SELF-HOSTED DEVELOPMENT INFRASTRUCTURE

BUILDING YOUR OWN GIT HOSTING, CI/CD, AND SOFTWARE ENGINEERING PLATFORM FROM THE GROUND UP

GIT HOSTING
Own your code.
No compromises.

CI/CD
Automate builds.
Ship with confidence.

SECURE
Your data.
Your control.

SCALE
From one user
to thousands.

TREVOR NAGURO

Self-Hosted Development Infrastructure

Building Your Own Git Hosting, CI/CD, and Software Engineering Platform from the Ground Up

Steve Publications

This book is available at
https://leanpub.com/self-hosteddevelopmentinfrastructure

This version was published on 2026-08-18

Contents

Building Your Own Git Hosting, CI/CD, and Software Engineering Platform from the Ground Up

This book takes you from zero knowledge of self-hosting through to designing and operating a full-scale, organization-wide software development platform entirely under your own control. You will learn not only what to do but why each technology, configuration choice, and architectural decision exists – including when alternatives are better and what trade-offs each option introduces. Whether you are an individual developer seeking sovereignty over their code or a platform engineer building infrastructure for thousands of users, this book provides the technical depth and practical guidance needed to succeed.

Introduction

On August 17, 2026, GitHub went down. The outage lasted more than three hours. It broke code review tools, halted CI/CD pipelines, disabled Copilot, and left millions of developers worldwide unable to push, pull, or merge their work. This was not an isolated incident. Throughout 2025 and into 2026, GitHub experienced repeated reliability problems that pushed the platform below its own uptime standards, with outages and performance issues occurring nearly every week according to developer reports [1]. At the same time, pricing changes at major platforms — including GitHub introducing a per-minute fee for self-hosted runners and GitLab adjusting its tiered pricing — forced organizations to reconsider assumptions they had made about their development tooling costs.

These events are symptoms of a deeper structural reality: the modern software industry has concentrated an enormous amount of critical infrastructure into a handful of commercial platforms, and that concentration creates systemic risk. When your source code, CI/CD pipelines, issue trackers, package registries, and deployment automation all live on someone else's servers, you are dependent on their uptime, their pricing decisions, their security posture, and their legal jurisdiction. For many organizations this dependency is acceptable — even desirable — because it lets them focus on product development rather than infrastructure operations. But for others, the risks have become too great to ignore.

This book is about taking that critical infrastructure back under your own control. It is about building a self-hosted software development platform that gives you full visibility and authority over your data, your processes, and your dependencies — without sacrificing the capabilities you expect from modern development tools.

What This Book Covers

Self-hosting your development infrastructure is not simply installing GitLab on a server and calling it done. A production-grade self-hosted platform

touches many domains: version control hosting, authentication and authorization, CI/CD pipelines, artifact and container registries, backup and disaster recovery, monitoring and observability, network security, capacity planning, compliance requirements, and more. Each of these domains has its own complexities, trade-offs, and failure modes. This book walks through them systematically, building from foundations to advanced architecture.

The material is organized as a progressive learning path. Early chapters establish the conceptual and technical foundations: why organizations choose self-hosting, how Git actually works under the hood, and what infrastructure prerequisites you need before deploying anything. Middle chapters dive into specific technologies and operational practices: choosing and deploying a Git hosting platform, configuring security and identity management, building CI/CD pipelines, managing artifacts and containers, implementing backups and monitoring. Later chapters address advanced topics: scaling to thousands of users, migrating from hosted services, federated architectures, air-gapped deployments, and complete reference architectures that show how all the pieces fit together into a coherent whole.

Throughout the book you will find concrete commands, configuration files, scripts, and step-by-step procedures that you can adapt to your own environment. Every technical recommendation comes with an explanation of why it is recommended, what alternatives exist, and under what conditions those alternatives might be preferable. The goal is not to give you a set of recipes to follow blindly but to build your understanding so that you can make informed decisions for your specific situation.

Who This Book Is For

This book targets developers, DevOps engineers, platform engineers, systems administrators, SREs, and technical architects who want full control over their software development toolchain. It assumes basic familiarity with command-line usage, networking concepts, and general computing – you should know what a server is, how to run commands in a terminal, and what HTTP means. However, it does not assume prior experience with Git hosting infrastructure, Linux server administration, or DevOps tooling. If you have used Git as a developer but never operated a Git server, this book will take you from that starting point to production-ready expertise.

The technical depth is calibrated for experienced practitioners while remaining accessible to motivated beginners. When the book introduces advanced concepts — database tuning, Kubernetes orchestration, distributed systems patterns — it first establishes the foundational knowledge needed to understand them. Conversely, when covering fundamentals like Git protocols or SSH authentication, it does not oversimplify: understanding how these things actually work matters when you are responsible for operating them in production.

How to Use This Book

Read this book sequentially from start to finish. The chapters are deliberately ordered so that each one builds on concepts introduced earlier. Chapter 1 establishes the motivation and landscape. Chapter 2 covers Git fundamentals from an infrastructure operator's perspective — not a beginner's tutorial for using Git but an explanation of how repositories, protocols, and server-side operations actually work. Chapters 3 through 5 walk through preparing your infrastructure and deploying a Git hosting platform. Subsequent chapters expand outward into security, CI/CD, artifacts, backups, monitoring, scaling, migration, and advanced patterns. Chapter 14 brings everything together with complete reference architectures at different scales.

You do not need to follow every command or configuration in this book on your own hardware while reading. The book is designed to be studied as a coherent narrative first, then used as a reference during actual implementation. However, if you do want to experiment along the way, many of the procedures are written so that they can be executed in order on a fresh server. When commands or configurations differ between development demonstrations and production recommendations, the book explicitly calls out the distinction.

