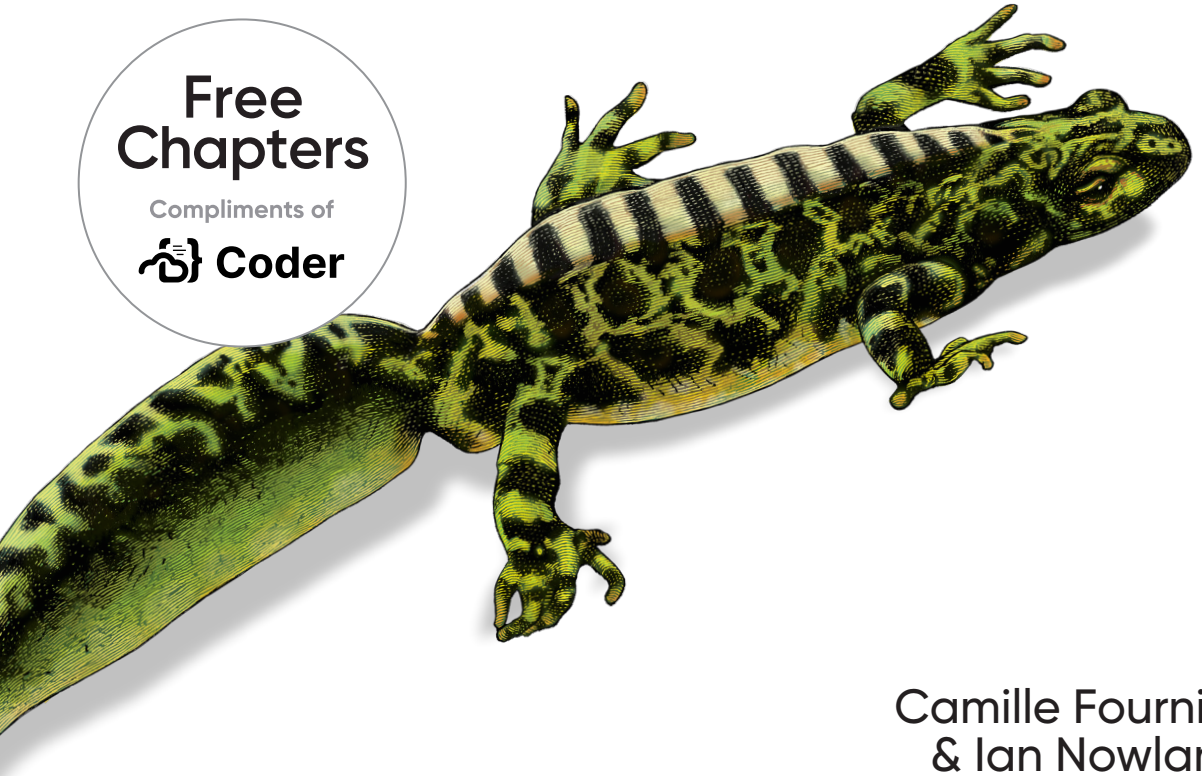
O'REILLY®

Platform Engineering

A Guide for Technical, Product, and People Leaders



Camille Fournier
& Ian Nowland
Foreword by Nicole Forsgren

Coder | CLOUD DEVELOPMENT ENVIRONMENTS

Onboard developers to new projects **in minutes (not weeks)**

Get started

OPEN SOURCE | SELF-HOSTED | IDE AGNOSTIC

Platform Engineering

A Guide for Technical, Product, and People Leaders

This excerpt contains Chapters 1, 12, and 13. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Camille Fournier and Ian Nowland

O'REILLY®

Platform Engineering
by Camille Fournier and Ian Nowland

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: David Michelson
Development Editor: Virginia Wilson
Production Editor: Kristen Brown
Copyeditor: Rachel Head
Proofreader: Piper Editorial Consulting, LLC
Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery

October 2024: First Edition

Revision History for the First Edition
2024-10-08: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098153649* for release details.

978-1-098-15364-9

[LSI]

Table of Contents

Foreword from Coder

Complexity is a constant in today's development workflows. Whether it's juggling cloud/private infrastructure, managing developer environments, or ensuring security compliance, teams are stretched thin trying to keep everything running smoothly. Platform engineering steps in to solve this, creating a framework that cuts through complexity and allows your teams to focus on innovation and operations.

Cloud development environments (CDEs) are a relatively new addition to the platform engineering space. They take the burden of administration off developers by providing fast, standardized environments that eliminate the need for local setup, help streamline workflows, and ensure consistency and security across teams. With CDEs, teams don't waste time configuring environments—they get straight into writing code and delivering value.

This book gets into the nuts and bolts of how platform engineering, combined with CDEs, allows teams to manage complexity rather than be overwhelmed by it. The real value lies in reducing friction at every step—whether it's spinning up environments or integrating tools—so your teams can focus on delivering real business value, and not wasting time with administration.

If you're dealing with fragmented systems, slow onboarding, or the constant grind of managing infrastructure, this book will show you how to turn that chaos into control. It's not just about tools; it's about building a system that works for your team, not against it.

CHAPTER 1

Why Platform Engineering Is Becoming Essential

She swallowed the cat to catch the bird, she swallowed the bird to catch the spider, she swallowed the spider to catch the fly; I don't know why she swallowed a fly—Perhaps she'll die!

—Nursery rhyme

Over the past 25 years, software organizations have experienced a problem: what to do with all of the code, tools, and infrastructure that is shared among multiple teams? In reaching for a solution, most have tried creating central teams to take responsibility for these shared demands. Unfortunately, in most cases this has not worked particularly well. Common criticisms have been that central teams provide offerings that are hard to use, they ignore customer needs in favor of their own priorities, their systems aren't stable enough, and sometimes all of the above.

Instead of fixing these central teams, some have tried getting rid of them entirely, giving each application team access to the cloud and their choice of open source software (OSS). However, this exposes those application teams to the operational and maintenance complexity of their choices, so instead of creating efficiencies and economies of scale, even small teams end up needing site reliability engineering and DevOps specialists. And even with these dedicated specialists, the cost of managing the complexity continues to threaten the productivity of the application teams.

Others, while embracing the best of the cloud and OSS, have not given up on central teams; they've stuck with the model, certain that the benefits outweigh the downsides. The best have succeeded by building platforms: developing shared offerings that other engineers can comfortably build on top of. They have become experts at managing the complexity of the cloud and OSS while presenting stability to their users, and they are willing to listen to and partner with the application teams to continually evolve and meet the company's needs. Whether or not they've called their

efforts platform engineering, they embody the mindset, skills, and approach necessary for solving the problem of ever-growing complexity (the fly) without swallowing ever-larger animals in the process.

To set the stage, in this chapter we'll cover:

- What we mean by platforms, and a few other important terms we'll use throughout the book
- How system complexity has gotten worse in the era of cloud computing and OSS, leaving us in an "over-general swamp" of exposed complexity
- How platform engineering manages this complexity and so frees us from the swamp

This chapter has a slight emphasis on infrastructure and developer tooling, but don't worry, this book isn't just for people working on infrastructure or developer platforms! We'll use systems common to all developers to provide a tangible illustration of the current state of affairs, but the underlying challenge of managing complexity is common to all kinds of internal platform development.

Defining "Platform" and Other Important Terms

Before we get started, let's define several important terms we'll be using throughout this book, so we all have the same frame of reference:

Platform

We use Evan Bottcher's definition from 2018, with a couple of terms updated. A platform is a foundation of self-service APIs, tools, services, knowledge, and support that are arranged as a compelling internal product. Autonomous application teams[1] can make use of the platform to deliver product features at a higher pace, with reduced coordination.

A corollary here is to ask: what, then, isn't a platform? Well, for the purposes of this book, a platform requires you to be doing platform engineering. So, a wiki page isn't a platform, because there's no engineering to be done. "The cloud" also is not a platform by itself; you can bring cloud products together to create an internal platform, but on its own the cloud is an overwhelming array of offerings that is too large to be seen as a coherent platform.

1 We'll sometimes call these teams your "users" or "customers," if it makes more sense in the context.

Platform engineering
: Platform engineering is the discipline of developing and operating platforms. The goal of this discipline is to manage overall system complexity in order to deliver leverage to the business. It does this by taking a curated product approach to developing platforms as software-based abstractions that serve a broad base of application developers, operating them as foundations of the business. We will elaborate on this in Chapter 2.

Leverage
: Core to the value of platform engineering is the concept of leverage—meaning, the work of a few engineers on a platform team reduces the work of the greater organization. Platforms achieve leverage in two ways: making applications engineers more productive as they go about their jobs creating business value, and making the engineering organization more efficient by eliminating duplicate work across application engineering teams.

Product
: We believe that it is essential to view a platform as a product. Developing platforms as compelling products means that we take a customer-centric approach when deciding on the features of a platform. This implies a core focus on the users, but it requires more than just performatively hiring product managers and calling it a day. With the word "product" we strive to achieve for platforms what Steve Jobs created with Apple products: against a broad range of demand for features the product is deliberately and tastefully curated, both through what it does and, more importantly, through what it leaves out.

The Over-General Swamp

There are many types of internal platforms, and the advice in this book is relevant to all of them. However, we see the most acute pain today in the infrastructure and developer tooling (DevTools) spaces, and we see this driving the most demand for platform engineering. That is because these systems are the ones most closely integrated with the public cloud and OSS. These two trends have driven a lot of industry change over the last 25 years, but rather than making things uniformly better, they are increasing the ownership costs of systems over time. They make applications easier to build but harder to maintain, and the more your system grows, the slower you get—like you're walking through a swamp.

This comes back to the economic realities of writing and maintaining software. You might believe that the major cost of software is associated with the act of writing it. In fact, most of the cost is related to its upkeep, support, and maintenance.[2] Estimates

2 For a good diagram of the software lifecycle, see *https://oreil.ly/iDM5u*.

suggest that at least 60–75% of the lifetime cost of software accrues after initial development, with about a quarter of that dedicated purely to migrations and other "adaptive" maintenance.[3] Between required upgrades for security patches, retesting of the software, migrations to new versions of underlying dependencies, and everything else, software costs a lot of engineering time in maintenance overhead.

Rather than reducing maintenance overhead, the cloud and OSS have amplified this problem, because they provide an ever-growing layer of *primitives*: general-purpose building blocks that provide broad capabilities but are not integrated with one another.[4] To function, they need "glue"—our term for the integration code, one-off automation, configuration, and management tools. While this glue holds everything together, it also creates stickiness, making future changes much harder.

The *over-general swamp* forms as the glue spreads. Each application team makes independent choices across the array of primitives, selecting those that allow them to quickly build their own applications with the desired cutting-edge capabilities. In their rush to deliver, they create whatever custom glue is needed to hold everything together, and they're rewarded with praise for shipping fast. As this repeats over time, the company ends up with the type of architecture seen in Figure 1-1.

