is quick. Just enough background to make sense of the decisions that follow.

Chapter 1

Version Control Evolution

Chapter 2

Git's Origin Story

Chapter 3

Distributed vs Centralized

Chapter 4

Git's Philosophy

Part II

Architecture

How is a 20-year-old codebase organized? Where does functionality live? How do pieces connect? This part maps the terrain before we dive into details.

Chapter 5

Codebase Structure

Chapter 6

Command Dispatch System

Chapter 7

Platform Abstraction

Chapter 8

The Entry Point and Initialization

Part III

The Data Model

Git's core insight. Everything builds on this. Content-addressable storage. Four object types. The DAG.

Chapter 9

Content-Addressable Storage

Chapter 10

Object Types

Chapter 11

The DAG

Chapter 12

References

Chapter 13

The Index

Chapter 14

Repository Structure

Part IV

Algorithms

The complex parts. Graph algorithms. Diff. Merge. Tree comparison. Where algorithmic choices determine performance and correctness.

Chapter 15

Graph Traversal

Chapter 16

Merge Base Computation

Chapter 17

Three-Way Merge

Chapter 18

Merge Strategies

Chapter 19

Diff Algorithms

Chapter 20

Rename Detection

Chapter 21

Tree Comparison

Chapter 22

Object Lookup

Chapter 23

Reachability Algorithms

Chapter 24

History Simplification

Chapter 25

Blame Algorithm

Chapter 26

Packfile Generation

Part V

The Object Model

Git's core insight. Content-addressable storage. Four object types. How they relate. Why this model is powerful. This is the heart of everything.

Chapter 27

Content-Addressable Storage

Chapter 28

Object Types and Structure

Chapter 29

Object Serialization

Chapter 30

Loose Objects

Chapter 31

References System

Chapter 32

The Index (Staging Area)

Chapter 33

RefLog

Part VI

Pack Files and Compression

How Git stores thousands of objects efficiently. Delta compression. Pack formats. Transfer protocols. The engineering that makes Git fast.

Chapter 34

Pack File Format

Chapter 35

Pack Index Format

Chapter 36

Delta Compression

Chapter 37

Deltification Process

Chapter 38

Multi-Pack Index (MIDX)

Chapter 39

Cruft Packs

Chapter 40

Pack Protocol

Chapter 41

Pack Window Management

Part VII

Engineering Concepts

Patterns, tricks, and techniques that appear throughout the codebase. How Git handles complexity, maintains performance, and stays extensible.

Chapter 42

Immutability and Its Consequences

Chapter 43

Copy-on-Write Patterns

Chapter 44

Lazy Loading and Evaluation

Chapter 45

State Machines

Chapter 46

Callback and Iterator Patterns

Chapter 47

Bitmap Indices

Chapter 48

Prefix Compression

Chapter 49

Sparse Data Structures

Chapter 50

Atomic Operations

Chapter 51

Error Handling Strategies

Chapter 52

Extension and Plugin Architecture

Part VIII

Performance and Optimization

Git is fast. Not by accident. Specific optimizations, careful profiling, constant attention to performance. How it stays fast as repositories grow.

Chapter 53

Operation Complexity

Chapter 54

Caching Strategies

Chapter 55

Parallelization

Chapter 56

I/O Optimization

Chapter 57

Network Optimization

Chapter 58

Geometric Repacking

Chapter 59

Commit Graph Optimization

Chapter 60

Profiling and Benchmarking

Part IX

Security and Integrity

Git's "trust but verify" model. Cryptographic hashing. Attack surfaces. How Git protects data integrity and handles malicious input.

Chapter 61

Cryptographic Hashing

Chapter 62

Object Integrity

Chapter 63

Input Validation

Chapter 64

GPG Integration

Chapter 65

Attack Surfaces

Chapter 66

Fuzzing and Testing

Part X

Advanced Features

Features built on the core. More complex because they combine multiple primitives. Understanding implementation reveals trade-offs.

Chapter 67

Rebasing Internals

Chapter 68

Submodules

Chapter 69

Worktrees

Chapter 70

Partial Clone

Chapter 71

Sparse Checkout

Chapter 72

LFS Integration

Chapter 73

Garbage Collection

Part XI

Comparisons and Evolution

Git vs other systems. What trade-offs were made. What we can learn from alternatives.

Chapter 74

Git vs Mercurial

Chapter 75

Git vs Subversion

Chapter 76

Git vs Perforce

Chapter 77

Modern VCS

Chapter 78

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.