What This Book Does Not Cover

This book focuses on self-hosted infrastructure for software development — Git hosting, CI/CD, artifact management, and the supporting infrastructure that makes those services operational and secure. It does not cover general-purpose cloud computing, application architecture, or business-level IT strategy beyond what is directly relevant to running a development platform. It

does not provide exercises, quizzes, or homework assignments. Every page is instructional content meant to deepen your understanding of self-hosted development infrastructure.

The book also does not claim that self-hosting is universally the right choice. Chapter 1 examines when self-hosting makes sense and when it does not. For some organizations — particularly small teams with limited operational capacity — a hosted service remains the better option. This book will help you determine which category your situation falls into, and if self-hosting is appropriate, give you everything you need to do it well.

The journey ahead covers a lot of ground. By the end, you will understand not just how to deploy a self-hosted Git server but how to architect, operate, secure, scale, and maintain a complete software engineering platform that your organization can rely on. Let us begin.

Chapter 1: Why Self-Host? The Case for Owning Your Development Infrastructure

Before installing any software or provisioning any servers, you need to understand why self-hosting matters, what problems it solves, and whether it is actually the right choice for your situation. This chapter examines the forces driving organizations toward self-hosted development infrastructure, analyzes the economics and trade-offs involved, and helps you determine when self-hosting makes sense — and when it does not.

The Shift Away from Fully Managed Services

For more than a decade, the dominant trend in software development tooling has been consolidation around fully managed, hosted platforms. GitHub became the default home for source code. GitLab.com offered an integrated DevOps platform. Bitbucket Cloud served teams already invested in the Atlassian ecosystem. These services worked well enough that most organizations stopped thinking about where their development infrastructure lived — it was simply there, like electricity from a wall outlet.

This concentration created enormous efficiency gains. Teams no longer needed to maintain Git servers, configure web interfaces, manage databases, or patch software. They could focus on writing code while the platform providers handled operations at internet scale. The hosted model aligned perfectly with the broader shift toward cloud computing: pay for what you use, scale on demand, and let specialists handle the plumbing.

However, that convenience came with dependencies that many organizations did not fully appreciate until recent events forced them to confront it. Throughout 2025 and 2026, several factors converged to make those dependencies impossible to ignore:

Reliability problems at major platforms increased noticeably. GitHub experienced repeated outages in both years, including incidents lasting hours that affected core services like code hosting, pull requests, and CI/CD pipelines [1]. The average resolution time for GitHub incidents in 2024 was approximately 106 minutes, and the frequency of disruptions increased significantly into 2025 and 2026 [2]. When your entire development workflow depends on a single platform, even brief outages cascade into lost productivity across every team that relies on it.

Pricing changes altered economic calculations. GitHub introduced new fees for self-hosted runners in its Actions service, charging $0.002 per minute of usage on customer-managed infrastructure [3]. GitLab adjusted its tiered pricing structure multiple times, moving features previously available in free tiers into paid plans. Atlassian reduced storage limits for Bitbucket Cloud free users to a hard 1 GB ceiling in April 2025, triggering migrations from the platform [4]. These changes signaled that "free" or low-cost hosted tiers were becoming less sustainable for providers and less predictable for customers.

Data sovereignty requirements tightened globally. Cumulative GDPR fines reached €7.1 billion as of early 2026, with data transfer violations remaining a high-risk enforcement area [5]. The EU-US Data Privacy Framework provides partial mechanisms for transatlantic data flows, but Microsoft's own guidance acknowledges it cannot guarantee that EU data will never be requested by US authorities [5]. Organizations in regulated sectors — government, defense, healthcare, finance — increasingly face requirements that their development data remain within specific jurisdictions or under their direct control.

Supply chain security concerns intensified. High-profile incidents involving compromised packages and build systems made organizations more conscious of how much of their software supply chain runs on external infrastructure they do not control. When your CI/CD pipelines execute on shared runners owned by a third party, when your container images are pulled from public registries, and when your build dependencies come from external package repositories, the attack surface extends far beyond your own codebase.

These pressures have driven measurable changes in behavior. Searches for "self-hosted Git" increased by 100 to 140 percent as developers sought alternatives following GitHub's reliability issues and as AI-assisted coding tools made teams more aware of how deeply their workflows depend on single vendors [6]. Projects like Forgejo, the community-governed fork of Gitea, gained significant adoption — the Zig programming language moved its canonical repository

from GitHub to Codeberg (which runs Forgejo) in November 2024 as a visible statement of principle about software sovereignty [7].

The shift away from fully managed services is not a rejection of cloud computing or hosted infrastructure more broadly. It is a recognition that development tooling is critical infrastructure, and critical infrastructure deserves architectural decisions based on risk assessment rather than convenience alone.

What Self-Hosting Actually Means in 2026

"Self-hosted" means different things to different people. At one extreme, it means running everything on hardware you own in a data center you control – servers, networking equipment, power systems, physical security. At the other extreme, it means deploying open-source software on cloud virtual machines where you manage the applications but someone else manages the underlying infrastructure. Both are forms of self-hosting, and both fall short of what most people mean when they say "self-hosted" without qualification.

For the purposes of this book, self-hosting your development infrastructure means deploying and operating the software that runs your development platform on infrastructure where you control:

- **Data storage**: Your source code, CI/CD configurations, artifacts, logs, and metadata reside on storage systems under your administrative control. You decide where they are physically located, how they are backed up, who can access them, and what happens to them when an employee leaves or a project ends.
- **Software versions and updates**: You choose which versions of Git hosting software, CI/CD tools, databases, and supporting services to run. You control the timing of upgrades and can test changes before deploying them to production.
- **Security configuration**: You configure firewalls, authentication mechanisms, access controls, encryption settings, and monitoring according to your organization's security requirements rather than accepting a provider's defaults.
- **Operational procedures**: You define backup schedules, disaster recovery plans, incident response processes, and maintenance windows. When

something goes wrong, your team diagnoses and fixes it using tools and knowledge you control.