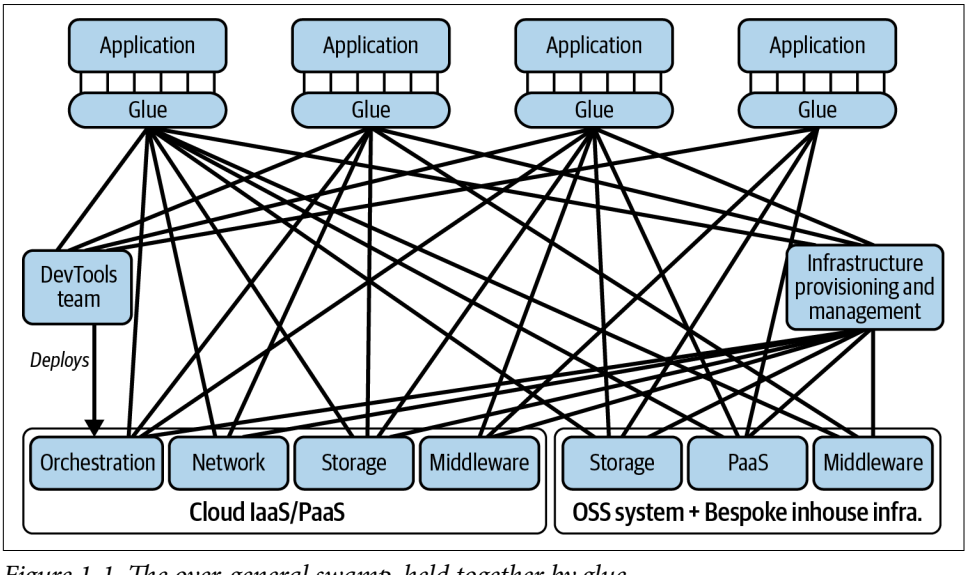


Figure 1-1. The over-general swamp, held together by glue

3 See Jussi Koskinen's paper on software maintenance costs at *https://oreil.ly/EFNZ6*.

4 This is literally what they were called in the 2003 AWS vision document (see *https://oreil.ly/n4ie_*).

The problem with the swamp isn't just the messy architecture diagram; it's how difficult it is to change that sticky mess over time. That's important because applications are constantly evolving, due to new features or operational requirements. Every OSS and cloud primitive also undergoes regular changes, and all of this requires updating the glue that binds them. With the glue smeared everywhere, seemingly trivial updates to primitives (say, a security patch) require extensive organization-wide engineering time for integration and then testing, creating a massive tax on organizational productivity.

The key to avoiding this situation is to constrain how much glue there is, which aligns with the old architectural principle of "more boxes, fewer lines." Platforms allow us to do this, and thus to extract ourselves from the swamp. By abstracting over a limited set of OSS and vendor choices in an opinionated manner, specific to your organizational needs, they enable separation of concerns. You end up with an architecture more like Figure 1-2.

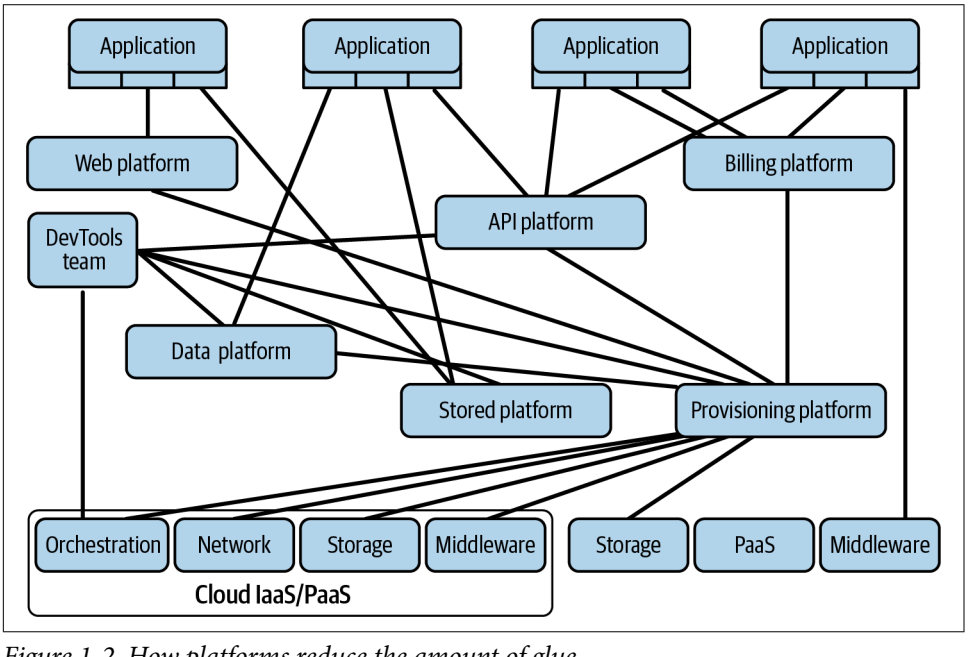


Figure 1-2. How platforms reduce the amount of glue

In sum, platforms constrain the amount of glue needed by implementing the concepts of abstraction and encapsulation and creating interfaces that protect users from underlying complexity (including the complexity of an implementation that needs to change). These concepts are about as old as computer science itself—but if they're so well known, why does the industry need platform engineering? To answer that

question, we'll start with a look at how enterprise software engineering has changed over the last quarter century.

How We Got Stuck in the Over-General Swamp

The software industry has changed immensely over the past 25 years, kicking off with the widespread use of the internet. For those of you who have been in the industry for a while, we don't need to tell you how much this affected every aspect of software development, but for the relative newcomers, it's no exaggeration to say that the over-general swamp largely exists due to the internet itself and the pressure to ship more, faster, without failure. Let's look at the key changes that led to us getting stuck here, and the implications of that result.

Change #1: Explosion of Choice

The internet generated incredible demand for new software, and software has to run on hardware, no matter what the name "serverless" might imply. The initial wave was realized by provisioning a lot more hardware in data centers, and this led to the growth of infrastructure engineering. Every company was buying a lot more servers and network gear, negotiating with their data center providers, installing hardware in ever greater quantities all across the world—big I infrastructure doing big E engineering powering the big I internet.

We don't want to minimize the challenges that were overcome in this relatively short period of time. However, application developers interacting with infrastructure teams were constantly frustrated by the extent of hardware issues they had to deal with. They suffered from a limited but constantly changing menu of server choices, frequent data center capacity issues, and weird hardware-related operational problems that no one would help debug—the common response was "nothing in the system logs, must be your software."

It's no surprise that when the public cloud came along, frustrated application developers were eager to jump over to a world where they could call an API and seemingly control their own destiny. Despite reasonable concerns about the architectural complexity, security risks, reliability, and cost, even large, conservative companies were driven to some level of cloud adoption.

Unfortunately, those reasonable concerns have proven not just valid, but worse than feared. While the cloud promised platforms (PaaS) that would make applications independent of infrastructure, what has seen wide adoption is IaaS, which in many cases has tied applications to infrastructure even more than before. Reminding you of the difference:

- With *infrastructure as a service* (IaaS), the vendor's APIs are used to provision a virtualized computing environment with various other infrastructure primitives, which run an application more or less like it would be run on physical hosts.
- With *platform as a service* (PaaS), the vendor takes full ownership of operating the application's infrastructure, which means rather than offering primitives, they offer higher-level abstractions so that the application runs in a scalable sandbox.

Figure 1-3 shows a high-level comparison of the two approaches.

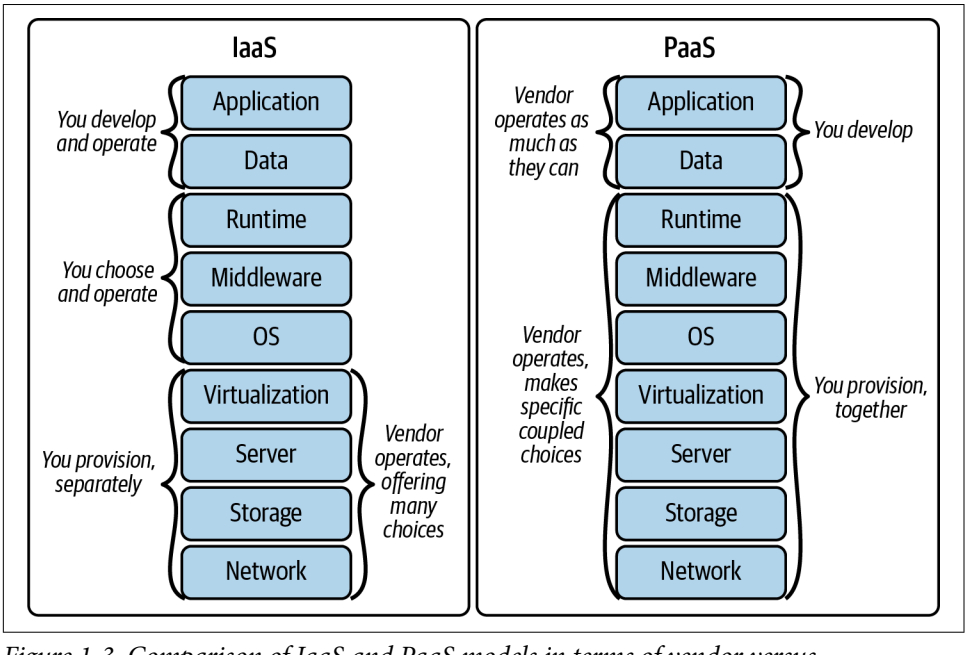


Figure 1-3. Comparison of IaaS and PaaS models in terms of vendor versus customer responsibility

Initially, it was hoped that application teams would embrace fully supported PaaS offerings—solutions as user-friendly as Heroku but capable of handling greater complexity.[5] Unfortunately, these platforms have struggled to support a wide range of applications and to integrate with existing applications and infrastructure. As a result, almost all companies doing in-house software development at scale embrace IaaS to run that software, preferring to accept the added complexity of provisioning and operating their infrastructure in order to get the flexibility of choice.

5 Other full service PaaSes that failed to see broad success were Force.com, AWS Elastic Beanstalk, and Google AppEngine. As a result, vendors often use the term *PaaS* for more flexible offerings, which means they need to be combined with other IaaS and so have similar problems around complexity.

The rise of the orchestration system Kubernetes is in many ways an admission that both PaaS and IaaS have failed to meet enterprise needs. It is an attempt to simplify the IaaS ecosystem by forcing applications to be "cloud native" and thus need less infrastructure-specific glue. However, for as much as it standardizes, Kubernetes has not been a complexity win. As an intermediary layer trying to support as many different types of compute configurations as possible, it is a classic "leaky" abstraction, requiring far too much detailed configuration to support each application correctly. Yes, applications have more YAML glue and less Terraform glue,[6] but as we've discussed, a goal of platform engineering is to reduce the total amount of glue.

