



Git Internals





Git Internals



Mahdi

bitsgenix@gmail.com



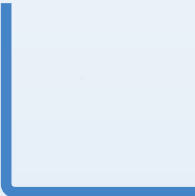
FIRST EDITION



Version fb4e084-dirty

© 2026 Mahdi

<https://github.com/papyrxis/git-internal.git>



Git Internals

Written by Mahdi

First Edition

Published: 2026

Copyright Notice

Copyright © 2026 Author Name

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit:

<http://creativecommons.org/licenses/by-sa/4.0/>

Version Information

Version: fb4e084-dirty

Built: 2026-01-02 23:35:31 UTC

For the latest version, visit:

<https://github.com/papyrxis/git-internal.git>

Typeset with \LaTeX using Papyrxis workspace.

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.

Contents

Preface	ii
Introduction	v
Contents	viii
I Context	1
1 Why Git Was Built	3
2 Core Philosophy	4
II The Data Model	5
3 Content-Addressable Storage	7
4 The Four Object Types	8
5 The Commit Graph (DAG)	9
6 References and HEAD	10
7 The Index (Staging Area)	11
8 Repository Layout	12
III Codebase Architecture	13
9 Source Tree Organization	15
10 Command Dispatch	16
11 Platform Abstraction Layer	17
12 Configuration System	18
13 Testing Infrastructure	19
IV Storage and Compression	20
14 Loose Objects	22
15 Pack Files	23
16 Delta Compression	24
17 Multi-Pack Index (MIDX)	25
18 Commit Graph File	26

V	Core Algorithms	27
19	Diff Algorithms	29
20	Three-Way Merge	30
21	Merge Strategies	31
22	Merge Base (LCA)	32
23	Graph Traversal	33
24	Rename Detection	34
25	Tree Comparison	35
26	Blame Algorithm	36
VI	Performance Engineering	37
27	Operation Complexity	39
28	Caching Strategies	40
29	Parallelization	41
30	I/O Optimization	42
31	Network Protocol	43
32	Geometric Repacking	44
VII	Engineering Patterns	45
33	Immutability	47
34	Callback and Iterator Patterns	48
35	Error Handling	49
36	Memory Management	50
37	State Machines	51
38	Extensibility	52
VIII	Security and Integrity	53
39	Cryptographic Foundation	55
40	Object Integrity	56
41	Input Validation	57
42	Attack Surfaces	58

43	GPG Integration	59
IX	Evolution and Maintenance	60
44	Historical Evolution	62
45	Backward Compatibility	63
46	Code Quality Practices	64
47	Scalability Improvements	65
48	Modern Alternatives	66
	Additional Resources	67
.1	Mathematical Background	67
.2	Code Examples	67
.3	Solutions to Exercises	67
.4	Further Reading	67
	Bibliography	67

Part I

Context

Why Git exists. What problems it solves. Just enough context to understand the engineering decisions that follow.

Chapter 1

Why Git Was Built

Chapter 2

Core Philosophy

Part II

The Data Model

Git's core insight. Everything builds on this. Four object types, content-addressable storage, the DAG. This is the foundation.

Chapter 3

Content-Addressable Storage

Chapter 4

The Four Object Types

Chapter 5

The Commit Graph (DAG)

Chapter 6

References and HEAD

Chapter 7

The Index (Staging Area)

Chapter 8

Repository Layout

Part III

Codebase Architecture

How is a 20-year-old, million-line codebase organized? Where does functionality live? How do components connect?

Chapter 9

Source Tree Organization

Chapter 10

Command Dispatch

Chapter 11

Platform Abstraction Layer

Chapter 12

Configuration System

Chapter 13

Testing Infrastructure

Part IV

Storage and Compression

How Git stores thousands of objects efficiently. The evolution from loose objects to sophisticated pack files.

Chapter 14

Loose Objects

Chapter 15

Pack Files

Chapter 16

Delta Compression

Chapter 17

Multi-Pack Index (MIDX)

Chapter 18

Commit Graph File

Part V

Core Algorithms

The complex parts. Where algorithmic choices determine correctness and performance.

Chapter 19

Diff Algorithms

Chapter 20

Three-Way Merge

Chapter 21

Merge Strategies

Chapter 22

Merge Base (LCA)

Chapter 23

Graph Traversal

Chapter 24

Rename Detection

Chapter 25

Tree Comparison

Chapter 26

Blame Algorithm

Part VI

Performance Engineering

Git is fast by design. Specific optimizations, careful profiling, constant attention to performance.

Chapter 27

Operation Complexity

Chapter 28

Caching Strategies

Chapter 29

Parallelization

Chapter 30

I/O Optimization

Chapter 31

Network Protocol

Chapter 32

Geometric Repacking

Part VII

Engineering Patterns

Recurring patterns in Git's codebase. How complexity is managed. What makes the code maintainable.

Chapter 33

Immutability

Chapter 34

Callback and Iterator Patterns

Chapter 35

Error Handling

Chapter 36

Memory Management

Chapter 37

State Machines

Chapter 38

Extensibility

Part VIII

Security and Integrity

Git's "trust but verify" model. How data integrity is maintained. Attack surfaces and mitigations.

Chapter 39

Cryptographic Foundation

Chapter 40

Object Integrity

Chapter 41

Input Validation

Chapter 42

Attack Surfaces

Chapter 43

GPG Integration

Part IX

Evolution and Maintenance

*How Git evolved over 20 years. What changed, what didn't, and why.
Lessons from maintaining a large, critical codebase.*

Chapter 44

Historical Evolution

Chapter 45

Backward Compatibility

Chapter 46

Code Quality Practices

Chapter 47

Scalability Improvements

Chapter 48

Modern Alternatives

Additional Resources

.1 Mathematical Background

Additional mathematical concepts and formulas.

.2 Code Examples

Complete code listings.

.3 Solutions to Exercises

Selected exercise solutions.

.4 Further Reading

Recommended books and papers.