Self-hosting does not require owning physical hardware. Many organizations self-host their development infrastructure on cloud providers like AWS, GCP, or Azure — provisioning virtual machines, managed databases, and object storage while deploying open-source Git hosting platforms and CI/CD tools on top. This approach combines the sovereignty benefits of self-hosted software with the operational convenience of managed infrastructure. The key distinction is that you own the stack from the application layer upward, even if someone else owns the metal underneath.

Similarly, self-hosting does not require doing everything yourself. You can use Infrastructure as Code tools like Terraform to define your platform declaratively, configuration management tools like Ansible to automate server setup, and managed services for components where sovereignty is less critical (such as DNS or email delivery). The goal is control over the parts that matter most — your source code, your build pipelines, your artifacts — while pragmatically leveraging external services where appropriate.

Security, Compliance, and Data Sovereignty Drivers

Security and compliance are among the strongest motivators for self-hosting, particularly for organizations operating in regulated industries or handling sensitive intellectual property. The specific requirements vary by organization, but several recurring themes emerge.

Data residency and jurisdictional control: Many regulations require that certain types of data remain within specific geographic boundaries or legal jurisdictions. GDPR restricts transfers of personal data outside the European Economic Area without adequate protections [5]. China's Data Security Law and Personal Information Protection Law impose localization requirements for data collected within China [5]. India's Digital Personal Data Protection Act similarly requires controllers to store and process personal data in India under certain conditions [5]. When your development infrastructure includes personally identifiable information — employee records, user data embedded in code, access logs tied to individual developers — hosting that infrastructure on a US-based provider may create compliance obligations that self-hosting eliminates.

Beyond consumer privacy regulations, government and defense organizations often have explicit requirements that their development data never leave controlled environments. FedRAMP authorization is mandatory for cloud service providers working with US federal agencies [8], but even FedRAMP-authorized services may not satisfy the requirements of classified or sensitive-but-unclassified systems. For these organizations, self-hosting on government-controlled infrastructure is not a choice – it is a prerequisite.

Intellectual property protection: Companies developing proprietary technology often have legitimate concerns about where their source code lives and who has access to it. When you host your code on a commercial platform, the provider's employees may have administrative access to your repositories. The provider's security team may need to examine your data in response to abuse reports or legal requests. The provider's backup systems may replicate your data across multiple data centers in multiple jurisdictions. For most organizations using mainstream platforms, these risks are acceptable – the providers have strong security practices and legal protections. But for organizations developing highly sensitive technology, military systems, or competitive advantages that cannot tolerate even theoretical exposure, self-hosting provides assurance that their code never leaves a controlled boundary.

Auditability and transparency: Regulated industries often require detailed audit trails of who accessed what data, when changes were made to systems, and how access controls are enforced. Self-hosted platforms give you complete visibility into your own audit logs. You can configure logging at the level of detail your compliance requirements demand, retain logs for the periods regulators mandate, and demonstrate during audits that your controls are functioning as intended. With hosted services, you are limited to whatever audit capabilities the provider exposes through their API or web interface – which may not satisfy strict regulatory requirements.

Supply chain integrity: Self-hosting extends beyond your own code to the tools and dependencies used to build it. A self-hosted CI/CD platform means your build pipelines run on infrastructure you control, reducing the risk of contamination from shared runner environments. A self-hosted container registry means your images are stored where you can enforce scanning policies, access controls, and signing requirements. A self-hosted package repository proxy means your builds pull dependencies through a controlled gateway that can validate checksums, block compromised packages, and maintain an audit trail of what entered your build environment.

Cost Economics: Hosted Versus Self-Hosted Over Time

Cost is often the first consideration in hosting decisions, but it is also one of the most misunderstood. The economics of self-hosting versus hosted services depend heavily on your scale, your team's existing capabilities, and how you choose to account for different types of costs.

Direct infrastructure costs: At low usage levels, hosted services are almost always cheaper in terms of direct infrastructure spend. GitHub Actions includes generous free tiers (50,000 free monthly minutes for enterprise accounts) and charges approximately $0.022 per Linux minute beyond included allowances [9]. GitLab.com offers similar tiered pricing with free quotas for individual users and small teams. For a team doing 10,000 build minutes per month on hosted runners, the direct cost is trivial – often less than $100 per month.

Self-hosted infrastructure has fixed costs that do not scale down. A single virtual machine capable of running a Git hosting platform plus CI/CD runners might cost $40 to $100 per month depending on specifications and provider. Add database storage, artifact storage, backup storage, and monitoring tools, and the monthly infrastructure bill quickly exceeds what you would pay for equivalent usage on a hosted platform.

However, the economics flip at higher usage levels. At 50,000 monthly Linux build minutes, self-hosted infrastructure typically beats per-minute SaaS pricing [4]. At 100,000 minutes or more, the compute savings from self-hosting overwhelm the labor overhead of managing runner infrastructure [9]. A Hetzner VPS at €40 per month provides approximately 75 times cheaper compute per minute than GitHub Actions hosted runners [10], though this comparison ignores the operational burden of managing that infrastructure.

Labor and operational costs: This is where many organizations miscalculate. Self-hosting requires someone to manage the infrastructure – install software, apply updates, configure backups, monitor health, respond to incidents, plan capacity upgrades. Even with excellent automation, there is a baseline operational load that does not disappear. Estimates suggest self-hosted CI/CD platforms require 4 to 20 hours per month of administrative time depending on complexity and scale [4]. A Jenkins installation for a 20-person team typically costs $800 to $2,500 per month including EC2 instances, storage, and engineer maintenance time at $100 to $150 per hour [11].