Kubernetes is also an example of the second source of complexity we mentioned. Matching the rise of the cloud has been the rise of OSS ecosystems for all types of software. Where once you paid a vendor for your development tools and middleware, now there are thriving and evolving ecosystems for a wide array of development tools, libraries, and even full independent systems like Kubernetes. The problem with OSS is the proliferation of choice. Application teams with specific needs can usually find an OSS solution that is optimal for them but not necessarily for anyone else at the company. The bespoke choice that lets them quickly ship their initial launch eventually turns into a burden, as they must independently manage the maintenance costs that came with their "free, like a puppy"[7] OSS choice.

Change #2: Higher Operational Needs

In parallel with this explosion of infrastructure primitives and applications using them came the question of who was going to operate them, and how. If you went back to the 1990s, before the internet took off, and looked at how companies developed and operated their in-house software applications, you would typically find two roles, which in most cases were staffed in entirely separate teams:

Software developer
: Responsible for architecture, coding, testing, etc., leading to software applications being delivered as monolithic distributions, handed off to someone else to operate

Systems administrator
: Responsible for all aspects of the production operation of software (in-house applications as well as vendor software and OSS) on the company's computers

As the internet took off and in-house software became more important to companies' success, these roles started to mutate. The importance of 24/7 operational support

6 We will discuss what this looks like in Chapter 2.

7 As per former Sun Microsystems CEO Scott McNealy, alluding to the long-term cost of adopting either OSS or puppies.

for an increasing number of applications initially led to the growth of *operations engineering* teams, which tended to be filled with a lot of early-career systems administrators—this was the proving ground they had to face before graduating into a less operational role.

You still see pockets of operations engineering in companies today, but the role is declining. As the 2000s progressed, software developers adopted the "Agile" model of regular releases of incremental functionality, as a better way to get feedback and so ship a better product. Agile brought a challenge to the operations engineering model: with one team taking on all the responsibility for making code changes and pushing for fast release cycles and the other team taking on all the frontline responsibilities when the code had problems, there was some tension. As anyone who lived through it knows, "some tension" is putting it mildly; particularly after an outage caused by something that had been "thrown over the fence," there was usually a large amount of finger-pointing about which side was to blame. The problem was that there was generally no clear answer, because Agile had blurred the lines of responsibility.

This led to the creation and broad adoption of what the industry now calls *DevOps*. DevOps was framed as a model to integrate application development and operations activities, and it became associated as much with a culture change as a set of specific technologies or roles to adopt. That being said, the operational work didn't go away, and on the ground teams implemented it in two different ways:

Split
: Keep the separation between operations and development teams, but have the operations team do some amount of development, particularly around creating glue for pushing code to production and integrating it with infrastructure. Thus, the old operations team with operations engineers was now the DevOps team with DevOps engineers.

Merged
: Merge the operations and development teams into one. With this approach, described as "you build it, you run it," everyone who works on a system is on the same team, with all of them sharing in the operational work (the most salient aspect being part of the on-call rotation). While many teams succeeded with 100% software developers, others were more cross-functional, with specialists to own the glue that pushed code to production and integrated with infrastructure. At some companies, these engineers were also called DevOps engineers.[8]

In an act of parallel evolution, in about 2004 Google moved away from operations engineering toward something they called *site reliability engineering* (SRE). In 2015, during the upswing of DevOps popularity, Google published a book on its practices,

8 In other companies, they were called systems engineers or systems development engineers.

Site Reliability Engineering: How Google Runs Production Systems (O'Reilly). This caused a lot of excitement, because while many companies had been adopting DevOps, plenty were struggling with the practical complexities of making it work. With its heavy emphasis on reliability-oriented processes and organizational responsibilities, some thought SRE was the silver bullet the industry needed to finally balance operational and development needs, enabling the creation of much more reliable systems.

We would argue that SRE, as it was originally sold, has not been a widespread success outside of Google. The processes were too heavyweight; their success relied too much on the specific cultural capital and organizational focus that came from Google being the world's biggest search company. This was well summarized by former director of SRE at Google, Dave O'Connor, who after a couple of stints outside Google wrote a post in 2023 titled "6 Reasons You Don't Need an SRE Team" that concludes, "The next stage in removing our production training wheels as an industry is to tear down the fence between SRE and Product Engineering, and make rational investments in reliability as a mindset, based on specific needs."

There is no getting away from the needs of operating software. Every company that offers online software systems must have operational support for this software during applicable usage times (which may be working hours, 24/7, or somewhere in between). But how do you manage this in the most cost-effective yet sustainable way possible? You want to limit the places where you must have dedicated operations teams (or, using the terminology introduced earlier, "split" DevOps/SRE teams) and make it as easy as possible for the developers of the software to deploy and operate it themselves, achieving the initial vision of DevOps.

Result: Drowning in the Swamp

So you've got more application teams, making more choices, over a more complex set of underlying OSS and cloud primitives. Application teams get into this situation because they want to deliver quickly, and using the best systems of the day that fit the problem (or the systems they know best) will help them do that. Plus, if they've gotta own all the operational responsibility for this system themselves, they might as well pick their own poison!

Add to this that application engineers with new features are not the only ones wanting to ship as quickly as possible. The increasing surface of internet-accessible systems has led to an escalation of cyberattacks and vulnerability discoveries, which in turn means that infrastructure and OSS are changing faster to address these risks. We've seen upgrade cycles for systems and components move from years to months, and these changes mean work for application teams who must update their glue and retest or even migrate their software in response.

The pressure for change has created a swampy mess of glue mixed with the long-term consequences of individual team decisions. Every new greenfield project adds more choices and glue to this bog of complexity, and over time your developers get stuck in the mire. It's hard to navigate, slow to move through, and full of hungry operational alligators (or worse, crocs!). How do you extract yourself from this morass? It's no surprise that we think the answer is platform engineering, and next we will cover the ways in which it helps you do just that.

How Platform Engineering Clears the Swamp

If you've been stuck in the over-general swamp, you can appreciate the intellectual appeal of platform engineering. You're hiring more people in roles like infrastructure, DevTools, DevOps, and SRE engineer, but you never seem able to keep up with the new complexity arising from OSS and cloud systems. Your applications grow more complex, your application developers become less productive, and you need a way out. Building platforms to manage this complexity sounds great.

But building platforms takes significant investment. This includes the costs to build and support them, as well as the overhead associated with limiting application teams' choices of OSS and cloud primitives. Additionally, establishing a platform engineering team can incur organizational costs through reorganizations, role changes, and the overhead of rolling out a new focus area for the company. In this section, we explain how platforms and platform engineering will justify these investments and deliver long-term value.

Limiting Primitives While Minimizing Overhead

The explosion of choice wasn't all bad: greenfield applications can ship much faster now than in the past, and application developers feel more autonomy and ownership when they have systems they enjoy using. These benefits often get forgotten when companies start to focus on reducing the support burden and long-term costs that arise from the diversity of choices. In this situation, the first instinct of leadership is to prescribe a set of standards using appeals to authority. "Because I am the expert in databases," they say, "I will choose which databases you, the application teams, can use." Or, "I am the architect, so I decide on all of the software tools and packages." Or, "I am the CTO, so I decide everything." Inevitably, these experts will struggle to understand the business needs well enough to make optimal choices, and application teams will suffer. Standardization via authority isn't enough.

Platform engineering recognizes that modern engineering teams should have systems that they enjoy using, provided by teams that are responsive to them as customers and not just focused on cost reduction or their own support burden. Instead of prescribing a set of standards based on appeals to authority, platform engineering takes a customer-focused product approach that curates a small set of primitives able to meet

a broad range of requirements. This requires compromises in light of business realities, incremental delivery of good platform architecture, and a willingness to partner directly with application teams and listen to what they need. When done well, you can point to the demonstrated leverage of partnering to use the platform-provided offerings instead of appealing to the authority of the architect, database administrator, CTO, or platform VP. In this way, you can reduce the number of OSS and cloud primitives used, without the worst consequences of top-down mandates.

Reducing Per-Application Glue

On top of reducing the number of primitives in use, platform engineering aims to go one step further and reduce the coupling "glue" to those that remain. This removes most of the application-level glue, by abstracting the primitives into systemic platform capabilities that are able to meet broader needs. To illustrate this, we'll dive into the common challenge of managing Terraform.

OSS and cloud offerings are complex in a lot of ways, with one of the most costly ways being their configuration—the endless lists of parameters that, if not specified correctly, will eventually lead to issues in production. Nowhere is this more of a problem than in cloud configuration, for which the 2024 state-of-the-art tool is an OSS infrastructure as code (IaC) system called Terraform that provides a perfect illustration of how platform engineering addresses the downsides of glue.

When application engineering teams all started pushing hard for the smorgasbord of the IaaS cloud, most companies decided that the path of least friction was to give each team the power and responsibility to provision their own individual cloud infrastructure with their own configuration. In practice, that meant they became part-time cloud engineering teams, versed in configuration management and infrastructure provisioning. If you want infrastructure that is repeatable, rebuildable, and can be secured and validated, you need a configuration management and provisioning template like Terraform. So, the common approach was to have application development teams learn Terraform. In our experience, this led to the following progression:

1. Most engineers don't want to learn a whole new toolset for infrequent tasks. Infrastructure setup and provisioning are not an everyday core focus—not even for teams doing mature resiliency testing and regularly rebuilding the system from scratch. So, over time the work would get shunted either to unsuspecting new hires, or to the rare engineers who were interested in DevOps. In the best case this would lead to one or two people evolving into infrastructure provisioning experts who could write Terraform and own all of this for the team. However, most of the time these engineers didn't stick around on application teams for long, which pushed the work back onto new hires, who usually made a mess of it.

2. The shortage, combined with people cobbling together their own Terraform all over the company, often led leadership to centralize the work across multiple teams (or even the whole company). But rather than centralizing with the goal of building a platform, all the Terraform engineers were just pulled into a team that provided Terraform-writing services.
3. These centralized Terraform-writing teams became trapped in a feature shop mindset, taking in work requests and pumping them out. This meant no strong developers (the type that can change the structure of the Terraform to provide better abstractions) wanted to be part of it. Over time, the codebase devolved into a spaghetti mess, which slowed down application teams who wanted something slightly out of the norm and eventually created a security nightmare.

A better path is to realize that you need to do something more coherent than offer centralized Terraform-writing support, and think about how to evolve this group of experts from a "glue" maintenance center into an engineering center that builds things—namely, a platform. This will require you to go one level deeper in understanding your customers' needs, to develop opinions about which solutions to offer rather than just trying to make it easier for people to get access to whatever they want, and to think about what you can build that takes you beyond just the provisioning step.

As you move into new models for providing underlying infrastructure, it is important to centralize expertise and create efficiencies. Instead of each engineering team hiring their own DevOps and SRE engineers to support the infrastructure, a platform team can pool these experts and expand their remit to identifying broader solutions for the company. This not only supports the one-off changes but permits their expertise to be leveraged to create platforms that abstract the underlying complexity. This is where the magic starts to happen.

