PLATFORM ENGINEERING IN THE AGE OF AI

DESIGNING, BUILDING, AND OPERATING MODERN INTERNAL DEVELOPER PLATFORMS FOR CLOUD-NATIVE AND AI-POWERED WORKLOADS

- FROM FIRST PRINCIPLES TO PRODUCTION
- REAL-WORLD ARCHITECTURE AND CODE EXAMPLES
- SECURE, SCALABLE, AND AI-READY
- MEASURE, GOVERN, AND CONTINUOUSLY EVOLVE

CLOUD-NATIVE FOUNDATION | DEVELOPER EXPERIENCE | SECURITY BY DESIGN | OBSERVABILITY AND METRICS | AI-POWERED OPERATIONS

BORIS KARPOV

Platform Engineering in the Age of AI

Designing, Building, and Operating Modern Internal Developer Platforms for Cloud-Native and AI-Powered Workloads

Steve Publications

This book is available at
https://leanpub.com/platformengineeringintheageofai

This version was published on 2026-08-25

Contents

Designing, Building, and Operating Modern Internal Developer Platforms for Cloud-Native and AI-Powered Workloads

This book teaches you how to design, build, secure, operate, measure, scale, govern, and continuously evolve modern internal developer platforms that support both conventional cloud-native applications and AI-powered workloads. You will learn platform-engineering principles from first principles, progress through complete production-grade implementations, and understand how artificial intelligence is reshaping what platforms must provide and how they operate. The material assumes general software, cloud, or DevOps knowledge but explains specialized concepts thoroughly. Real code examples, step-by-step walkthroughs, and a progressive reference architecture guide you from foundational infrastructure to AI-enabled platform operations.

Introduction: Why Platform Engineering Matters Now

A software engineer at a mid-sized technology company sits down to build a new feature. She needs a database, authentication, logging, monitoring, a deployment pipeline, environment promotion controls, security scanning, cost tracking, and eventually production support. In an organization without a platform, she spends weeks negotiating with different teams for each capability. She writes Kubernetes manifests she does not fully understand, copies configuration from another service and hopes it is correct, and deploys to production with anxiety because nobody documented what happens when things break.

In an organization with a mature internal developer platform, she opens a developer portal, selects a template for the type of service she is building, fills in a few parameters, and clicks create. Within minutes she has a working application with CI/CD pipelines, Kubernetes deployments, database access, monitoring dashboards, alerting rules, security policies, and documentation all wired together according to organizational standards. When something goes wrong in production, the platform provides observability data that helps her diagnose the issue. She focuses on building features rather than assembling infrastructure.

This is the promise of platform engineering: reducing cognitive load through self-service capabilities so developers can deliver value faster while organizations maintain security, reliability, and cost efficiency at scale.

The advent of artificial intelligence and large language models has fundamentally changed both what platforms must provide and how they operate. Platform teams now need to offer self-service access to AI models, vector databases for retrieval-augmented generation, GPU infrastructure for inference, guardrails for safe AI use, and evaluation pipelines for measuring AI quality. Simultaneously, AI can enhance platform operations through code generation, infrastructure-as-code assistance, incident triage, log analysis, capacity forecasting, and even controlled autonomous remediation.

This book exists because the combination of cloud-native complexity and AI capability requires a new level of platform sophistication. Organizations that treat their internal developer infrastructure as an afterthought or a ticket-processing operation will find themselves unable to support modern workloads, retain engineering talent, or compete with organizations that have embraced platform engineering as a strategic discipline.

What You Will Learn

This book takes you from foundational platform-engineering principles through advanced production architectures and AI-enabled operations. By the end, you will be able to:

- Design an internal developer platform that treats developers as users and reduces their cognitive load through self-service capabilities
- Build cloud-native infrastructure foundations including Kubernetes clusters, networking, storage, identity, and secrets management using infrastructure-as-code and GitOps
- Implement CI/CD pipelines, golden paths, and software templates that standardize application delivery without constraining developer autonomy
- Deploy and operate a developer portal with service catalogs, documentation, search, and self-service actions
- Apply zero-trust security principles, policy as code, admission control, supply-chain security, and compliance automation to platform operations
- Build observability stacks using OpenTelemetry, Prometheus, and Grafana that provide meaningful visibility into both applications and the platform itself
- Define and measure service-level objectives for platforms, track developer experience metrics, and demonstrate platform value through outcomes rather than vanity metrics
- Implement FinOps practices including cost allocation, rightsizing, autoscaling, and developer-facing cost transparency
- Understand large language models, embeddings, vector search, retrieval-augmented generation, model serving, and AI agents at the depth needed to build platform capabilities around them

- Design AI gateways that provide unified access to multiple models with routing, fallbacks, caching, rate limiting, token cost management, and observability
- Build RAG services and vector database infrastructure as reusable platform products that application teams can consume without deep AI expertise
- Implement AI agents that interact safely with platform systems including repositories, CI/CD, cloud APIs, Kubernetes clusters, and incident management through proper identity, least privilege, human-in-the-loop controls, and failure containment
- Integrate AI into the platform engineering lifecycle itself for code generation, infrastructure-as-code assistance, pull-request analysis, incident response, log analysis, capacity forecasting, and developer support while knowing when deterministic automation remains preferable
- Govern AI use through frameworks aligned with NIST AI RMF, the EU AI Act, and organizational requirements including model approval processes, data classification, output validation, and audit trails
- Structure platform teams using patterns from Team Topologies, define product roadmaps, manage stakeholder relationships, prevent bottlenecks, and drive adoption across engineering organizations

How This Book Is Organized

The book progresses systematically through fifteen chapters plus a comprehensive reference blueprint at the end. Chapters build on each other, so reading sequentially is recommended even if you are already familiar with some topics. Each chapter introduces concepts from first principles before advancing to production-grade implementations.

Chapters one through three establish foundation: why platform engineering exists and how it relates to DevOps and SRE, platform-as-a-product thinking and organizational models, and the cloud-native technical bedrock of containers, Kubernetes, and cloud infrastructure.

Chapters four through nine cover the core platform capabilities every modern organization needs: infrastructure as code and GitOps for declarative management, CI/CD pipelines for automated delivery, developer portals and golden paths for self-service, security and governance controls, observability

and reliability engineering, and FinOps for cost management. Each chapter includes complete implementation walkthroughs using widely adopted technologies.