If your organization already has a platform engineering or DevOps team whose job is to manage infrastructure, the marginal cost of adding self-hosted development tools may be low. Those engineers are already paid to operate systems; deploying another system on familiar technology stacks uses existing skills and processes. But if you do not have dedicated infrastructure staff and would need to pull developers away from product work to maintain servers, the opportunity cost can be enormous.

Hidden costs on both sides: Hosted services have their own hidden costs that organizations often overlook. Network egress fees add up when transferring artifacts between self-hosted runners and external services — GitHub Actions charges £0.055 per GiB per month beyond included storage, and CircleCI charges approximately £0.20 per GB for network and storage beyond plan limits [12]. Vendor lock-in costs emerge when migrating away from a platform becomes difficult due to proprietary features, custom integrations, or data export limitations. The cost of downtime during provider outages — measured in developer hours lost across the organization — is rarely factored into pricing comparisons but can be substantial for large teams.

Self-hosted infrastructure has hidden costs too. Security vulnerabilities in self-hosted software require prompt patching; if your team misses a critical update because someone forgot to check release notes, the cost of remediation far exceeds any savings from not paying for managed services. Backup failures that go undetected until you need to restore can be catastrophic — and they happen more often than people expect when backups are automated but never tested.

The break-even analysis: There is no universal answer to whether self-hosting is cheaper. The break-even point depends on your specific situation:

- Monthly build minutes: Below 25,000 minutes, hosted services almost always win on pure cost [9]. Above 100,000 minutes, self-hosted compute savings become compelling.
- Team size and existing infrastructure expertise: Organizations with dedicated platform teams have a significant advantage in managing self-hosted infrastructure efficiently.
- Security and compliance requirements: For organizations where hosted services cannot meet regulatory requirements, the question is not about cost but feasibility — self-hosting may be the only option regardless of economics.

- Long-term planning: Hosted service pricing can change at any time, while self-hosted infrastructure costs are more predictable (though subject to inflation and capacity needs).

Operational Burden and Hidden Costs of Self-Hosting

Self-hosting is not free labor. Every system you deploy creates ongoing obligations that persist as long as the system runs. Understanding these obligations before committing to self-hosting is essential for making an informed decision.

Patch management: Open-source software receives security updates regularly. Git hosting platforms, databases, reverse proxies, container runtimes, operating systems — all of them accumulate vulnerabilities over time and require timely patching. In a hosted environment, the provider applies these patches automatically. In a self-hosted environment, someone on your team must monitor release notes, test patches in a non-production environment, schedule maintenance windows, apply updates, and verify that everything works afterward. Missing critical security patches is one of the most common causes of breaches in self-hosted infrastructure.

Backup and disaster recovery: Hosted providers typically include backup as part of their service, often with point-in-time recovery capabilities and geographically redundant storage. Self-hosting requires you to design, implement, and test your own backup strategy — deciding what to back up, how often, where to store backups, how long to retain them, and how to verify that restoration works when needed. This is not optional infrastructure; it is the difference between recovering from a mistake in hours versus losing everything permanently.

Monitoring and alerting: You need to know when your infrastructure is unhealthy before users tell you. Self-hosted platforms require monitoring of system resources (CPU, memory, disk space, network), application health (service availability, response times, error rates), and operational metrics (backup completion, certificate expiry, storage utilization). Setting up meaningful monitoring that generates actionable alerts — not noise — requires careful configuration and ongoing tuning.

Capacity planning: As your organization grows, your infrastructure needs grow with it. More developers mean more repositories, more CI/CD jobs,

more artifacts, more database entries. Self-hosting requires you to anticipate these growth patterns, plan capacity upgrades before resources run out, and architect systems that can scale without requiring complete rebuilds. Hosted providers absorb this complexity – they scale automatically as your usage grows (and bill you accordingly).

Security operations: Beyond patching, self-hosted infrastructure requires ongoing security work: configuring firewalls correctly, managing access controls, rotating credentials, auditing logs for suspicious activity, responding to security advisories, and maintaining compliance with internal and external requirements. For organizations without dedicated security staff, this work often falls to whoever is already managing the infrastructure – stretching their capacity further.

Knowledge continuity: Self-hosted infrastructure creates a dependency on the people who built and maintain it. If those people leave the organization without properly documenting what they did, the infrastructure becomes a liability rather than an asset. Good documentation, runbooks, and knowledge transfer processes are essential but often neglected until someone critical departs unexpectedly.

When You Should Not Self-Host

Self-hosting is not universally superior. For many organizations, hosted services remain the better choice – and pretending otherwise leads to poor decisions that hurt both productivity and security. Here are situations where self-hosting is likely a bad idea:

Small teams without infrastructure expertise: If your organization has fewer than 10 developers and no dedicated DevOps or infrastructure staff, self-hosting will consume a disproportionate amount of time and attention. The operational burden described above does not disappear because you are small; it just gets shouldered by people whose primary job is writing product code. For these teams, the productivity cost of managing infrastructure often exceeds any benefits from data control or cost savings.

Organizations that cannot commit to ongoing maintenance: Self-hosting is not a one-time project that you complete and forget. It is an ongoing operational commitment. If your organization does not have the resources, discipline, or organizational support to maintain infrastructure over years –

applying patches, testing backups, monitoring health, responding to incidents – then self-hosting creates more risk than it mitigates. A neglected self-hosted platform with unpatched vulnerabilities and untested backups is far less secure than a well-maintained hosted service.