Centralizing the Cost of Migrations

We will mention migrations often in this book, as we believe managing migrations is an important part of a platform's value. Applications and primitives have long but independent lifetimes, during which they each undergo many changes. The combination of these changes creates high maintenance costs. Platform engineering reduces these costs by:

Reducing the diversity of OSS and cloud systems in use
: The fewer primitives you have, the less likely it is that you'll need to do a migration because of one.

Encapsulating OSS and vendor systems with APIs
: While platform APIs are often imperfect at encapsulating all aspects of the OSS and vendor systems they leverage, even "good enough" APIs that abstract a lot of their implementation will allow the platform to protect its applications from needing to change when the underlying systems change.

Creating observability of platform usage
: Platforms can provide various mechanisms to standardize collection of metadata around both their own use and that of underlying OSS and vendor systems. This visibility into the dependency state of the applications using your platform should allow you to ease the burden of upgrades when those dependencies need to change.

Giving ownership of OSS and cloud systems to teams with software developers
: When APIs are later shown to be imperfect, unlike traditional infrastructure organizations, platform teams have software developers who can write the non-trivial migration tooling that makes the migration transparent to most application teams.

Allowing Application Developers to Operate What They Develop

The goal of mature DevOps was to simplify accountability through a "you build it, you own it" approach. Despite this having been a popular idea for over a decade, many companies have not managed to execute on this model. We believe that, for those that have succeeded, a major contributor to this success is the leverage that their platforms provide through abstracting the operational complexity of underlying dependencies.

No one loves being on call. But when teams are only on call for issues caused by their own applications, we have found that a surprising number are willing to take on operational responsibility. After all, why wouldn't they stand behind the business-critical systems they spend their days creating? For too many companies, however, the operational problems caused by the infrastructure, OSS, and its glue completely dominate the problems in the application code itself.

An example of this can be seen as applications seeking higher resiliency are deployed across multiple availability zones, cloud regions, or data centers. This leaves application teams exposed to intermittent cloud provider issues such as networking problems, and the 2 a.m. alerts that inevitably follow. Platform engineering addresses this by building resilient abstractions that can handle application failover on behalf of the application teams, reducing the number of late-night wakeup calls they receive.

When most of the underlying systems' operational complexity is hidden behind platform abstractions, this complexity can be owned and operated by your platform team. This requires you to limit the options that you support, so that you can push

the abstraction boundary upward into a core set of offerings, each handling a broad set of application use cases. It also requires that you have high operational standards within your platform team, so that application teams are comfortable relying on them.

Yes, building and operating platforms that handle these issues is hard, especially when it comes to getting application teams to accept limitations on their choices. But the only alternatives are either directly exposing your entire organization to these issues or perpetuating your use of operations teams (by any name), and so in turn perpetuating the accountability problems, negative impact on agile development, and finger-pointing.

Empowering Teams to Focus on Building Platforms

If you want to leverage OSS and vendor primitives but reduce the complexity that slows progress later, you need teams that can build platforms to manage those primitives and their complexity. There are four platform-adjacent approaches that are popular today, all of which bring valuable skills to the organization, but none of which are set up to have the combination of focus and skills needed for building platforms. Table 1-1 summarizes these approaches and why they are not adapted to this task.

Table 1-1. Platform-adjacent approaches and why they struggle to build platforms

Approach	Focus	Why they struggle to build platforms
Infrastructure	Robust operation of underlying infrastructure	Little focus on abstracting infrastructure to simplify applications, particularly across multiple infrastructure components
DevTools	Developer productivity up to production delivery	Little focus on solving developer productivity challenges related to systems in production running on complex infrastructure
DevOps	Application delivery to production	Little focus on ensuring their automation/tools help the widest possible audience
SRE	System reliability	Little focus on systemic issues other than reliability, often delivering impact through organizational practices instead of developing better systems

Individuals from each of these backgrounds might assert that they personally want to build more platforms rather than glue, but their organization won't let them. We empathize; we are not describing individuals, but rather how these approaches have evolved within organizations and how organizations typically define the respective teams' missions. However, the problem remains—individuals' roles are limited by the mission of their team, and changing a team's mission is not easy when the greater organization expects it to just do what it always has done.

Platform engineering asks each of these groups of engineers to come out of their silos and work in teams with a broader mission to create platforms that provide balance. This involves:

- *For infrastructure teams*, balancing infrastructure capabilities with developer-centered simplicity
- *For DevTools teams*, balancing development experience with production support experience
- *For DevOps teams*, balancing optimal per-application glue with more general software to support a lot more applications
- *For SRE teams*, balancing reliability with other system attributes like feature agility, cost efficiency, security, and performance

As a deliberate reset of organizational expectations, platform engineering gives you the ability to create teams that focus on building the technologies to finally clear the swamp.

Do Platforms Support Innovation?

As you're hopefully starting to see, platforms can cure all kinds of developer pain points, make your systems faster and more secure, make your developers more productive, deal with migrations automatically, and shorten the feedback loops for getting things done. And while we recognize that it can take quite some time to achieve all of these outcomes, we believe that this is an ideal worth striving for.

But what about the other good things that a platform might do? We're engineers, after all, so it's natural to expect our platforms to also support innovation and experimentation, because we know that innovation is the growth engine of our companies. Indeed, they can, but we want to clarify what this means, because platforms can get in the way of innovation and experimentation as much as they can support it.

If we are speaking purely of business innovation that can be developed within the context of the existing technology offerings, yes, platforms support that innovation. After all, by making application developers more productive, and in particular enabling them to push new features to production safely (such as through the use of feature flags and A/B testing), platforms allow them to build more faster, and thus support rapid experimentation with business ideas using the existing technology.

However, there will always be innovations that the platform by its nature does not support, and even fights against. Most significant innovation involving technology is going to require tools that don't exist yet in the company to be brought to bear on a problem. The data space is a great example, because it moves so quickly. You may have an excellent platform that supports easy access to relational databases and enables most of the engineers at the company to do their jobs well. But if a team

realizes they need a storage option with very different performance characteristics than your relational database offering in order to power a new, innovative business opportunity, they are going to leave your platform, at least partially, to build out this idea. If and when it comes to fruition, you may find that the new storage system is a good thing to pull into the platform offerings—but the innovation here is not enabled by the platform! That doesn't mean you should try to cram every new idea into the platform; rather, the best path is often to let these ideas develop independently, then merge in only those that are successful and have widespread demand.

It's tempting for platform teams to seek to quash innovation and experimentation that would take people off the platform. Much of the time, these ideas are a waste of engineering effort, driven by the "not invented here" bias that drives software engineers everywhere to build and create their own solutions to problems. But in some cases, these teams are right that they need to do something outside of the norm. If the platform team fights against all exceptions to using their offerings, or insists that they be the ones to build all new offerings that the teams might need, they not only push their systems to be too general but also risk inhibiting healthy innovation along the way.

So, yes, your platform should support easy experimentation and innovation within the bounds of the known, by making developers more productive and focused on the application layer. But you will not be the be-all, end-all support of innovation, and in fact, if you want to support innovation, you'll need to let some teams go their own way for a while to prove out new ideas. Making smart choices about when to push people toward central offerings and when to let them spin out their own alternative "shadow platforms"[9] is a key skill for platform engineering leaders, and one we will discuss more in Chapter 10.

Wrapping Up

We're on a complexity collision course, and many of us are already hitting the wall. Whether it's with the challenge of making DevOps effective, dealing with a million snowflake decisions, managing the increasing complexity of infrastructure as code, or simply dealing with the required upgrades and migrations that come with all software products, we need help. This is the reason that we believe platform engineering is becoming more and more important for the industry. By combining a product mindset with software and systems engineering expertise, you can build platforms that give you the leverage to manage this complexity for your company.

9 This is the platform equivalent of "shadow IT"—systems deployed by departments other than the central department, to fill gaps or bypass limitations and restrictions that have been imposed by central systems.

CHAPTER 12
Your Platforms Are Trusted

> *Trust is like the air we breathe—when it's present, nobody really notices; when it's absent, everybody notices.*
>
> —Warren Buffett

After the internal focus on building alignment within your team and its products, the next area of success is external: earning the trust of everyone else. You may ask, why should we put trust (a feeling or belief) ahead of results? Surely if you deliver platforms that manage complexity and so deliver leverage to the organization's application teams, that is a more important signal than a second-order signal of trust?

When you get your platform team to a point where they are delivering value continuously, trust will follow. However, you have platforms in production today. Features and improvements for these platforms take time to deliver, and their delivery requires customer trust, in the form of patience and partnerships for testing, validating, and adoption. Without trust, a single unfortunate event can render your carefully crafted product roadmaps useless, forcing you to scramble with throwaway work to manage the crisis.

We have seen platforms lose trust in three main ways:

Operations
: Not demonstrating operational ability at the scale customers need

Big investment buy-in
: Not seeking buy-in on large investments before starting, under the assumption that no one outside the platform team should care

Being a bottleneck
: Becoming a bottleneck to business initiatives, and so reducing rather than creating business leverage.

In this chapter, we'll discuss how you can avoid losing trust in each of these areas.

A Success Red Herring: Thinking Trust in a Leader Is Trust in the Platform

One of the worst management mistakes in any type of engineering is when the team leader oversteps their role as facilitator for collaborative decisions and instead acts as a benevolent dictator, personally making all the calls—be they management, engineering, or product decisions. It might seem easier to trust a single person who understands how product, stakeholder, engineering, and management decisions intersect, who can short-circuit conflicts by dictating solutions, and who is most accountable for the team's success. However, this approach undermines trust in the long term by failing to foster trust in any other members of the team.

It's true that the benevolent dictator setup can be efficient, especially when the leader is a strong communicator and decision maker with a small team and few users and stakeholders to wrangle. In these scenarios, the leader can use 1:1 meetings with all of these individuals to understand the nuts and bolts around any conflict, personally commit to the needed action, and provide directions to their team, thereby avoiding lengthy documents detailing trade-offs and contentious deliberation meetings.

The problem with this situation is that what makes it efficient also makes it brittle. It only works because you have one person with deep expertise in the platform who (right now) has the time to have regular conversations with all types of users and stakeholders to maintain their personal trust. Once the number of customers becomes too big for one person to handle, or that person moves on (often burned out from trying to work too many hours a week), you are left with a situation where not only do you no longer have a decision maker, but you also don't have any trust.

Now the platform team will have to start from scratch, building up inherently slower group mechanisms around trust and decision making. It can take months, if not years, for the product teams to figure out how to negotiate decisions and establish trust with one another and their stakeholders. With hindsight, it becomes clear that it would have been far more efficient for the decision maker to delegate and share some of that responsibility earlier, to build trust within the team.

Does this mean you should never allow someone to take on the benevolent dictatorial role? No. In fact, a lot of the agility at the scrappy and scalable stages comes from having someone with a pioneer or settler mindset in this sort of leadership role, making fast decisions with a small team and a small number of customers. However, this approach is not sustainable and does not scale to larger team sizes or customer numbers. If you are such a leader, you need to challenge yourself to start delegating. This will be hard on you and the stakeholders, because it will slow down the decision-making process in the short term. But it's worth it, because you'll be building your stakeholders' trust in the whole team for the long term.