Chapters ten through thirteen address AI specifically. Chapter ten explains machine learning, large language models, embeddings, vector search, RAG architecture, and model serving at the depth platform engineers need. Chapter eleven shows how to build AI capabilities as reusable platform products including model gateways, GPU infrastructure management, RAG services, and inference cost controls. Chapter twelve covers agentic platform engineering: how AI agents can interact with platform systems safely through proper architectures, permissions, human-in-the-loop controls, and failure containment. Chapter thirteen examines AI-assisted operations across the development lifecycle from code generation to incident response, distinguishing where AI adds measurable value from where conventional automation remains better.

Chapters fourteen and fifteen address governance, testing, lifecycle management, and organizational adoption. Chapter fourteen covers comprehensive platform testing including AI evaluation pipelines, API versioning, upgrades, migration patterns, and AI governance frameworks. Chapter fifteen addresses real-world implementation strategies for different organizational contexts including startups, growing engineering organizations, large enterprises, regulated industries, and legacy modernization scenarios.

The book concludes with a complete reference blueprint that synthesizes everything into a realistic production architecture and detailed implementation plan covering organizational assessment, platform strategy, team design, cloud foundations, repositories, APIs, infrastructure as code, GitOps, CI/CD, developer portal, service catalog, golden paths, identity, secrets, security, policy, observability, reliability, FinOps, AI gateways, model services, RAG, agent infrastructure, evaluations, AI security, governance, disaster recovery, scaling, lifecycle management, adoption, metrics, and continuous improvement.

The DevForge Reference Platform

Throughout the book, we build one progressive reference platform called DevForge. We begin with foundational cloud and Kubernetes infrastructure and progressively add capabilities: GitOps, CI/CD, identity, secrets, policies, a developer portal, service catalogs, golden-path templates, observability,

security controls, databases and shared services, self-service environments, AI services, model access, RAG capabilities, agent services, AI evaluations, cost controls, governance, and automated operations.

By the end of the book, DevForge represents a complete production-grade internal developer platform that supports both conventional applications and AI-powered workloads. You can use it as a reference architecture to adapt for your own organization rather than starting from scratch. Each chapter adds capabilities to DevForge, showing how components integrate and how developers ultimately consume the platform.

Technology Choices and Trade-offs

The book uses specific technologies for examples but explains why each was chosen, what alternatives exist, and when another approach might be preferable. Primary choices include Kubernetes for container orchestration, Terraform for infrastructure as code, Argo CD for GitOps, GitHub Actions for CI/CD, Backstage for the developer portal, OPA and Kyverno for policy enforcement, OpenTelemetry with Prometheus and Grafana for observability, HashiCorp Vault for secrets management, and LiteLLM for AI gateway implementation.

These choices reflect widespread adoption, strong documentation, active communities, and proven production use. They are not presented as universally optimal. Organizations should evaluate tools against their specific constraints including existing investments, team skills, compliance requirements, operational capacity, and budget. The principles matter more than the tools, and good platform engineering can be achieved with different technology combinations.

Assumptions and Prerequisites

This book assumes you have general software development knowledge including familiarity with programming concepts, version control using Git, basic Linux command line usage, and fundamental networking concepts such as HTTP, DNS, and TLS. You should understand what containers are at a conceptual level even if you have not extensively worked with them. Prior experience

with cloud platforms, DevOps practices, or infrastructure management is helpful but not required for understanding the material.

Specialized topics including Kubernetes internals, GitOps patterns, policy as code, AI model architectures, agent frameworks, and observability design are explained from first principles. The goal is that any competent software engineer, operations professional, or technical leader can read this book and come away with actionable knowledge.

Version Sensitivity and Rapidly Changing Landscapes

Platform engineering itself is a relatively mature discipline with stable principles, though tools evolve continuously. AI and large language models are changing much faster. Model capabilities, pricing, APIs, frameworks, and best practices shift on timescales of months rather than years. Where information is version-sensitive or rapidly changing, the book notes the date and context explicitly. You should verify implementation details against current official documentation before applying them in production.

The distinction between timeless principles and rapidly changing tools runs throughout the book. Principles such as treating developers as users, reducing cognitive load through well-designed abstractions, maintaining security by default, measuring outcomes rather than activity, and building for operational sustainability will remain relevant regardless of which specific technologies are fashionable. Tools come and go; good platform engineering fundamentals endure.

A Note on AI Hype and Critical Evaluation

Artificial intelligence is transforming software development and platform operations in genuine ways. At the same time, the industry is saturated with exaggerated claims about what AI can do today versus what might be possible in the future. This book evaluates AI capabilities critically, distinguishing production-proven techniques from promising but immature approaches. It explains limitations including hallucinations, nondeterminism, context constraints, latency, cost, privacy concerns, security risks, evaluation difficulty, vendor dependency, and operational complexity.

There are situations where AI adds clear measurable value to platform engineering. There are equally many situations where a simple script, rule-based system, API call, controller pattern, workflow engine, or human judgment remains safer, cheaper, more predictable, easier to audit, and better understood. The book shows you how to evaluate which approach is appropriate for each situation rather than defaulting to AI because it is trendy or avoiding it because it is risky.

Now let us begin with the foundations: understanding where platform engineering came from, why it emerged as a distinct discipline, and what problems it actually solves.

Chapter 1: The Platform Engineering Imperative

Platform engineering did not appear overnight. It emerged gradually as organizations scaled their software delivery operations, encountered recurring problems that earlier approaches could not solve cleanly, and recognized that treating internal infrastructure as a product rather than a service desk yielded better outcomes for developers and the business alike. Understanding this evolution matters because it clarifies what platform engineering is meant to accomplish, where it fits relative to DevOps and SRE, and why the arrival of AI makes platform capabilities more critical than ever.

From Chaos to Craft: How Software Delivery Got Complex

In the early days of software development at many organizations, infrastructure was simple. A handful of servers hosted applications that were deployed by engineers who logged in via SSH, copied files, restarted services, and prayed nothing broke. When systems failed, the same people who built the software also fixed them because there was no separation between roles. This model worked for small teams with few services because the total complexity was manageable and everyone shared context about how things operated.