Projects where speed matters more than control: Startups in early stages, hackathon projects, proof-of-concept experiments – these situations prioritize velocity over sovereignty. Spending weeks setting up self-hosted infrastructure when you should be validating your product-market fit is a strategic mistake. Use hosted services to move fast; revisit the hosting decision when your project matures and the trade-offs shift.

Teams that cannot justify the investment: If your usage patterns (build minutes, storage needs, user count) fall well below the break-even points discussed earlier, and you do not have compelling security or compliance requirements that mandate self-hosting, then hosted services are simply the more economical choice. Self-hosting for cost savings alone, at low scale, almost always results in higher total cost when labor is properly accounted for.

Situations where hosted features are critical: Some organizations depend on specific features of hosted platforms – GitHub's massive ecosystem of integrations and third-party tools, GitLab.com's built-in SAST/DAST scanning, Bitbucket's tight Atlassian ecosystem integration. Replicating these capabilities in a self-hosted environment may require significant engineering effort or may not be feasible at all. If those features are essential to your workflow, the cost of losing them by self-hosting may outweigh the benefits of sovereignty.

The decision to self-host should be deliberate and informed – driven by specific requirements and risk assessments rather than ideology or reaction to headlines. The rest of this book assumes you have decided that self-hosting is appropriate for your situation and focuses on helping you do it well. If after reading this chapter you conclude that hosted services remain the better choice for now, that is a valid conclusion. Revisit this decision as your organization grows, your requirements change, or the landscape evolves – but make it consciously rather than by accident.

Chapter 2: Git Fundamentals for Infrastructure Operators

You probably already know how to use Git as a developer: clone repositories, create branches, commit changes, push and pull, open pull requests. This chapter is not about those day-to-day operations. It is about what happens on the server side when those operations occur — the internals that matter when you are responsible for hosting Git infrastructure rather than merely using it.

Understanding how Git actually works under the hood changes how you design, operate, and troubleshoot self-hosted infrastructure. It explains why certain configurations exist, why some operations are slow while others are fast, what can go wrong during repository transfers, and how to maintain repository health over time. This knowledge is foundational for everything that follows in this book.

How Git Repositories Actually Work: Objects, References, and Packfiles

A Git repository is not a filesystem of files with metadata attached. It is a content-addressable object store with references layered on top. Understanding this distinction is essential for understanding how Git servers operate.

Objects: Every piece of data in a Git repository — every file content, every directory structure, every commit, every tag — is stored as an object identified by its SHA-1 hash (or SHA-256 in newer configurations). There are four types of objects:

- **Blob objects** store file contents. A blob has no filename, no timestamp, no parent relationship — it is raw bytes identified by the hash of those bytes. If two files anywhere in your repository have identical content, they share a single blob object. This deduplication is one reason Git repositories are more compact than they might appear.

- **Tree objects** represent directory structures. A tree contains entries that map filenames to either blob hashes (for files) or other tree hashes (for subdirectories), along with file mode information. The root tree of a commit represents the entire filesystem snapshot at that point in history.
- **Commit objects** record a point in history. A commit contains a reference to a root tree (the filesystem state), zero or one parent commits (linking it into history), author and committer metadata, a timestamp, and a commit message. Branches and tags ultimately point to commit objects.
- **Tag objects** (annotated tags) provide typed references to commits with additional metadata like signer information and messages. Lightweight tags are simply refs pointing directly to commits without an intervening tag object.

Every object's name is the hash of its complete content — type, size, and data. This means objects are immutable: you cannot change an object without changing its hash, which would make it a different object entirely. This immutability is fundamental to Git's integrity guarantees and its ability to detect corruption.

References: If objects are the data, references are the navigation system. Refs are named pointers to commits (or tags) that provide human-readable names for positions in history. They live in the `.git/refs` directory of a repository:

- **Branch refs** (`refs/heads/main`, `refs/heads/develop`) point to the latest commit on each branch and move forward as new commits are added.
- **Tag refs** (`refs/tags/v1.0.0`) point to specific commits and do not move (by convention).
- **Remote-tracking refs** (`refs/remotes/origin/main`) record what your local repository knows about branches on remote servers.

References are small text files containing the 40-character SHA-1 hash of the commit they point to. When you push a branch update, you are updating a ref on the server to point to a new commit — not uploading an entire history.

Packfiles: Storing every object as an individual file would be extremely inefficient for large repositories with millions of objects. Git solves this by packing objects into packfiles — compressed binary files that store multiple objects together using delta compression. A packfile stores some objects in full

and represents others as deltas (differences) against base objects, dramatically reducing storage requirements for repositories with many similar versions of the same files.

Packfiles live in `.git/objects/pack/` and are accompanied by index files (`.idx`) that allow Git to quickly locate individual objects within a pack without scanning the entire file. When you run `git gc` (garbage collection), Git repacks loose objects into new packfiles, removing unreachable objects and optimizing delta chains for better compression.

For infrastructure operators, this object-store architecture has several important implications:

- Repositories are self-contained and portable — copying all objects and refs from one location to another creates a complete replica.
- Integrity can be verified by recomputing hashes — if an object's stored hash does not match the hash of its contents, corruption has occurred.
- Transfer operations can be highly efficient — clients and servers compare known object hashes and only transfer objects the client does not already have.
- Storage grows with history — unlike some version control systems that discard old versions, Git keeps all objects reachable from any ref, so repository size reflects complete history.

Git Transport Protocols: Local, File, SSH, HTTP, and Git

Git supports multiple transport protocols for communicating between clients and servers. Each protocol has different characteristics regarding authentication, performance, firewall compatibility, and operational complexity. Understanding these differences is essential for configuring self-hosted infrastructure correctly.