Trust in How You Operate

You're probably thinking this is going to be a rehash of Chapter 6: put in some practices around on-call and support rotations, backed by some SLOs, change management, and operational reviews, and you deserve trust, right? As usual, things are not so simple.

We believe all these practices are essential to ensure rigor and hold platform leadership accountable. However, you can tick all the boxes and still lack the operational trust of many senior engineers in application teams. Before they migrate to your platform, this lack of trust may show up as standoffs, prolonged timelines, and demands for vague "proofs of concept." Even after adoption, it can cause issues—for instance, when some operational hiccup leads customers to pressure leadership into letting them build a "simpler" shadow platform that suits their needs better.

We get it. Earlier in our careers *we were* those senior application engineers who didn't want to adopt shiny new platforms, or who inherited an application on a shaky general-purpose platform that we wanted to change out for something simpler. Beyond the seeming lack of control over our fate, the most frustrating part of being on the application engineering side of things was when the platform team seemed to underestimate how much impact we felt due to their critical failures. It was a double trust problem: "Not only are they bad at operating things, they have no idea they are bad at operating things!"

The root of the challenge is that you only get good at operating foundational systems at scale by operating foundational systems at scale. When Ian was at Amazon, this was such a recurring problem that they developed a saying: "There is no compression algorithm for experience. You can't learn certain lessons without going through the curve." Where does this put you as a leader of a platform team, if you see that your senior users don't really trust your team operationally? You still have two levers:

1. Accelerate the curve by hiring and empowering leaders with operational experience at scale.
2. Optimize the curve by ordering new use cases based on tolerance for operational risk.

Let's look at both, with examples from our backgrounds.

Accelerate Trust by Empowering Experienced Leaders

When Camille started her first head of platform engineering job, much of the team she inherited was struggling with operational stability. The engineering team and the systems had been built in a scrappy fashion, and they'd grown a lot over the prior few years without much investment in system improvements, let alone rearchitectures.

The team had some new managers who had come from operational roles at bigger companies, and they were confident they knew what was needed to stabilize the systems, but they needed help—cover to focus on this work in light of customer demands for features, and help inspiring their teams of mostly software engineers to see the value of doing it.

Looking back, Camille considers herself quite lucky. There was a problem, yes, but all of the ingredients to solve the problem were already there: talented, experienced managers; strong engineers; support from the CTO. With the hiring side taken care of, Camille's contribution was providing empowerment—taking the trust problem seriously, setting a broad cultural mandate about how they would approach operations as an organization, and communicating this in a way that both the team as a whole and the stakeholders/customers would understand.

To do this, she used what is now a key tool in her management toolkit—the operational excellence OKR. OKRs were a well-established practice at the company, but historically they'd always been focused on new capabilities. Camille established an objective of improving operational stability, and got each of her leaders to commit to measurable key results their teams would deliver against this objective. She then shared this broadly to all of engineering as part of their OKR town hall, to her organization in detail in the team all-hands, and even to the executive management team (her peers and so major stakeholders) in quarterly reports.

Creating measurable goals enabled managers to explain to their stakeholders why they were focusing on operational stability work instead of features, and what they could expect to see from the work. Calling this out as an organization-wide focus area made the team take the work more seriously, and as time went on Camille assigned ownership of this objective to key up-and-coming leaders in the organization, which gave them the chance to lead cross-organization initiatives. Tracking this OKR also provided evidence for the impact of operational initiatives. This evidence was useful for explaining the value of the work during promotion conversations as well, which had in the past only looked at new feature delivery as evidence of promotability.

The work delivered meaningful outcomes. Customer satisfaction surveys for these systems showed measurable improvements. The on-call burden for the systems became more manageable, which improved the happiness of the engineering teams. And the conversations at a senior level moved away from blaming the platforms for their constant operational failures and toward more collegial discussions about new opportunities and features.

Optimize Growth in Trust by Ordering Use Cases

Part of gaining trust means waiting to push adoption until you're confident that the systems can support the application's business needs. This reminds us of another lesson from our time working together, when many of the compute and storage

platforms were new, and their teams wanted to drive adoption to prove their value. But the teams hadn't done enough performance testing to understand the actual (rather than theoretical) performance SLOs of their platforms. The result was that when an application would try to migrate, the system would struggle to meet its performance needs, causing latency problems and occasional brownouts. Even when this was done as a controlled proof-of-concept trial, the failures fed into a lack of trust, giving the impression that the platforms weren't ready, and the platform engineers could not understand operational demands like application engineers did.

In this case, there was also an opportunity that was being ignored. Many of the potential use cases for the platforms were supporting internal users in their day-to-day workflows. These workflows were important, but could tolerate some amount of latency and downtime. With the new attention to stability, Camille's leaders used the lens of performance sensitivity to think about whether they were moving the right use cases onto their offerings. They started to evaluate based not just on the platform features but also on how confident they were that the platform could meet the customers' operational and performance demands.

This changed the way they thought about their roadmaps and features. Instead of thinking of an offering as done the minute they got one customer successfully onboarded, the teams took a staged approach, starting by onboarding less critical applications. These applications provided data they could use for performance tuning and ironing out other bugs. Once they had these improvements in hand, they used them to gain the trust of the next tranche of more critical use cases, and so on.

There are no shortcuts to scaling up a team's operational ability, but by empowering the right leaders who put trust ahead of adoption, you will move faster up the curve.

Trust in Your Big Investments

Big investments, whether in a new platform or a major rearchitecture, require an enormous amount of faith ahead of demonstrated value. Not only do they take a long time to fully deliver (usually years), but they pull developers away from delivering faster value on the current platforms. As a result, customers waiting on the results of this big investment are prone to criticize the motivation behind the project. They may accuse the platform team of prioritizing "resume-driven development," putting fancy new technologies ahead of more mundane work that they believe would provide more immediate business value. Engineers love to grumble about each other, and you can't avoid all such feedback. Success means that your key stakeholders understand and trust the rationale behind the investment.

If you skip this and initiate the work by saying "Trust me, this is important; it's my team, and I'm responsible," you are headed for trouble. When users come to you with pain points and you respond that you can't address their needs because

the platform is undergoing a rearchitecture, you can expect that they will start to complain upward. If you haven't already gotten buy-in, these user complaints will result in pressing questions from senior stakeholders about your strategy: Why are we funding this work when the current users aren't getting what they need? Why are they using technology X? Who signed off on this? Unless the new project is going perfectly (and when does that happen?), you can find your entire roadmap flipped over. To avoid this, you need to get the trust up front.

Seek Technical Stakeholder Buy-in for Trust of Rearchitectures

When rearchitecting, it's critical to spend time explaining to stakeholders what you're doing and why *before* you start the work. This is why in Chapter 8 we suggest a formal decision-making process to guide these investments, which generates a record that shows not just the justifications for the decision to fund the project, but that your teams are held to high standards for such justifications.

While management stakeholders might be satisfied with evidence that you have gone through a strict vetting of these investments, senior ICs (staff engineers and the like) will want to see more, particularly around technical decisions. That is why even if your company doesn't have a standard "design review" or RFC process, you should still produce a yearly project proposal, similar to what we discussed in Chapter 7. In the spirit of Amazon's "Have Backbone; Disagree and Commit" leadership principle, if you don't let senior engineers in customer teams give their feedback before you start, you should not expect them to "shut up and commit" when you later push for their teams' adoption.

Seek Executive Sponsorship for Trust of New Products

When proposing a new product, you have an opportunity to get more than baseline technical and investment buy-in. This is a chance to get executive sponsorship from more senior stakeholders who can bring a bigger-picture perspective of what might be most meaningful to the business. Platform leaders often get focused on the technical goals: can we scale, can we operate, can we reduce costs, and so on. They can end up focusing on these goals in a vacuum, forgetting that platforms are expensive to build and have a high opportunity cost for the business; after all, these engineers could be building other things for the company. Furthermore, sometimes platform engineers (and leaders) confuse the platform itself with the outcomes they're trying to drive—the existence of a new platform isn't an outcome.

Bringing in other leaders to hear about what matters to their areas of the business can help you avoid blind spots in the platform design. It's easy to assume that everyone cares about costs, or performance, or 24x7 availability, but often when you dig in you realize that the real problem is not what you thought. They can also provide guidance as to whether you're aligned to their technology strategy; you might think

that a core application or architecture pattern is critical, while they are planning to cut that investment in favor of another area of business growth.

Maintain Old Systems to Retain Trust

Even with stakeholder buy-in, big investments are high-risk activities. Executive sponsorship lasts only so long; if you're working on a 12-month or longer project, you want to get out of the mindset that legacy improvements are pointless throwaway work, because your users won't see the new system for a long time yet. We don't just mean doing basic KTLO work here. You need to keep investing in system improvements until load on the old system is significantly falling. Further, as we discussed in Chapter 10, sometimes you need to add new features as well, either to accommodate urgent business needs or just to mollify your customers and their stakeholders.

No matter how confident you are in the big investments, others will have reasonable doubts; if you don't give some ground to maintaining their old systems in the meantime, you will lose their trust.

Gaining Trust Requires Flexibility on What Is "Right"

In the following example, we return to the time when Ian was working as a compute platform leader. One of his teams was in the process of digging themselves out of a lot of operational instability. While they had put the right leadership in place to improve the operational practices, there was still mistrust from important stakeholders—especially one of the most business-critical teams, which we'll call Icicle.

The Icicle team had a workload that was very sensitive to performance latency, and they had historically solved that problem by running their workload only on highly customized bare-metal servers. The problem with this was that these servers had low utilization and high cost. Their own business leadership wanted to improve their cost efficiency, but they trusted the judgment of their engineering team over that of the platform team. And the Icicle engineers saw that the current platform's approach to reducing costs (oversubscription of the servers) was causing unpredictable latency problems, which they were not willing to tolerate.

By treating this as a technical problem, the two teams had reached a stalemate over what was the "right" next step. The compute team wanted the Icicle engineers to provide "hard SLOs" that would allow the compute team to design and test a solution. The Icicle team wanted the compute engineers to build an extensive "stress test engine" to prove that their platform would perform under real-world conditions. The result of the stalemate was low trust, to the point that the Icicle engineers proposed to staff up their own shadow platform team to meet their special needs.

To resolve this stalemate, Ian and his leadership team changed not only their roadmap but also their product strategy. They put together a new offering that ripped

out all of their platform's oversubscription features. Yes, the new offering was more expensive than the older offering, but it was still a substantial improvement over the bare metal the Icicle team was using.

Even with this concession, the Icicle engineering team was unconvinced it would be operated to their standards. As a result, the platform team first shipped it for data science users, delivering improved performance to a highly visible business group and building confidence in the system design through this effort. Only after six months of demonstrated operational success did they earn enough trust to get the Icicle engineering team to commit to moving to the platform.

By being flexible in finding a solution that would meet the needs of both technical and business stakeholders, and showing that they were committed to operating that solution to high standards, the team was able to move past a stalemate that was fundamentally about a lack of trust.