The introduction of virtualization, then containers, then cloud computing multiplied the number of environments, services, dependencies, and configuration options available to engineers. Kubernetes alone exposes hundreds of resource types and thousands of configuration parameters. Cloud providers offer dozens of managed services each with their own APIs, quotas, networking models, and security controls. A single microservice might depend on a container runtime, an orchestrator, multiple cloud services for databases and caching, an ingress controller, a service mesh, monitoring tools, logging systems, secrets management, CI/CD pipelines, artifact registries, policy engines, and identity providers.

This complexity created several interconnected problems. First, individual teams began reinventing infrastructure solutions independently because no centralized capability existed that met their needs conveniently. One team built its own deployment tooling while another wrote custom Terraform modules while a third manually configured cloud consoles. The result was fragmentation: different standards, inconsistent security postures, duplicated effort, and knowledge silos where only specific individuals understood how particular systems operated.

Second, the cognitive load on developers increased dramatically. Engineers who wanted to focus on building features now needed to understand networking policies, Kubernetes admission controllers, certificate rotation, database connection pooling, autoscaling thresholds, cost allocation tags, compliance requirements, and incident response procedures. Many lacked the time or interest to develop deep expertise in all these areas. The consequence was slower delivery as developers spent more time figuring out infrastructure than writing application code, along with more mistakes as people configured things incorrectly due to insufficient understanding.

Third, organizations found themselves stuck between two unsatisfactory extremes. On one side, giving every team complete autonomy produced the fragmentation and inconsistency described above. On the other side, centralizing all infrastructure decisions in a single operations team created bottlenecks where development velocity depended on how quickly that team could respond to tickets and requests. Neither approach scaled well as organizations grew.

Platform engineering emerged as a response to these specific problems. It provides a middle path: centralized investment in shared capabilities that are exposed through self-service interfaces so teams can move quickly without reinventing infrastructure or waiting for approvals, while maintaining consistency, security, and operational standards across the organization.

DevOps, SRE, and the Seeds of Platform Thinking

Platform engineering draws directly from three predecessor disciplines: DevOps, Site Reliability Engineering, and cloud engineering. Understanding their relationships clarifies what platform engineering adds rather than simply replacing existing practices with a new label.

DevOps began as a cultural and procedural movement aimed at breaking down barriers between development and operations teams. The core insight was that organizations which treated development and operations as separate silos with conflicting incentives would suffer from slow delivery, poor quality, and blame-shifting when systems failed. DevOps advocated for shared responsibility: the people who built software should also be responsible for its operation in production, often summarized as you build it you run it. DevOps emphasized automation of repetitive tasks, continuous integration and deployment, infrastructure as code, monitoring and feedback loops, and a culture of experimentation and learning from failures rather than punishing individuals for mistakes.

DevOps succeeded at changing how many organizations thought about software delivery. It reduced handoffs between teams, accelerated feedback cycles, and popularized practices that became industry standards. However, DevOps as originally conceived did not fully address what happens when an organization scales to dozens or hundreds of teams all practicing DevOps independently. Without some form of shared platform, each team ends up choosing its own tools, writing its own automation, and developing its own patterns. The cultural benefits of DevOps remain but the operational complexity multiplies.

Site Reliability Engineering emerged at Google in the late 1990s as a way to apply software engineering principles to operations problems. Rather than treating infrastructure management as manual procedures performed by specialized operators, SRE treats reliability as an engineering discipline with measurable objectives, automated solutions, and systematic approaches to reducing toil. Key SRE concepts include service-level indicators that measure actual user-facing behavior, service-level objectives that define acceptable reliability targets, error budgets that quantify how much unreliability is tolerable before reliability work takes priority over feature development, and blameless postmortems that focus on system improvements rather than individual accountability.

SRE introduced rigor to reliability engineering and gave organizations concrete frameworks for balancing speed of delivery against system stability. However, SRE teams at scale often find themselves supporting many different application teams with varying needs and maturity levels. Without a platform layer, SREs can become overwhelmed by repetitive support tasks across multiple services rather than focusing on cross-cutting reliability improvements that benefit everyone.

Cloud engineering focuses on designing, provisioning, and managing infrastructure using cloud provider capabilities. Cloud engineers understand networking architectures, identity and access management, storage options, compute models, cost structures, compliance requirements, and disaster recovery strategies specific to environments like AWS, Azure, or Google Cloud Platform. They build the foundational infrastructure that applications depend on but often do not directly engage with the developer experience of consuming that infrastructure.

Platform engineering synthesizes insights from all three disciplines. It takes the cultural emphasis on collaboration and automation from DevOps, applies the rigor of measurable reliability objectives from SRE, leverages cloud engineering expertise for infrastructure design, and packages everything into self-service capabilities that reduce cognitive load for development teams. As Luca Galante of Weave Intelligence describes it, platform engineering addresses scale inefficiencies through an internal developer platform using a platform-as-a-product approach, complementing rather than replacing DevOps culture and SRE practices.

The Cloud Native Computing Foundation defines platform engineering as the practice of planning and providing computing platforms to developers and users, encompassing people, processes, policies, technologies, and the business outcomes that drive them. The CNCF further notes that platform engineering emerged from the cross-functional cooperation promised by DevOps as an explicit form of that cooperation in enterprises where scale made informal collaboration insufficient.

The Cognitive Load Crisis in Modern Engineering

Cognitive load is a concept from psychology describing the amount of mental effort being used in working memory. Human cognitive capacity is limited: researchers estimate that most people can hold approximately seven items in working memory at once, and complex tasks require more than simple item retention. When cognitive load exceeds available capacity, performance degrades, errors increase, and stress rises.

In software engineering, cognitive load comes from many sources. Developers must understand the business domain they are working in, the architecture of the systems they maintain, the programming languages and frameworks

they use, the testing strategies they employ, the deployment processes they follow, the monitoring tools they rely on, the security requirements they must satisfy, and the organizational policies they need to comply with. Each new technology, tool, or process adds to this load.

When organizations lack a coherent platform, cognitive load compounds through several mechanisms. First, developers encounter inconsistency: different services use different deployment configurations, different monitoring setups, different secret management approaches, different database access patterns. Understanding how one service operates does not transfer directly to understanding another because each team made independent choices. Developers must learn the specifics of each system they interact with rather than relying on predictable patterns.