Local filesystem: When the remote URL is a local path (e.g., `file:///path/to/repo.git` or simply `/path/to/repo.git`), Git accesses the repository directly through the filesystem. This is the fastest possible transport but requires the client and server to share the same machine or mounted filesystem. Local transport is useful for development workflows and backup operations on a single server but not for networked clients.

SSH: The SSH transport (`git@server.example.com:path/to/repo.git`) invokes Git commands on the remote server over an encrypted SSH connection. For fetches, the client runs `git upload-pack` on the server; for pushes, it runs `git receive-pack`. SSH provides:

- Strong authentication via public keys (the standard method for developer access)
- Encrypted communication protecting both code and credentials
- Wide compatibility — virtually every Git hosting platform supports SSH
- Firewall-friendly — uses port 22 by default, which is commonly allowed

SSH authentication typically uses per-user keys stored in the git user's `~/.ssh/authorized_keys` file. Self-hosted platforms manage these keys through their web interfaces, adding or removing entries as developers register or revoke access. The SSH daemon configuration on the server should restrict the git user to Git operations only — typically using a forced command like `/usr/bin/git-shell -c "$SSH_ORIGINAL_COMMAND"` to prevent shell access.

HTTP/HTTPS: The smart HTTP transport (`https://server.example.com/path/to/re` uses standard web protocols with special endpoints for Git operations. Clients make POST requests to `git-upload-pack` and `git-receive-pack` handlers on the server, which process the data and return packfiles or acknowledgments. HTTPS provides:

- Familiar authentication mechanisms (basic auth, token auth, OAuth)
- Easy firewall traversal — uses ports 80 and 443, universally allowed
- Integration with web-based hosting platforms — the same URL serves both Git operations and the web interface
- SSL/TLS encryption for data in transit

HTTP transport requires a web server configured to handle Git requests. Most self-hosted platforms (Gitea, Forgejo, GitLab) run their own HTTP servers or integrate with Nginx/Apache to serve Git over HTTPS. Authentication is typically handled via personal access tokens rather than passwords for security reasons.

Git protocol: The native Git protocol (`git://server.example.com/path/to.repo.git` uses a dedicated daemon listening on port 9418. It is the fastest transport for read-only access because it has minimal overhead — no encryption, no authentication handshake. However:

- It provides no authentication (repositories are either public or blocked)
- It provides no encryption (data transfers in plaintext)
- It supports fetch only, not push
- Most modern hosting platforms do not enable it by default

The Git protocol is occasionally used for large public repositories where read performance matters and security is less of a concern. For self-hosted development infrastructure handling proprietary code, SSH or HTTPS are almost always preferable.

Protocol version 2: Git introduced protocol v2 in 2018 to address scalability issues with the original protocol. Protocol v2 provides:

- A request-response model that allows servers to control what information is sent
- Support for fetching specific submodules without downloading everything
- Improved handling of large repositories with many refs and shallow clones

Most modern Git clients and servers support protocol v2, though it is not universally enabled by default on all hosting platforms. When configuring self-hosted infrastructure, ensure protocol v2 is available for clients that request it – this improves performance especially for large repositories.

Branching Models and Collaboration Workflows in Practice

How your organization structures branches and collaborates on code has direct implications for server load, storage requirements, and access control policies. While branching strategies are primarily a development concern, infrastructure operators need to understand common patterns to design systems that support them effectively.

Centralized workflow: In its simplest form, all developers clone from a single central repository, make changes on local branches, and push directly to shared branches (typically `main` or `master`). This is straightforward but provides no protection against broken code reaching the main branch – anyone

with write access can push directly. Infrastructure implications are minimal: one repository, straightforward access controls.

Feature branch workflow: Developers create feature branches from `main`, work on their changes, and submit pull or merge requests for review before integrating into `main`. This is the most common pattern in modern development and is what platforms like GitHub, GitLab, Gitea, and Forgejo are optimized for. Infrastructure implications:

- Many short-lived branches per repository, increasing ref count
- Pull request operations generate additional server-side processing (mergeability checks, status aggregations)
- Branch protection rules enforce that only approved merges reach `main`

Gitflow: A more formalized branching model with separate `develop`, `release`, and `hotfix` branches alongside feature branches. Gitflow provides structure for managing parallel development streams but adds complexity – more long-lived branches, more merge operations, more opportunities for branch drift. Infrastructure implications:

- Higher ref count due to multiple persistent branches
- More complex access control requirements (different teams may own different branch types)
- Increased CI/CD workload as pipelines run on multiple branch types

Trunk-based development: Developers make small, frequent changes directly to a single main branch (or short-lived feature branches that merge back within hours or days). This pattern minimizes merge conflicts and simplifies history but requires strong automated testing to catch problems early. Infrastructure implications:

- Lower ref count but higher push frequency
- Heavy reliance on CI/CD for pre-merge validation
- Potentially higher database load from frequent webhook events and status updates

Fork-based workflow: Common in open-source projects, each contributor maintains their own fork of the repository and submits pull requests from their fork to the upstream project. Infrastructure implications:

- Many repository copies (one per contributor), multiplying storage requirements
- Mirror synchronization between forks and upstream when enabled
- More complex access patterns as contributors interact across repository boundaries

The branching model your organization uses affects infrastructure sizing. A team of 50 developers using trunk-based development with one repository generates different load patterns than a team of 50 using Gitflow with 20 repositories and dozens of long-lived branches each. Understanding these patterns helps you provision appropriately and troubleshoot performance issues when they arise.

Remote Operations: Fetch, Push, Clone, and Server-Side Processing

When developers run Git commands that communicate with a server, specific processes execute on the server side. Understanding these processes explains what resources operations consume, what can go wrong, and how to optimize server configuration.