Trust to Prioritize Delivery

Finally, you need trust that your platform will not slow business delivery. It doesn't matter how much you manage complexity or make developers productive in the longer term; platforms that are a bottleneck to delivering business value clearly have questionable leverage in that moment. Even when the initiative is something understood to be difficult, like standing up your platforms on a new cloud vendor, people outside of the platform team tend to underestimate the complexity of the work. As the bottleneck drags on, they lose trust and start questioning every aspect of the platform team's decision making, sometimes even questioning the utility of the platform entirely.

In this section we bring together three activities that are critical to avoiding these bottlenecks: velocity of delivery, prioritization, and challenging assumptions of product scope.

Create a Culture of Velocity

When they hear complaints that the platform team is a bottleneck, it is common for stakeholders and your executives to blame a lack of planning. And it is true that if you have not done any of the planning work we discuss throughout this book, they may have a point. If you're prioritizing big rearchitectures or building new platforms when the business demands are not being met by your existing platforms, you may very well have a planning and prioritization problem to solve.

But in the face of an agile and dynamic business, it's a mistake to think that planning solves all trust issues. Circling back to Chapter 1, there is a reason Agile won over waterfall—there is enough uncertainty about the business value of most features that it is far higher-leverage to build something fast, get feedback, and make it better.

If that's a two-week iteration cycle for application teams, momentum is absolutely destroyed by a platform team saying "this needs to wait until next quarter's OKRs." That is why planning—by your team or by your customer teams—cannot be the only solution to delivery bottlenecks. Not only do you waste their time seeking clarity of value that the business cannot provide, but you also create a culture that insists that the business not providing perfect roadmap requirements is a fault, as opposed to a fact of life.

When Ian led a platform organization that often found itself crucial to an application organization's dynamic new feature needs, he shaped its culture to uphold the value of velocity: balancing the throughput that came with planning via a responsive, agile approach to unplanned application demands. There were two goals to this:

- To stress to his team that it was not acceptable to resist a new application team ask just because it wasn't in their earlier plans
- To remind his stakeholders that not telling his team early about their needs would result in higher costs, because the plan would have to change to accommodate the new work

Prioritize Projects to Free Up Team Capacity

We asked Diego Quiroga, a senior platform engineering leader, to describe his experience of turning around a team that was on the brink of becoming a bottleneck to the business. What follows is from Diego.

PLATFORM PERSPECTIVES

As I stepped into the engineering leader role for our platform team, I began to learn about our domain and look for interesting problems to solve. One team reporting to me was a small platform team with the critical responsibility of managing an array of foundational services powering an enterprise social network. Application teams relied on this platform's diverse capabilities to craft customer-facing features.

The platform team remained integral to the development of these features. During each quarterly planning cycle, application teams would articulate their requests for new capabilities or enhancements of existing ones. Due to the team's limited bandwidth, several of these asks inevitably fell below the line, and the backlog continued to grow.

While the organization valued our efforts, there was mounting concern over the team's ability to manage the growing backlog, especially as a delay in certain features would jeopardize the organization's objectives for business growth. Previously, in an attempt to tackle this dynamic above all others, the team had compromised their investment in operations, leading to not just an increased on-call burden, but also negative perceptions regarding the platform's operational stability.

With fixed headcount as a constraint, I worked with the team to understand the nature of our customers' requests, seeking patterns that might reveal opportunities for efficiencies elsewhere. Looking back over a year's worth of data, we identified several recurring requests, such as configuration changes across a complex chain of services required to establish new feeds. This presented an opportunity to package these requests in a self-service manner that would minimize future workload for the team.

In the face of being an active bottleneck, making the case for an investment in team efficiency presented challenges. Balancing the immediate value of tangible features against a promise of increased team throughput was a tough sell. To maintain the trust of leadership that allowed us to sustain the effort, it was critical for us to demonstrate the impact of the investments with clear visuals and metrics.

Following the completion of the project, we introduced this new "self-serve" capability to our portfolio. Customer requests that had previously demanded the undivided attention of a platform engineer for an entire month now required only a few consulting sessions, allowing us extra capacity to dedicate to other pieces of work. In a scenario where a long-term roadmap from the application teams was not available, relying on past trends for investment decisions was a calculated bet. This one paid off well and set a precedent for seeking out similar opportunities in subsequent planning cycles.

We adopted a similar strategy to enhance our team's response time to support requests. Application teams regularly sought the team's assistance for code reviews, guidance on utilizing platform capabilities, and resolving operational issues. As the engineering organization expanded, the team fielded an average of 30 weekly support requests. Despite having a dedicated engineer to manage triage, the pressure mounted for quicker responses, especially toward the end of each quarter. To address this, we allocated capacity to develop new troubleshooting dashboards and self-diagnosis tools, offloading noncritical workloads to the application teams. Implementing "canned responses" that directed users to documentation also was effective in reducing the volume of such requests.

With the bottlenecks addressed, the team's projects evolved into more interesting, impactful, and high-leverage work, increasing the engagement of the engineers. Customers benefited too—with the newfound surplus capacity, we were able to consistently address performance and reliability issues and establish a reputation for operational excellence. Overall, while the analysis was not cheap and the projects somewhat of a risk in returning value, they were absolutely critical in transitioning us from being a bottleneck for the business to being its trusted foundation instead.

Challenge Assumptions About Product Scope

Prioritizing delivery for platforms is a constant give and take; ideally, we build broadly useful offerings that meet the majority of our customers' needs, and we aren't under constant pressure to quickly add features as part of the critical path for an application team. But there are some cases where the key characteristics of the platform give it a scope that leads to inherent bottlenecks—namely:

1. The platform is trying to expose a large surface area of functionality.
2. The platform is trying to support a diverse set of applications.
3. The platform is developed in such a way that it cannot trust its own users to unblock themselves.

A classic example of this occurs at companies that put responsibility for all public cloud adoption on a centralized cloud enablement team, and charge them with ensuring not only that developers can get access to the cloud offerings they need quickly, but also that there is significant security vetting of the offerings and the way teams can use them. This case almost perfectly hits the three characteristics we just described:

1. The surface area of the public cloud offerings that developers might want to use is huge and, worse, changes pretty quickly.
2. Unless the company is very small, the team is going to be supporting a wide set of applications and developers who probably have many divergent opinions about which of the public cloud offerings they want to use for their applications.
3. The best way to resolve this would be to let application teams resolve it for themselves, but they can't trust these developers with the superuser access to do whatever they want for security reasons, so they are stuck trying to prioritize and negotiate what will be enabled, with new requests coming in all the time.

To reduce the bottleneck in situations like this, we have had some success with diminishing the scope, by supporting fewer application types and providing more curated, higher-level product offerings. Instead of unlocking cloud primitives for everyone, we built platforms that orchestrate compute and storage for major use cases, providing a smaller but focused surface area that allowed the platform team to make the right choices about the underlying management of the cloud infrastructure. This may seem like introducing an unnecessary middleman, but it allowed for a platform that integrated core company concepts around identity management and security, handled the complexity of the cloud on behalf of users, and drove major leverage for the company.

Even this solution was imperfect; for cases where teams wanted access to cloud products outside of those wrapped in the platform, there was still a bottleneck to

evaluate, secure, and enable the new product. But limiting that to edge cases by building rich platforms for the common 80% meant that the team was able to clear these bottlenecks more quickly.

Where you are building platforms to enable your users without granting them too much trust, it's important to think about the platform features with an eye toward how you will manage or avoid these bottlenecks:

- Have you considered limiting the scope by supporting only certain types of applications?
- Have you iterated and identified the right abstraction that will support customers without exposing such a large surface area?
- Have you designed a system where users can contribute to unblocking themselves by limiting the control points that might require a security/compliance review?
- Have you included extensibility mechanisms for some platform features to be augmented by your users themselves?

You may need all four of these approaches to solve your biggest challenges: a limited scope with good abstractions to cover the common cases, and better practices for extensibility and user-driven contributions for the edges.

Tying It Together: The Case of the Overcoupled Platform

We'll call this story from our past "The Case of the Overcoupled Platform." The problems started with a two-year push to build platforms that were "batteries included." This was in reaction to a prior generation of platforms whose benevolent dictators were all thinking in silos as they built their individual platforms, and so nothing worked together. "Batteries included" was used to convey the vision of a heavily aligned approach to product strategy going forward. The vision painted was much less about platforms being "products" and more about platforms enabling "workflows," so customers would be able to use the platforms without having to build things themselves, and without being troubled by the operations of what was underneath. In many ways, this sounds like an ideal "glueless" platform—isn't this type of end-to-end focus the reason people love Apple products?

So what was the problem with applying it to these internal platforms? Unfortunately, the high bar of "batteries included" meant that the platform teams needed to design the end-to-end workflow impact for every use case before they could start writing code. This was hard enough that the initial offerings took shortcuts, and as new feature sets developed over time everything became deeply coupled. This deep coupling made rearchitectures especially hard, leading to the platform teams deciding they needed to rewrite everything into v2s that would deliver architectural improvements

with feature innovations. Of course, these ambitious scopes caused massive delays and increasing customer frustration.

These platform teams were caught in their own swamp, offering end-to-end workflows that were always not quite done, not quite ready, not quite reliable, and not quite what the customers wanted. This meant that while "batteries included" had been a great trust unifier in the early days of close collaboration and progress, as the platform organization's delivery ground to a halt due to the coupling, and the solution was seemingly only to build v2s, trust had very much dissipated; more than one stakeholder gave the feedback that "your organization builds new platforms for the sake of building new platforms."

To correct for this, Camille needed to switch up the approach, and she knew she had to address not just the cultural aspect but the product side as well. Thus, she set forward an OKR objective of "building blocks, not batteries included" (as mentioned in Chapter 11). This metaphor was based on the following three concepts:

Treat building blocks as foundational.
: In their efforts to quickly create the first version of the batteries-included workflows, the team had integrated different platforms at the component level, as opposed to using well-defined APIs. As workflow features grew, these components were often not operationally stable—poorly defined interfaces led to systems that were difficult to change, difficult to test, and difficult to monitor. To address this, the team paused on some of the workflow features to make sure the building blocks of those workflows were solid.

Blocks are composable.
: Component-level coupling was not just a problem for stability, it was a problem for improving the "batteries included" workflows that crossed platforms. That coupling meant changes in one platform sometimes had unexpected side effects in other parts of the workflow, greatly slowing down feature delivery. Worse, fixating on platform use as workflows rather than abstractions completely precluded advanced customers unblocking themselves by building their own workflows. The building blocks approach recognized that, even while the team would still provide end-to-end wrappers for common workflows, they also needed individual platform abstractions to isolate side effects. This allowed platform teams to debug and manage their systems more easily, and it allowed trusted customers to "pierce" the workflow abstraction[1] and unblock themselves.

1 See Will Larson's article "Providing Pierceable Abstractions" for a larger writeup of this idea.

Blocks can be switched out incrementally.