Second, developers face discovery problems: figuring out what tools exist, where documentation lives, who owns particular systems, how to request access to resources, which processes apply to their work. Time spent searching for information is time not spent building features, and incomplete information leads to mistakes.

Third, developers deal with fragmentation: their workflow requires switching between multiple unrelated tools and interfaces. They might use GitHub for code, Jenkins for CI/CD, a cloud console for infrastructure, Slack for communication, Jira for tracking work, Confluence for documentation, Prometheus for metrics, PagerDuty for alerts, and Vault for secrets. Each tool has its own login, navigation model, terminology, and quirks. Context switching between tools fragments attention and slows progress.

Platform engineering addresses cognitive load through several deliberate strategies. Standardization reduces the number of different patterns developers must learn by establishing conventions that apply across services. Self-service capabilities eliminate the need to request resources from other teams or wait for approvals for routine operations. Developer portals provide a single entry point for discovering tools, accessing documentation, creating new services, and managing existing ones. Golden paths offer opinionated but well-supported workflows that handle complexity behind the scenes so developers can follow a clear route from idea to production without assembling infrastructure manually.

The goal is not to eliminate all cognitive load: understanding systems deeply is essential for building them well. The goal is to reduce unnecessary cognitive

load so developers can focus their mental energy on problems that matter rather than on figuring out how to deploy code or configure monitoring.

What Platform Engineering Is and Is Not

Platform engineering has become popular terminology, which means it gets used in different ways by different people. Clarifying what platform engineering actually entails helps avoid confusion with related concepts and prevents organizations from applying the label incorrectly.

Platform engineering is the discipline of designing, building, and maintaining internal developer platforms that provide self-service capabilities for software development teams. These platforms abstract infrastructure complexity through well-designed interfaces while enforcing organizational standards for security, reliability, observability, and cost management. Platform engineering treats the platform as a product with users who have needs that must be understood, requirements that must be prioritized, experiences that must be measured, and feedback that must be incorporated into continuous improvement.

A platform team is a group of engineers whose primary responsibility is building and operating this internal platform. They work like a product team serving internal customers rather than like a service desk responding to tickets or like a governance body enforcing rules through approval processes. Their success is measured by how well the platform enables development teams to deliver value quickly and reliably, not by how many requests they process or how many policies they enforce.

Platform engineering is not simply DevOps with a new name. DevOps remains relevant as a cultural approach emphasizing collaboration between development and operations. Platform engineering builds on that culture by providing shared infrastructure that makes collaboration easier at scale. An organization can practice DevOps principles without having a formal platform team, especially if it is small enough that informal coordination works adequately.

Platform engineering is not SRE either, though the two are closely related. SRE focuses on reliability engineering practices for specific services or systems. Platform engineering builds the shared infrastructure and tooling that make it easier for many teams to practice SRE principles consistently.

In mature organizations, platform teams and SRE teams work together: the platform team provides capabilities like monitoring infrastructure, incident management tools, deployment automation, and policy enforcement, while SRE teams apply reliability practices to critical services and contribute learnings back into the platform.

Platform engineering is not a centralized ticket-processing operation. One of the most common anti-patterns in organizations attempting platform engineering is creating a platform team that becomes a bottleneck where all infrastructure requests must go through. This defeats the purpose: developers are still blocked waiting for approvals, cognitive load has not been reduced because they still need to understand what to request and why, and the platform team spends its time on repetitive tasks rather than building better self-service capabilities. A real platform reduces tickets by automating common workflows so developers can serve themselves.

Platform engineering is not about hiding all infrastructure complexity from developers. Over-abstraction creates problems: developers who do not understand how their systems are deployed cannot troubleshoot effectively when things go wrong, they make poor architectural decisions because they cannot see resource implications, and they become dependent on the platform team for every change that falls outside predefined templates. Good platforms expose enough information and control for developers to understand and operate their systems while abstracting away repetitive complexity that does not require individual decision-making.

Platform engineering is not a one-time project with a completion date. Platforms must evolve continuously as developer needs change, new technologies emerge, organizational requirements shift, and operational experience reveals areas for improvement. Treating platform building as a project with a fixed scope and timeline leads to platforms that are outdated before they are fully adopted.

How AI Changes the Platform Equation

The emergence of large language models and generative AI has introduced fundamental changes to what platforms must provide and how they can operate. These changes affect both the capabilities developers need from platforms and the tools platform teams use to build and run those platforms.

From the developer perspective, AI introduces new workload types that traditional platforms were not designed to support. Applications now incorporate large language models for natural language processing, code generation, content creation, and decision support. These applications require access to model APIs, vector databases for retrieval-augmented generation, GPU infrastructure for running inference, prompt management systems, evaluation pipelines for measuring output quality, guardrails for ensuring safe behavior, and cost tracking for monitoring token consumption. Developers building AI-powered features need platform capabilities that make these resources available through the same self-service interfaces they use for conventional infrastructure.

AI also changes how developers interact with existing platform capabilities. AI coding assistants like GitHub Copilot help developers write application code faster by suggesting completions based on context. These tools can extend to platform interactions: generating Kubernetes manifests from natural language descriptions, creating Terraform configurations for infrastructure requirements, writing CI/CD pipeline definitions, producing monitoring queries, or drafting incident response procedures. The platform must be designed so that AI assistance is effective: clear documentation, consistent patterns, well-defined APIs, and predictable behavior all make it easier for AI tools to generate correct outputs.

From the platform team perspective, AI offers new capabilities for automating operations and reducing toil. AI can analyze logs and metrics to detect anomalies that indicate emerging problems before they cause outages. It can assist with incident response by suggesting root causes based on historical data and current symptoms. It can optimize resource allocation by forecasting capacity needs based on usage patterns. It can generate documentation from code and configuration. It can answer developer questions about platform capabilities by searching knowledge bases and providing accurate responses.

At a more advanced level, AI agents can interact with platform systems autonomously to perform routine operations: creating development environments on request, promoting deployments through environments after automated tests pass, remediating known issues by executing predefined procedures, or triaging support tickets by categorizing them and suggesting responses. These capabilities are promising but require careful design to ensure safety: agents must operate within well-defined boundaries, respect least privilege principles, include human approval for high-risk actions, provide

full audit trails, and have rollback mechanisms when things go wrong.