Clone: When a client runs `git clone`, the server executes `git upload-pack` which:

1. Advertises all refs available in the repository
2. Negotiates with the client to determine which objects the client needs (none, for a fresh clone)
3. Creates and sends a packfile containing all reachable objects

A full clone of a large repository can be expensive — it requires reading many objects from disk, computing deltas, compressing them into a packfile, and transmitting potentially gigabytes of data. Servers hosting many clones simultaneously need adequate CPU for compression and sufficient disk I/O bandwidth. Shallow clones (`git clone --depth 1`) reduce this cost significantly by fetching only the most recent commits.

Fetch: When a client runs `git fetch`, the server again executes `git upload-pack` but with negotiation:

1. The client tells the server which commits it already has (via ref tips)
2. The server determines which additional objects are needed to make those commits reachable
3. Only new objects are packed and transferred

Fetch is typically much cheaper than clone because most objects are already on the client. However, fetch operations from many clients simultaneously can still generate significant load – especially if repositories have high commit volume and clients fetch frequently.

Push: When a client runs `git push`, the server executes `git receive-pack` which:

1. Receives new objects from the client in a packfile
2. Unpacks and stores those objects locally
3. Validates that ref updates are allowed (hooks, permissions, branch protection)
4. Updates refs to point to new commits

Push operations are write-intensive – they modify repository state and trigger server-side hooks. Most hosting platforms run hooks for every push: updating issue boards, triggering CI/CD pipelines via webhooks, running code scanning, updating search indexes. A single push can cascade into significant downstream processing. Rate limiting and queueing mechanisms help manage this load when many developers push simultaneously.

Garbage collection: Repositories accumulate unreachable objects over time – commits that were on branches that got deleted, blobs from files that were removed, intermediate packfiles from previous repacks. Periodic garbage collection (`git gc`) removes these objects and optimizes remaining ones into efficient packfiles. On self-hosted platforms, GC is typically scheduled automatically for each repository.

GC is CPU and I/O intensive – it reads all objects, determines reachability, creates new packfiles with optimized deltas, and deletes old files. Running GC on many large repositories simultaneously can saturate server resources. Good hosting platforms stagger GC across repositories and schedule it during low-activity periods.

Large Files and Git LFS Architecture

Git is designed for text-based source code, not large binary files. Storing large files (images, videos, compiled binaries, datasets) directly in Git repositories causes problems: the repository grows rapidly, clones become slow, and every version of every large file remains in history forever consuming space.

Git Large File Storage (LFS) solves this by replacing large files with small pointer files in the Git repository while storing the actual file contents on a separate LFS server. The pointer file contains metadata (file path, OID hash, size) that allows Git LFS to retrieve the real content when needed.

How LFS works:

1. A developer adds a large file to their working directory and stages it with `git add`.
2. Git LFS filters detect the file matches configured patterns and replace its contents with a pointer file before Git stores it.
3. The actual file content is uploaded to the LFS server alongside the Git push.
4. On clone or checkout, Git fetches the pointer files normally, then Git LFS downloads the actual file contents from the LFS server based on pointers.

Infrastructure implications:

- LFS requires additional storage separate from Git objects – typically a filesystem directory or object storage bucket.
- LFS adds another network endpoint that clients must reach (often served through the same hosting platform's HTTP interface).
- LFS storage can grow very quickly – every version of every large file is stored separately with no deduplication across repositories.
- LFS operations add latency to clone, fetch, and checkout commands proportional to the size of tracked files.

For self-hosted platforms like Gitea, Forgejo, and GitLab, LFS storage is configured as a directory path or external storage backend. Capacity planning for LFS requires understanding what types of files developers will store and at what volume – a team storing Docker images in LFS will consume orders of magnitude more storage than a team storing only design mockups.

Repository Maintenance: Garbage Collection, Pruning, and Integrity

Self-hosted Git infrastructure requires ongoing maintenance to keep repositories healthy, compact, and verifiable. Neglected repositories grow inefficient over time — accumulating unreachable objects, fragmented packfiles, stale references — which degrades performance and wastes storage.

Garbage collection scheduling: Most hosting platforms run automatic GC on repositories periodically (weekly or monthly is common). The exact timing and conditions vary:

- Gitea/Forgejo: Automatic GC runs during repository creation and can be scheduled via cron for periodic maintenance.
- GitLab: Runs housekeeping jobs that include GC, with configurable schedules per repository group.

For large deployments with hundreds of repositories, staggering GC across time windows prevents resource contention. Running GC on all repositories simultaneously at midnight creates a spike that can impact user experience. Spreading GC operations across days or using rate limiting ensures maintenance work does not interfere with development activity.

Pruning stale references: When branches are deleted or tags are removed, the refs disappear but objects they pointed to may remain until GC runs. Some hosting platforms also accumulate internal refs (for pull requests, CI/CD pipelines, webhooks) that should be cleaned up when no longer needed. Periodic ref pruning ensures these do not accumulate indefinitely.

Repository integrity verification: Git's content-addressable design makes corruption detectable but does not prevent it — disk errors, incomplete writes, or software bugs can corrupt objects. The command `git fsck` checks repository integrity by verifying that every object's hash matches its contents and that all refs point to valid objects.

For critical repositories, periodic `fsck` runs are a good practice. Some hosting platforms run integrity checks automatically; others require manual scheduling. When corruption is detected early (a few corrupted objects), recovery from backup or mirror is straightforward. Undetected corruption

discovered only when someone needs a specific historical commit can be much harder to resolve.

Packfile optimization: Over time, repositories accumulate many small packfiles from individual pushes and incremental GC runs. Periodic aggressive repacking (`git gc --aggressive`) creates fewer, larger packfiles with better delta compression — reducing storage at the cost of higher CPU usage during the operation. This is most beneficial for large, active repositories with high write volume.