As we mentioned in Chapter 11, the whole platform organization went through an alignment process that picked out some big initiatives to pause so that the team could focus on the most important ones and still have time to deliver solutions for more immediate demands. It was not enough to promise that a big v2 would revolutionize the platform and solve all the users' problems eventually; proposals were now evaluated based on (1) whether they could be delivered incrementally as rearchitectures, (2) their migration costs, and (3) executive support for the potential business value.

This approach meant that, in some places, the team backtracked on usability in order to stabilize, decouple, and remove bottlenecks. Camille built senior stakeholder support through incremental unblocking, including our earlier example of Ian's team delivering a better offering for the high-performance users. Going back to the Apple analogy, the platform offerings became a lot more like early Android devices—not as polished, but allowing a lot more options. You can argue your preference there, but we are confident that for internal engineering platforms the value of stability and future flexibility for platform customers cannot be sacrificed in the name of ideal usability.

Wrapping Up

Trust takes much longer to build than it does to destroy. Many events outside of your control can erode trust: black swan operational issues, major business changes that you can't keep up with, team turnover that leaves you unable to execute despite thorough planning. Knowing that these risks are ever-present, leaders must work hard to shore up trust through everything they do.

We see this as one of the most common ways that platform leaders fail their companies. Through their own hubris, they believe that they know better, they don't bother to communicate with adequate transparency, and they trust their teams to the exclusion of listening to their customers and stakeholders. When you accept that success in the job requires building and maintaining trust, you take the steps necessary to deliver trustworthy platforms that can keep up with business demands.

CHAPTER 13
Your Platforms Manage Complexity

> *We must design for the way people behave, not for how we would wish them to behave.*
>
> —Donald A. Norman, *Living with Complexity*

We started this book by describing the "why" behind platform engineering. What is the problem to be solved? The rapid increase of complexity in technology is slowing application engineering teams down, and the business is getting less value per developer over time. Why do we need platform engineering? Because it takes a holistic approach to the problem of complexity, allowing a team of software and systems experts to reduce the drag of complexity on the application teams.

This does not mean that platforms remove all complexity. Platforms generate leverage by effectively *managing* complexity, not eliminating it; as a leader of a platform team you must become very comfortable addressing complexity in everything you do.

In this chapter, we'll highlight four areas where complexity needs to be managed to ensure you are on a successful path:

- *Accidental complexity*, where attempts by a platform to address complexity in fact just move the problem somewhere else, often creating new work for humans
- *Shadow platforms*, which are a delicate game of letting application organizations be agile, without ending up with a complex outcome of many similar shadow platforms
- *Uncontrolled growth*, where the only way a platform organization manages complexity is under an assumption it can hire new engineers tomorrow to deal with the tech debt created today

- *Product discovery*, or understanding that, for some problems, it will take iterative attempts at delivery to discover the product that actually reduces complexity for both the customer and the platform team

As this chapter's introductory quote points out, we must be realistic about human behavior when designing solutions. A technology approach is necessary but not sufficient; leverage comes when we combine technology with an understanding of human and organizational dynamics to tackle all aspects of complexity.

A Success Red Herring: The Single Pane of Glass

The idea of bundling everything into a "single pane of glass" is a popular concept in tech UX these days. Many open source and vendor tools promise to give you a single UI to control your whole system, manage your whole development experience, or streamline all of your communication. Reducing cognitive load for users by providing one UI for everything seems like a smart way to remove unnecessary complexity, and many teams invest heavily in building unified UIs, betting that they will solve their UX problems. These initiatives usually start off strong and deliver value for some common use cases, but in our experience, this early success does not sustain itself over time.

Camille saw this firsthand when a DevEx platform team she managed decided that developers had too many different places to go to find information about their work: one for code reviews, one for build progress, one for tickets, one for code search, and, of course, their chosen editor for writing code and the command line for various other activities. To address this complexity, the team brought some of these activities together into a single pane of glass web UI to improve flow and protect people from context switching.

This seemed like a good idea at first, but over time the team realized that in order to keep everyone in their in-house interface, they would have to re-create all of the workflows from each of the underlying vendor tools they were using.[1] These vendors in turn were themselves each trying to become the single pane of glass by providing hooks and integrations with one another. Over time, the team realized that their "single pane of glass" was becoming either an extra stop between the developer and the UI they needed to get to or a worse version of the real thing.

They also hit another complication that's common when building tools and platforms for developers: the developers didn't want to use the interface. Developers are picky about the way they work. Many want nothing to do with UIs, and prefer to do as much as possible on the command line; others want everything integrated with their IDE; still others want ChatOps integrations (but only when they're on-call).

1 This is reminiscent of the challenges in wrapping vendor/OSS APIs for internal use, only arguably worse, as you not only have to keep up with API changes but also must keep up with UI/UX changes.

The platform's single pane of glass would only work for one persona; not only do different humans have different personas, but the same human may operate with the same tools as different personas depending on the role (support engineer, software developer, project manager) they are playing that day. The team ultimately realized that it was better to rely on the integrations available in GitHub, Slack, Jira, etc., and integrate their platform into these common offerings to account for the different personas and their demands.

As this example shows, the single pane of glass concept is often best generalized to something else. While we might build these experiences for certain scenarios, the goal is not so much the pane of glass itself but the ergonomics of a setup where everything a user needs is within reach. Recognizing that your systems will need to accommodate different personas using different tools at different times, the basic building block of your ergonomic environment is the API and the corresponding data model. By starting not with the single pane of glass but with the APIs that will power that interface, you leave yourself room to develop different experiences depending on the persona. You can create an easy UI for basic use cases, but allow developers who prefer the command line to integrate there (and rely on the existing UIs for your tools where possible as these are likely to be better than anything you can provide).

UIs are inherently complex and hard to build right, so if your goal is to reduce complexity, we recommend you start by ensuring your products have accessible, documented, coherent API access. Follow REST standards for your APIs, as closely as is possible. Name things consistently; do one thing per call and don't require stateful sequences of calls to do one thing; plan for backward-consistency and try not to change your APIs too often once they are released. From here, you can explore integrations with command-line tools, chatbot-type interfaces, IDE support, and yes, web interfaces. It's still hard to reduce the complexity of your API layer, but if you neglect that, it's unlikely a UI will solve the problem.

Managing the Accidental Complexity of Human Coordination

A key measure of success in managing complexity is evaluating how much glue application teams still need to build to work with your platforms. As introduced in Chapter 1, "glue" refers to the code, automation, configuration, and tools that these teams build to hold things together. Glue is a response to the complexity of managing the underlying systems, and platforms should aim to create abstractions that eliminate the need for each application team to build their own. Less overall glue is a sign that you have reduced the infrastructure complexity for application teams.

There is another type of glue that we didn't talk about in Chapter 1: human glue. This is the "glue work" that Tanya Reilly so eloquently describes in her talk and blog post "Being Glue": all of the manual workarounds, documentation, and coordination

needed to resolve gaps between the things a team needs to do and what they are actually doing. In a quest to limit the technical glue from Chapter 1, some platform teams end up creating a new "accidental" complexity by over-relying on human glue.

Imagine a platform that hasn't bothered with operational tooling for its application teams. It's like trying to drive an old car with the hood welded shut—you don't expect most drivers to know how to fix their engine, but they still need to know where the smoke is coming from. When an incident happens and the platform hasn't provided enough diagnostic tools, the application team is stuck. Instead of handling the issues themselves, they have to get on a call with the platform team to figure out if the platform is the culprit, which is frustrating for both teams. As we covered in Chapter 6, exposing platform metrics and using synthetic monitors is key to avoiding these escalations. Complex outages will still sneak up on you, but with proper tooling, you can stop using platform engineers and DevOps/SREs as human dashboards.

If you're looking for a rule of thumb to know whether you're doing enough to manage complexity, ask yourself how often you rely on "human glue" to resolve issues. Do you rely on manual processes to coordinate open source software upgrades, or to drive fixes for common application outages? As engineers, we believe that humans should be reserved for managing the truly complex scenarios, and we should apply software to resolve the merely complicated. In the following sidebar, we give an example for the case of migrations; in general, the less you need to rely on humans to coordinate programs across your platform and the application teams that use it, the better you are at removing complexity from your users' lives.

Managing Migration Complexity

We covered one of our migration success stories in Chapter 9: in the face of a major operating system version upgrade, Camille challenged her team to do everything they could to complete the migration without human project management support. The team responded by first writing a small piece of code that tracked each host, whether it needed to be upgraded, and by whom. This was run daily to produce a report that showed how the migration was progressing. The report was then fed into Jira, which automatically created and assigned tickets with details about what needed to be done to complete the migration.

Of course, this wasn't as simple as it sounds. A key element for success was the ownership metadata registry mentioned in Chapter 12, which provided tracking of which code belonged to which team. That data was used to bootstrap the process of figuring out where to assign the tickets. The team then wrote code to apply heuristics to various identifiers associated with the system resources in order to find the most likely person to assign the tickets to, and thus minimized the churn of incorrect assignments. This turned into a useful system for tracking and maintaining ownership data more broadly, which could then power other migrations.

This changed the way most migration exercises were approached across the company, reducing the complicated human-driven processes into more predictable machine-driven ones. Instead of defaulting to project managers tracking spreadsheets and endless status update meetings, the team spent time making the tool more powerful through more nuanced dependency mappings and smart reminders, and they put more time into automating common elements of the migrations so there was less migration work for the customer teams. Over time, the biggest challenge we had with this process was that too many groups wanted to use it to drive migrations before they had thought through the details of the migration process.

We did not completely eliminate the need for TPMs, but their role evolved from doing hand-to-hand combat with each team in the migration path to acting as overseers and ambassadors. They scaled to support more migrations because there was less manual work for them to track in each migration, and they could focus on the *weird* 20%. If, for example, we saw that to complete a migration we needed to get a big customer storage system upgraded and that work required a lot of coordination with the customer team, we could deploy TPM support at this point to enable the unblocking of another automatable tranche.

The goal of all of this is to treat TPMs as the rare specialists you bring in when you can't think of any other engineering-driven tricks that could make the migrations either automatic or self-service on the part of the customer. In our experience this will still mean you need to have folks with this skill set on staff, but they should be the few that you all admire for their discipline, attention to detail, and organizational savvy.

Managing the Complexity of Shadow Platforms

We discussed shadow platforms in Chapter 10, but as a reminder, they are the duplicative platforms that application engineering teams sometimes build for themselves. In general, shadow platforms increase the overall complexity of the company's software; however, they are usually built to reduce the complexity for a particular area. Because of this local view, there will always be some amount of platform work happening outside of your platform team. The goal is not to restrict all of it, but to be aware of it.

Trying to stop every application engineering team from ever building something that could be considered part of the platform remit is a fool's errand. At scale, it's impossible. Moreover, it's simply ill-advised to try to halt all platform-related experimentation and innovation happening in teams outside of your organization. These teams, as experts on their own needs, will take a pioneering mindset to knocking down their problems. When your platform fits their needs they will probably use it, but when there's an advantage for them to build their own thing, sometimes they will do just that. In doing so, they might build the first draft of the next valuable platform offering.