The Cloud Native Computing Foundation has begun discussing platform engineering evolution for AI-native workloads, referring to an emerging framework sometimes called Platform Engineering 2.0 that expands traditional platform capabilities to support AI-era requirements including model access, GPU orchestration, vector search infrastructure, agent runtimes, evaluation systems, and safety controls.

AI also introduces new risks that platforms must address. Prompt injection attacks can manipulate AI behavior to extract sensitive information or perform unauthorized actions. Data leakage can occur when developers inadvertently send confidential information to external model providers. Model hallucinations can produce incorrect outputs that applications treat as factual. Excessive autonomy in AI agents can lead to unintended consequences if agents take actions beyond their intended scope. Cost□□ can happen when poorly constrained AI usage consumes large volumes of tokens or GPU resources without visibility or controls.

A mature platform engineering organization treats these AI capabilities and risks as first-class concerns rather than afterthoughts. It provides approved model access through gateways that enforce policies, it manages GPU infrastructure efficiently to avoid waste, it offers RAG templates that follow security best practices, it implements evaluation pipelines that catch quality regressions before they reach production, it governs agent behavior through identity, permissions, and approval controls, and it monitors AI usage for both cost anomalies and security incidents.

The combination of cloud-native complexity and AI capability makes platform engineering more important than ever. Organizations that invest in building strong internal developer platforms will find themselves better positioned to adopt AI safely and effectively because they have the governance, observability, automation, and self-service foundations that make managing AI workloads tractable at scale. Organizations that neglect platform engineering will struggle with fragmented AI adoption, inconsistent security, uncontrolled costs, and operational chaos as teams experiment independently without shared standards or support.

Platform engineering is not a luxury for well-funded organizations with large engineering teams. It is a practical necessity for any organization serious about delivering software reliably, efficiently, and securely in an in-

creasingly complex technological environment. The next chapter examines how to approach platform engineering as a product discipline rather than an infrastructure project, because getting the mindset right matters as much as getting the technology right.

Chapter 2: Platform-as-a-Product

Building a successful internal developer platform requires more than assembling tools and writing automation scripts. It requires treating the platform as a product with real users who have needs that must be understood, experiences that must be designed intentionally, quality that must be measured continuously, and value that must be demonstrated to stakeholders. The difference between a platform that developers love using and one that becomes another bureaucratic hurdle often comes down to whether the team building it thinks like product engineers or like infrastructure administrators.

Treating Developers as Customers

The foundational principle of platform-as-a-product is simple but frequently violated: your internal developers are users, not subordinates. They have goals they are trying to achieve, constraints they are working within, pain points that slow them down, and preferences about how they want to work. A platform team that ignores these realities will build capabilities that look impressive on architecture diagrams but fail to solve actual problems developers face in their daily work.

Understanding developer needs requires deliberate effort. Platform teams should conduct user research through interviews, surveys, observation sessions, and feedback channels just as external product teams do. What tasks do developers spend the most time on? Where do they encounter friction? What information is hard to find? Which tools are confusing or unreliable? What would make their jobs easier? The answers often differ from what platform engineers assume based on their own experience.

One effective approach is the Jobs-to-be-Done framework, which focuses on understanding the progress developers are trying to make in specific situations rather than asking them what features they want. For example, a developer might say they need better Kubernetes documentation when what they actually need is the ability to deploy a service quickly without having to understand Kubernetes at all. The job is get my code running in a test

environment so I can demonstrate functionality to stakeholders, not learn container orchestration.

Another approach borrowed from product management is maintaining a prioritized backlog of platform capabilities based on user impact and effort required. Platform teams should regularly review this backlog with developer representatives, explain what they are working on and why, solicit feedback on priorities, and adjust based on changing needs. Transparency about the roadmap builds trust and helps developers understand that the platform team is focused on their problems rather than pursuing technology for its own sake.

Product thinking also means accepting that not every developer need can or should be met through platform capabilities. Platforms must make trade-offs between standardization and flexibility, between ease of use and power, between supporting common workflows and accommodating edge cases. Good product teams are explicit about these trade-offs and communicate them clearly to users so expectations are realistic.

A critical aspect of treating developers as customers is measuring satisfaction and acting on feedback. If developers consistently report that a particular platform capability is difficult to use or unreliable, the platform team should prioritize fixing it rather than building new features. Platforms that ignore user feedback create resentment and drive developers toward shadow IT solutions that bypass platform controls entirely.

Platform Teams and Operating Models

How platform teams are structured within an organization significantly affects their effectiveness. Different models work better in different contexts depending on organizational size, existing maturity, cultural norms, and the specific problems being addressed. Understanding the options helps organizations choose an approach that fits their situation rather than copying structures from companies with fundamentally different circumstances.

The Team Topologies framework by Matthew Skelton and Manuel Pais provides a useful lens for thinking about platform team structure. It defines four fundamental team types: stream-aligned teams that own end-to-end delivery of business capabilities, enabling teams that help other teams build new skills or adopt new technologies, complicated-subsystem teams that manage components requiring specialized expertise, and platform teams that

provide self-service capabilities to support stream-aligned teams with minimal friction.

In this model, platform teams serve as an enabling layer for the majority of engineering organizations which should be organized as stream-aligned teams around business domains or value streams. Platform teams reduce cognitive load by providing standardized infrastructure, tooling, and processes so stream-aligned teams can focus on their specific domains without needing deep expertise in every underlying technology. The interaction between platform and stream-aligned teams should follow an X-as-a-Service pattern where the platform provides capabilities through well-defined interfaces rather than requiring close collaboration for every request.

Several concrete operating models emerge from this framework depending on organizational context:

The embedded model places platform engineers within individual product or domain teams rather than in a centralized group. These engineers work closely with developers to understand their needs and build platform capabilities that directly address those needs. This model works well in smaller organizations where close collaboration is easy and the total number of teams is limited. It risks fragmentation as different embedded engineers may make inconsistent choices about tools and patterns.

The centralized model creates a dedicated platform team separate from product teams. This team owns all platform capabilities and serves all development teams through self-service interfaces. Centralization enables consistency, focused investment, and deeper specialization in platform technologies. It risks creating distance between platform builders and platform users if the team does not actively engage with developer needs.

The hub-and-spoke model combines centralized and embedded approaches. A central platform team defines standards, maintains core capabilities, and provides expertise, while platform engineers embedded in product teams adapt those standards to local contexts and provide hands-on support. This model balances consistency with flexibility but requires strong coordination to avoid divergence.