Storage monitoring: Repository storage should be monitored continuously. Unexpected growth can indicate problems: a developer accidentally adding large files without LFS, a script creating excessive branches or tags, a repository importing external data without cleanup. Setting alerts on storage growth rates helps catch these issues before they consume all available disk space and halt operations.

Understanding Git's internals — objects, refs, packfiles, protocols, and maintenance requirements — provides the foundation for everything else in this book. The platforms you deploy, the configurations you choose, and the operational procedures you implement all build on how Git actually works. With that foundation established, we can move on to preparing the infrastructure where your self-hosted platform will run.

Chapter 3: Preparing Your Infrastructure Foundation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Choosing Hardware and Virtualization: From Single Server to Cluster

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Operating System Selection and Minimal Install

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Networking Fundamentals: IPs, Firewalls, Ports, and Segmentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

TLS Certificates: Let's Encrypt, ACME, and Certificate Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Reverse Proxies: Nginx, Caddy, and Traefik for Routing Traffic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Chapter 4: Choosing a Git Hosting Platform

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

The Self-Hosted Git Platform Landscape in 2026

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Gitea: Lightweight, Fast, and Community Driven

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Forgejo: Fork, Philosophy, and Governance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

GitLab CE/EE: Feature-Rich and Complex

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Comparison Matrix: Features, Resources, and Operational Profile

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Decision Framework: Matching Platform to Your Requirements

Chapter 5: Deploying a Git Hosting Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Installation Methods: Package Manager, Docker, Binary, and Source

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Database Configuration: PostgreSQL, MySQL, SQLite Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Storage Layout: Repositories, Avatars, Attachments, and LFS Objects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Initial Configuration and Application Settings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Authentication Setup: Local Accounts, SSH Keys, and Password Policies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

First Repositories, Users, and Access Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Chapter 6: Security, Identity, and Access Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Authentication Strategies: Local, LDAP, Active Directory, and SAML/OIDC

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Setting Up Single Sign-On with OAuth2 and OpenID Connect Providers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Authorization Models: Users, Groups, Teams, and Repository Permissions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Two-Factor Authentication and MFA Enforcement

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Secrets Management for CI/CD and Infrastructure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Server Hardening: SSH Security, Firewall Rules, and Patching Strategy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Vulnerability Management and Security Monitoring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Chapter 7: CI/CD Pipelines and Build Infrastructure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Pipeline as Code: YAML Syntax, Stages, Jobs, and Artifacts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Runner Architecture: How CI Agents Work Under the Hood

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Deploying Self-Hosted Runners: Docker, Shell, and Kubernetes Executors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Build Caching, Dependency Proxies, and Performance Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Testing Integration: Unit, Integration, E2E, and Security Scanning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Deployment Automation: From Pipeline to Production Environment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Chapter 8: Artifact Management and Registries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Why Self-Host Registries: Control, Compliance, and Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Container Registry Options: GitLab Registry, Harbor, and Distribution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Package Repository Management: Maven, npm, PyPI, Go Modules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Vulnerability Scanning in Images and Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Retention Policies, Storage Limits, and Cleanup Automation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Integrating Registries with CI/CD Pipelines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Chapter 9: Backups, Disaster Recovery, and High Availability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Backup Philosophy: What Matters and Why RPO/RTO Drive Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Backing Up Git Repositories: Bare Clones, Bundles, and Mirror Pushes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Database Backups: Logical Dumps, Physical Copies, and Point-in-Time Recovery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Application Configuration and State Backup Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Disaster Recovery Planning and Restoration Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

High Availability Architectures: Load Balancers, Clusters, and Failover

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Chapter 10: Monitoring, Observability, and Logging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

The Three Pillars: Metrics, Logs, and Traces in Infrastructure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

System Monitoring: Prometheus, Node Exporter, and Alertmanager

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Application Metrics: Git Platform Health, CI Runner Status, Queue Depth

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Log Aggregation: Loki, ELK Stack, and Structured Logging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Dashboards and Alerting: What to Monitor and When to Wake Up

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Incident Response Procedures and Postmortem Culture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Chapter 11: Scaling, Performance, and Capacity Planning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Understanding Bottlenecks: CPU, Memory, Disk I/O, and Network

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Scaling the Git Platform: Horizontal vs Vertical Approaches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Database Performance Tuning and Read Replicas

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

CI/CD Scaling: Runner Pools, Dynamic Provisioning, and Queues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Capacity Planning Methodology: Metrics, Growth Curves, and Headroom

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

When Kubernetes Makes Sense (and When It Does Not)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Chapter 12: Migration from Hosted Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Migration Planning: Inventory, Dependencies, and Risk Assessment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Repository Migration Tools and Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Migrating Issues, Wikis, and Project Metadata

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

CI/CD Configuration Translation Between Platforms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

DNS Cutover Strategy and Minimizing Downtime

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Post-Migration Validation and Rollback Procedures

Chapter 13: Advanced Patterns and Federation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Repository Mirroring: Push, Pull, and Automatic Sync Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Federated Git Hosting Across Multiple Sites or Clouds

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Air-Gapped and Highly Regulated Environments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Infrastructure as Code for Your Development Platform

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Reproducible Builds and Supply Chain Security

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Developer Experience: Local Environments, DevContainers, and Tooling

Chapter 14: Complete Reference Architectures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Reference Architecture 1: Single Server for Individuals and Small Teams

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Reference Architecture 2: Multi-Service On-Premises Platform

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Reference Architecture 3: Distributed Enterprise Development Ecosystem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Implementation Blueprint: Phased Rollout Plan

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Ongoing Operations Checklist and Maintenance Schedule

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Future Considerations and Emerging Technologies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/self-hosteddevelopmentinfrastructure.