The federated model distributes platform ownership across multiple semi-autonomous teams, each responsible for a specific domain of platform capabilities such as CI/CD, observability, security, or AI infrastructure. These teams collaborate through shared standards and interfaces while maintaining

independence in their domains. This model scales well in large organizations but requires mature communication and governance practices.

Regardless of which model an organization chooses, several principles apply universally. Platform teams must maintain strong connections to the developers they serve through regular interaction, transparent communication, and responsive feedback loops. They must measure their success by developer outcomes rather than internal metrics such as number of tickets resolved or features shipped. They must balance standardization with flexibility, providing clear defaults while allowing exceptions when justified. They must invest in documentation, training, and support so developers can use platform capabilities effectively without constant assistance.

A common anti-pattern to avoid is the ticket-processing bottleneck where the platform team becomes a centralized approval body that every infrastructure request must pass through. This model kills autonomy, slows delivery, and frustrates developers who feel blocked by bureaucracy rather than enabled by the platform. The goal of platform engineering is to reduce tickets by automating common workflows so developers can serve themselves, not to create more tickets by centralizing decisions.

Developer Personas and User Research

Different developers have different needs from a platform based on their roles, experience levels, domains, and workflows. Understanding these personas helps platform teams design capabilities that serve the broadest range of users effectively rather than optimizing for a narrow subset.

Common developer personas in organizations with internal platforms include:

The application developer builds business logic and features for specific products or services. This person typically has strong programming skills but may have limited infrastructure expertise. They need simple, well-documented ways to deploy code, access databases and external services, configure environments, view logs and metrics, and troubleshoot issues without becoming a Kubernetes expert or cloud architect.

The platform engineer designs, builds, and maintains the internal developer platform itself. This person has deep expertise in infrastructure technologies, automation, security, and operations. They think in terms of abstractions, APIs,

patterns, and scalability. Their needs include powerful tooling for managing complex systems, clear interfaces for exposing capabilities to other developers, and mechanisms for enforcing standards consistently.

The DevOps engineer or SRE focuses on reliability, performance, and operational excellence for specific services or the platform itself. This person bridges development and operations, understanding both application behavior and infrastructure dynamics. They need observability tools that provide deep visibility, automation capabilities for reducing toil, and access to infrastructure controls for tuning and remediation.

The security engineer ensures that applications and infrastructure meet organizational security requirements and external regulatory obligations. This person thinks in terms of risk, compliance, vulnerabilities, and controls. They need platform capabilities that enforce security policies automatically, provide audit trails for compliance evidence, enable vulnerability scanning and remediation workflows, and support incident response when security issues arise.

The data engineer or AI engineer builds data pipelines, machine learning models, or AI-powered features. This person has specialized needs including access to compute resources for training and inference, data storage and processing capabilities, model deployment infrastructure, experiment tracking, and evaluation tools. They may require GPU clusters, vector databases, or specialized frameworks that differ from conventional application workloads.

The engineering manager or technical lead oversees teams of developers and is responsible for delivery outcomes, team health, and architectural quality. This person needs visibility into team progress, platform adoption metrics, cost information, reliability indicators, and risk factors. They use the platform less directly than individual developers but depend on it enabling their teams to work effectively.

User research helps platform teams understand these personas in their specific organizational context. Techniques include:

Interviews provide deep qualitative insights into developer experiences, pain points, and aspirations. Conducting thirty to fifty interviews across different personas typically reveals recurring themes that inform platform priorities.

Surveys reach a broader audience quickly and can quantify satisfaction levels, usage patterns, and feature requests. Well-designed surveys with

specific questions yield more actionable data than open-ended requests for feedback.

Observation sessions where platform engineers watch developers work reveal friction points that developers may not articulate in interviews because they have become normalized as part of daily life. Seeing someone struggle to find documentation or navigate between tools is more revealing than hearing about it secondhand.

Usage analytics from the platform itself show which capabilities are used frequently, which are ignored, where users drop off in workflows, and how long tasks take to complete. These data help validate whether platform investments are delivering value.

Feedback channels such as dedicated Slack channels, email addresses, or in-product feedback buttons enable ongoing collection of user input without requiring formal research sessions. The key is responding to feedback visibly so users know their input matters.

The goal of understanding personas and conducting user research is not to build everything everyone wants but to prioritize platform investments based on real needs rather than assumptions. A platform that solves the most important problems for the largest number of developers well will achieve higher adoption and deliver more value than a platform that attempts to serve every possible use case at mediocre quality.

Platform Metrics That Matter: DORA, SPACE, and Beyond

Measuring platform effectiveness requires metrics that reflect actual outcomes rather than activity or output. Counting the number of features shipped, tickets resolved, or documents written tells you nothing about whether the platform is helping developers deliver value more quickly, reliably, or enjoyably. Good metrics align with business objectives while providing actionable insights for continuous improvement.

The DORA metrics, developed by Google Cloud and published annually in the Accelerate State of DevOps Report, have become the industry standard for measuring software delivery performance. The four key metrics are:

Deployment frequency measures how often an organization successfully releases to production. Elite-performing organizations deploy on demand, multiple times per day, while low-performing organizations deploy monthly or less frequently. Higher deployment frequency correlates with faster feedback cycles, smaller batch sizes, and reduced risk per deployment.

Lead time for changes measures the elapsed time from code commit to that code running in production. Elite performers achieve lead times of less than one hour while low performers experience lead times measured in weeks or months. Shorter lead times indicate streamlined workflows with fewer bottlenecks.

Change failure rate measures the percentage of deployments causing a failure in production such as requiring rollback, hotfix, or patch. Elite performers experience change failure rates below fifteen percent while low performers see rates above thirty percent. Lower failure rates indicate better quality practices including testing, review, and gradual rollout strategies.

Time to restore service measures how long it takes to recover from a production failure. Elite performers restore service in under an hour while low performers take days or longer. Faster restoration indicates effective incident response capabilities including monitoring, runbooks, automation, and cross-team coordination.

The 2024 DORA report surveyed more than thirty-nine thousand professionals globally and found that these four metrics remain validated as effective measures of software delivery performance. The report also introduced a fifth metric called rework rate measuring the proportion of unplanned deployments made to fix user-visible issues, reflecting growing concern about quality alongside speed.

DORA metrics provide a useful starting point for platform teams because they measure outcomes that platforms should directly influence. A well-designed platform that provides fast CI/CD pipelines, reliable deployment automation, effective testing frameworks, and good observability tools should improve all four metrics over time. However, DORA metrics alone do not capture the full picture of platform value because they focus on delivery performance without addressing developer experience, cost efficiency, or security posture.

The SPACE framework addresses this gap by providing a research-backed approach to measuring developer productivity across five dimensions: Sat-

isfaction, Performance, Activity, Communication and Collaboration, and Efficiency and Flow. Developed by researchers at GitHub, Microsoft, and the University of Victoria and published in ACM Queue in 2021, SPACE explicitly rejects single-metric approaches to productivity measurement that create perverse incentives and gaming behavior.

Satisfaction measures how fulfilled developers feel with their work, team, tools, and culture. It captures well-being, engagement, and retention risk. High satisfaction correlates with better performance and lower turnover but requires qualitative measurement through surveys rather than system metrics alone.

Performance measures the impact of development work on the business, including quality, value delivered, and reliability. This dimension overlaps with DORA metrics but also includes customer-facing outcomes such as feature adoption, user satisfaction, and revenue impact.

Activity measures visible work output such as commits, pull requests, code reviews, and deployments. While tempting as an objective metric, activity alone is a poor proxy for productivity because it does not capture quality or impact and can incentivize quantity over value. SPACE recommends using activity metrics cautiously and always in context with other dimensions.

Communication and Collaboration measure how effectively developers work together, share knowledge, and coordinate across teams. Metrics might include documentation discoverability, time to integrate contributions, response times for reviews, and cross-team dependency resolution speed.

Efficiency and Flow measure how smoothly work moves through the development process without interruptions, delays, or context switching. Metrics include cycle time, wait time between workflow stages, interruption frequency, and flow efficiency ratios comparing active work time to total elapsed time.

The SPACE framework recommends combining instrumented data from systems with perceptual data from surveys to create a balanced view that includes both objective outcomes and subjective experience. Engineering organizations with mature metrics programs typically implement DORA first to establish baseline delivery metrics, then add SPACE dimensions to monitor team health, collaboration patterns, and developer experience.

Beyond DORA and SPACE, platform teams should track metrics specific to platform adoption and value:

Platform adoption rate measures what percentage of development teams actively use the platform versus relying on ad-hoc processes or shadow IT. Low adoption indicates that the platform is not solving real problems or is too difficult to use.

Time to first deployment measures how long it takes a new developer or team to deploy their first application using the platform. This metric captures onboarding friction and the effectiveness of documentation and templates.

Self-service ratio measures what percentage of common infrastructure requests are completed through self-service capabilities versus requiring platform team assistance. A high self-service ratio indicates that automation is working effectively and reducing bottlenecks.

Platform reliability measures uptime, latency, and error rates for the platform itself. Developers cannot trust a platform that is frequently unavailable or slow, so platform SLOs must be as rigorous as application SLOs.

Cost per deployment or cost per developer tracks infrastructure and tooling costs normalized by usage to ensure the platform remains economically sustainable as it scales.

The key principle across all metrics is measuring outcomes that matter to developers and the business rather than vanity metrics that look impressive but do not drive improvement. A platform team should regularly review its metrics with stakeholders, explain what they mean, identify areas for improvement, and demonstrate progress over time. Metrics are not for judging teams but for understanding system behavior and guiding investment decisions.

Maturity Models and Adoption Strategies

Organizations rarely start with a mature platform from day one. Most begin with fragmented tooling, manual processes, and inconsistent practices, then gradually invest in platform capabilities as they encounter scaling problems and recognize the value of standardization. Understanding maturity progression helps organizations assess their current state, identify gaps, and plan realistic improvement paths without attempting to boil the ocean.

The CNCF Platform Engineering Maturity Model provides a framework for evaluating platforms across five aspects: Investment, Adoption, Interfaces, Operations, and Measurement. It defines four progressive levels: Provisional, Operationalized, Scalable, and Optimizing.

At the Provisional level, platform capabilities are ad-hoc and inconsistent. Different teams use different tools and processes. There is no centralized investment in shared infrastructure. Documentation is sparse or nonexistent. Platform-related work competes with feature development for team capacity without clear prioritization. Many organizations begin here because they have not yet recognized platform engineering as a distinct discipline worth dedicated investment.

At the Operationalized level, some shared capabilities exist and are used consistently by at least a subset of teams. Basic automation handles common workflows such as CI/CD or infrastructure provisioning. Documentation covers core capabilities. A designated team or individuals take ownership of platform maintenance. However, coverage is incomplete, many workflows still require manual intervention, and the platform may not be designed for scale.

At the Scalable level, the platform provides comprehensive self-service capabilities used by most development teams. Infrastructure is provisioned through code with consistent patterns across environments. Security policies are enforced automatically. Observability is standardized. The platform team operates like a product team with a roadmap, backlog, and regular engagement with users. The platform can support significant growth in the number of teams and services without proportional increases in platform team size.

At the Optimizing level, the platform continuously evolves based on data-driven insights into developer needs, usage patterns, and business outcomes. Advanced capabilities such as AI-assisted development, automated remediation, predictive capacity planning, and intelligent cost optimization are integrated. The platform actively contributes to competitive advantage by enabling faster innovation cycles than competitors can match. Few organizations reach this level fully because it requires sustained investment over many years.

As Martin Fowler notes regarding maturity models, the true outcome is not what level you are at but the list of things you need to work on. Maturity assessment is valuable as a diagnostic tool for identifying improvement areas, not as a badge of honor or a destination. Organizations should focus on making incremental progress in areas that matter most to their specific context rather than chasing arbitrary maturity scores.

Adoption strategies must match organizational reality. Large enterprises with hundreds of teams and legacy systems face different challenges than startups with ten engineers and greenfield architectures. Regulatory constraints

in industries like healthcare, finance, or government impose requirements that technology companies can ignore. Cultural norms around autonomy versus standardization vary widely.

A pragmatic adoption approach typically follows several phases:

Discovery and assessment involves understanding current pain points, mapping existing tools and processes, identifying quick wins that demonstrate value, and building a case for investment based on specific problems rather than abstract principles.

Minimum viable platform focuses on the smallest set of capabilities that solve the most painful problems for the largest number of developers. This might include standardized CI/CD pipelines, a simple deployment process, basic monitoring, and clear documentation. The goal is rapid demonstration of value rather than comprehensive coverage.

Expansion and standardization builds on initial success by adding capabilities iteratively based on user feedback and usage data. Golden paths are defined for common workload types. Security controls are integrated into workflows rather than bolted on afterward. The developer portal provides a unified entry point for platform capabilities.

Maturation and optimization refines the platform continuously based on metrics, feedback, and changing requirements. Advanced capabilities such as AI services, automated operations, and sophisticated cost management are added when they solve real problems rather than because they are trendy. The platform team operates with product discipline, prioritizing investments that deliver measurable value.

Throughout all phases, communication is critical. Developers need to understand why the platform exists, what problems it solves, how to use it, and how to provide feedback. Leadership needs to understand the investment required, the timeline for seeing results, and the metrics that demonstrate progress. Platform teams that operate in secrecy or assume adoption will happen automatically typically fail because developers continue using familiar processes rather than learning new ones.

Platform engineering as a product discipline requires ongoing commitment to understanding users, measuring outcomes, iterating based on feedback, and demonstrating value. The technology choices matter, but the mindset matters more. A platform built with expensive tools but designed without empathy for

developer needs will fail. A platform built with simple tools but designed around real developer problems can transform how an organization delivers software.

The next chapters move from principles to practice, covering the technical foundations that every modern internal developer platform must provide: cloud-native infrastructure, containers, Kubernetes, networking, storage, and the shared services that applications depend on. Understanding these foundations is essential before building the abstractions and automation layers that make them accessible to developers through self-service interfaces.

Chapter 3: Cloud-Native Foundations

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

Containerization and the Reproducible Runtime

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

Kubernetes Architecture and Core Concepts

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

Cloud Infrastructure Patterns and Multi-Account Design

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

Networking, DNS, TLS, and Ingress

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

Storage, Databases, and Shared Services

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

Chapter 4: Infrastructure as Code and GitOps

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

Declarative Infrastructure and State Management

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

Terraform Modules and Workspace Patterns

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

GitOps Principles and Controllers

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

Argo CD Implementation Walkthrough

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

Drift Detection, Policy Enforcement, and Rollback

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

Chapter 5: CI/CD and Delivery Pipelines

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

Pipeline Architecture and Design Principles

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

GitHub Actions Implementation Walkthrough

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

GitLab CI and Jenkins Alternatives

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

Artifact Management and Supply Chain Security

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

Environment Promotion Strategies

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

Chapter 6: Developer Portals and the Paved Road

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

The Developer Portal as Platform Front Door

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

Backstage Implementation Walkthrough

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

Service Catalogs and Ownership Metadata

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

Golden Paths and Software Templates

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

Reducing Cognitive Load Through Abstraction

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

Chapter 7: Platform Security and Governance

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

Zero Trust for Internal Platforms

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

Identity Federation and Workload Identity

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

Secrets Management Patterns

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

Policy as Code with OPA and Kyverno

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

Software Supply Chain Security and SBOMs

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

Chapter 8: Observability and Reliability Engineering

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

The Three Pillars Plus Events

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

OpenTelemetry Implementation Walkthrough

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

Metrics, Dashboards, and Alerting with Prometheus and Grafana

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

SLOs, SLIs, and Error Budgets for Platforms

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

Incident Management and Post-Incident Learning

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

Chapter 9: FinOps and Cost Engineering

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

Unit Economics of Cloud Infrastructure

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

Kubernetes Resource Efficiency and Rightsizing

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

Showback, Chargeback, and Developer Transparency

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

Capacity Planning and Autoscaling Strategies

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

Storage and Network Cost Optimization

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

Chapter 10: AI Foundations for Platform Engineers

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

From Traditional ML to Foundation Models

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

Large Language Models: Capabilities and Limitations

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

Embeddings, Vector Search, and Semantic Retrieval

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

Retrieval-Augmented Generation Architecture

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

Model Serving, Inference Infrastructure, and APIs

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

Chapter 11: Building AI Platform Capabilities

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

Self-Service Model Access and AI Gateways

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

GPU and Accelerator Infrastructure Management

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

RAG Services and Vector Database Operations

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

Model Deployment and Evaluation Pipelines

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

Token Economics and Inference Cost Controls

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

Chapter 12: Agentic Platform Engineering

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

Agent Architectures and Tool Calling

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

Permissions, Identity, and Least Privilege for Agents

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

Human-in-the-Loop Controls and Approval Gates

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

Multi-Agent Workflows and Orchestration

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

Failure Containment, Rollback, and Safety Patterns

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

Chapter 13: AI-Assisted Platform Operations

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

AI in the Development Lifecycle

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

Infrastructure as Code Generation and Review

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

Incident Triage and Root Cause Analysis with AI

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

Log Analysis, Anomaly Detection, and Capacity Forecasting

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

When Deterministic Automation Beats AI

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

Chapter 14: Platform Governance, Testing, and Lifecycle

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

Comprehensive Platform Testing Strategies

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

AI Evaluation Pipelines and Release Gates

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

API Versioning and Backward Compatibility

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

Platform Upgrades and Migration Patterns

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

AI Governance Frameworks and Controls

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

Chapter 15: Enterprise Adoption and Architectures

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

Organizational Assessment and Context Analysis

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

Build-Versus-Buy Decisions Across the Platform Stack

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

Team Structures and Operating Models

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

Phased Implementation Roadmaps

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

Anti-Patterns and Pitfalls to Avoid

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

Conclusion: Building Platforms That Endure

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

Platform Engineering in the Age of AI Reference Blueprint

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

Organizational Assessment Guide

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

Platform Product Strategy

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

Team Design and Operating Model

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

Cloud and Kubernetes Foundations

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

Repository Structure

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

Platform APIs and Custom Resources

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

Golden Paths for Common Workloads

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

Security Architecture Summary

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

Observability Architecture Summary

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

FinOps Architecture Summary

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

AI Platform Architecture Summary

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

Phased Implementation Roadmap

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

Production-Readiness Checklist

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

Security Checklist

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

AI Governance Checklist

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

Troubleshooting Guide

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

Glossary

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

References and Bibliography

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