To wrangle those shadow platforms, you need to build on the trust that we discussed in the previous chapter—that's what keeps you in the loop. Being informed gives you the chance to prepare for whatever comes your way, whether by embedding one of your engineers into the project, getting regular updates from the team, or even setting up expectations about what would need to happen in the future if this team decided they wanted you to take over the project.

If and when you decide to take over a shadow platform, it's important to realize that you are inevitably going to create some new complexity in the process. After all, you're aiming to make it useful beyond the scope of its original team, which usually means expanding its surface area. The trick at this point is to reduce the pioneer-driven complexity while corralling the new complexity within your platform team, rather than letting it leak out to the users.

An Example of Managing a Shadow Platform

In our experience, a successfully dealt-with shadow platform looks more like a well-managed mess than a black-and-white picture of great execution. The following story illustrates just how iterative and messy success can turn out to be.

About six months after Ian took over the second of five in-house compute platforms, the CTO started pushing for AI for data science, giving significant headcount to one of the business-facing technology executives to execute this strategy. This executive then hired a leader for the initiative who was a type that platform engineering leaders often cross swords with: a pioneering visionary eager for radical change. This new leader believed that the barrier to AI innovation was the coupling to existing flawed (in his mind) in-house platforms, which hindered data scientists from quickly adopting cutting-edge public cloud and OSS systems. He aimed to create a platform where each data scientist could have their own cloud account, using whatever IaaS infrastructure primitives and OSS they wanted, like they were at a small startup.

In handling this, Ian's first mistake was assuming that everyone could see how complex this would be, and that the pioneer would get this feedback and quickly change course. After all, most of the data scientists could barely administer their own development workstations, so individual cloud environments were going to be a mess. Furthermore, most data scientists were going to need an incredible amount of platform integration for their existing workflows to be usable, and that was the work Ian's org already had planned. It was just going to take a while to do it "right" and avoid unnecessary glue.

The pioneer, however, remained stubbornly on course and hired a team to drive this effort, planning to build shadow platforms wherever Ian's team wouldn't cooperate with his vision. He justified this by pointing out that about 10% of the data scientists came from engineering backgrounds and could handle the complexity of administering their own environments and writing their own platform glue. Since this platform was only for experimental work, which would need a rewrite to go into production, both the pioneer's team and the early users believed it would be fine to deliver a system that didn't provide the same operational foundation as the broader ones Ian's team was providing. They figured they could deal with those architectural issues later.

Once Ian realized that the original plan was going to stick, he began holding regular meetings with the pioneer leader and his team. The goal was to get serious about what it would take to increase the coverage of this platform beyond the 10% of advanced users to include everyone else. These meetings exposed the places where the pioneering team was overlooking the complexity of scaling their offering. For example, they hadn't thought about how to migrate users off of existing systems (which would take hundreds of developer years), or how to manage the administration for the nontechnical users.

However, it was clear that the need to "make progress on AI" meant the effort had backers up to and including the CTO. So, Ian had a choice: he could let the pioneer team build a complex shadow platform without his support, or he could figure out a compromise that would let him influence the work to avoid some of the inevitable complexity.

The compromise he landed on was to shake up one of his teams' roadmaps, freeing up two developers to support this initiative. This was the upside of the CTO's attention—when other stakeholders questioned the changes, Ian could tell them this was coming from the CTO's priorities. The developers he chose were settler types, and he gave them a difficult remit: "Your job is to not slow this project down, but to find the places where you can build the right 'long-term' components and use the opportunity to build those earlier than we would have."

In practice, they succeeded about halfway; keeping the project moving quickly was always going to mean creating some amount of glue that wouldn't scale. But even the half success was a positive thing, especially because as the system became real, two things happened:

1. The 10% of data scientists who had previously been blocked could now successfully access the cloud for iterative experimentation.
2. Now that they had something real to play with, some of the remaining 90% of data scientists were able to try the system, and Ian's teams' earlier concerns around the complexity of operations, administration, and integration became obvious to everyone.

This didn't immediately solve the standoff, and it took a couple of years of reconciliation before the whole thing was sunset in favor of an integrated platform. Still, in those two years the company was able to benefit from a lot of innovation that otherwise would not have been possible. This became a case study to show how the platform organization could partner with application engineering teams without slowing them down, as opposed to always being left out of the "scrappy" part of the cycle. All told, it was a successful example of partnering to create a better (and more sustainable) product, as described in Chapter 5.

Managing Complexity by Controlling Growth

Growth is addictive. When you have scaled out a platform team from its early days to a stable organization, it can be tempting to think that the only way to accomplish more is by adding more people to the mix. How else will you ever close the gaps in the product offerings? You need a full set of engineers to cover both development and the on-call rotation, a product manager, and all of the supporting apparatus that a new team might need. The only way to provide all of that is to grow and grow and grow.

The danger in this mindset is that unchecked growth contributes to the complexity that you're trying to avoid by building a platform. First, it reduces appetite for managing complicated stuff: there's a strong temptation to throw bodies at problems instead of investing in automation or rethinking the work. This then creates the kind of work that software engineers don't want to do: tedious on-call rotations where you are constantly doing manual fixes, migrations that require human follow-up across the company, and provisioning requests that take dozens of steps to complete. The longer you go without investing in automation, the more likely you are to end up in situations where automation can't do enough to manage the complexity, and you're stuck in a nonscalable staffing model just to keep up with support.

Growth also encourages complexity by removing the pressure to be smart about what you build and where you invest. When engineers can justify any pet project, they will often build without regard to what the customers need. Managers and product managers in turn realize that it's easier to build their own empire rather than getting alignment with their peers on how to solve problems. Growth gives everyone an excuse for why things aren't quite going well: another person to point the finger at, a newcomer who doesn't yet know how things work around here. In the worst case, you end up with a sprawling portfolio of half-baked ideas and products that don't quite fit together well; a complex swamp of offerings for your customers to navigate. And frankly, we're not sold on platform teams being the originators of many new initiatives. As we said in Part I, we expect most innovation to come from the application teams building what they need, not the platform team itself.

Even considering all of this, those of you who have spent your careers in growing companies might think we're off base to suggest slowing growth when you aren't being forced to do so. Why would you even pretend to do new things when you're barely keeping up with the rapid growth and scaling of the company itself? And why would we push platform teams to reduce growth when application engineering teams are just as guilty of indulging in it?

Remember, though, that there is a difference between platform teams and application engineering teams: a platform team is more likely to be seen as a "cost center" and therefore an area where efficiency is expected, rather than a revenue-generating organization. Guiding a culture of smart efficiency is part of the mandate for leaders in this space, and complexity is the enemy of efficiency. The platform engineering mandate is not driving efficiency through any means (say, outsourcing to manage the cost of manual approaches). You achieve efficiency by strategically simplifying through software engineering and product discovery.

We know that there are times when you need to grow, and entering new product areas can be one of these times. If you have run your organization so efficiently that you cannot pull from existing teams to cover a new offering, you will need to grow. And of course, there are times of scaling and company growth where expansion is sensible. But good platform leaders understand that their platforms deliver leverage, and that means they shouldn't need to grow at the same rate as the overall engineering team, once that leverage point is established.

As a guardrail, we recommend this rule of thumb: most of the new work in established areas should be funded by existing people on those teams. This forces management to sharpen their focus; if their KTLO workload has gotten out of hand, can they find ways to reduce that cost? Do they have a strong sense of the most important areas to work on, and are they divesting from adding to features that either are "good enough" or, worse, haven't shown their promised value?

Managing complexity implies not only that your platforms can support far more users than the number of developers of the platform but also that, once you have achieved baseline coverage for a product area, you do not need to linearly scale the number of platform engineers for every new thing you want to do in that space. This doesn't mean you should cut yourself to the bone and have no slack in your organization. This is why the measurement of KTLO + mandates + operational improvements is so important: knowing the minimum number of people you need to handle that workload gives you the absolute bottom of your potential team size (which is probably that number + 20% so everyone doesn't immediately quit). Above that baseline, you can exercise discretion and think about investments. Being thoughtful about the next set of work, incorporating customer demand, team demand, and your own strategic insights, and getting the most out of the team you have before you go and ask for growth is a sign of mature platform planning and leadership.

Managing Complexity Through Product Discovery

Product discovery is the work of understanding customer demands and creating, in the words of Silicon Valley Product Group, "a product solution to this problem that is usable, useful, and feasible." Product discovery is not just needed for the products you build from scratch; it is also an important exercise to go through when you are creating platforms that are based heavily on open source systems. If you want to create curated product offerings, discovery is key. Yet under constant pressure of new potential shadow platforms, many teams take customer demands literally, providing whichever open source system the customer asks for without taking the time to determine whether it is the right product solution for the overall platform.

This leads to a common predicament, where teams that have provided (or, more commonly, inherited) these open source systems are stuck with operational complexity that grows linearly with the number of users and use cases. You can reduce the coefficient of linearity with investments in automation, but from a design perspective, most major open source products have too wide a surface area exposed to the users. This is most evident in distributed OSS for data processing, such as RDBMSs (PostgreSQL, MySQL), Cassandra, MongoDB, and Kafka. These systems are highly complex by nature, and the OSS vendor model drives them to compete with one another by adding more and more features, which means they have very broad interfaces.

Some leaders immediately jump to the conclusion that they have to standardize and limit choices in order to manage this challenge. That's a great idea, but how do you do it? In our experience, while application developers on the whole may agree that fewer infrastructure choices would make their lives easier, they rarely agree on what the limited set of choices should be. If you're disciplined (and have senior leadership support), you may be able to establish standards early enough to avoid annoying your customers by taking away features they're already using and forcing migrations to reduce duplication. But this can backfire if it slows application teams down much, and it's politically unpopular. Most of the time standardization happens only when the platform team hits a breaking point, where the support burden of so many OSS offerings prompts the question of why you need so many in the first place.

There's an option in between "let a thousand flowers bloom, until you can't stand it anymore"[2] and "offer a strict platform that allows for very little variation," and, as you might have guessed, it involves your product culture. To do this, you need to take the time to understand your customers. Explore *why* teams are using their chosen tools, whether it's out of habit or for specific must-have features. Through an iterative process of product discovery, you can develop the insight needed to curate

2 See Peter Siebel's article "Let a 1,000 Flowers Bloom. Then Rip 999 of Them Out by the Roots".

your offerings, reduce complexity, and better meet your customer needs. In our final story, we will walk through just such an exercise and the many iterations that led to success.

Tying It Together: Balancing Internal and External Complexity

In this story, Ian had a team of about 10 people who owned a collection of OSS systems (PostgreSQL, Kafka, and Cassandra). The team followed a data reliability engineering (DRE) approach,[3] which meant they offered a "platform" that provided all of the support and provisioning for these systems, with the team's proactive engineering spent on automation, particularly around resilience and autoscaling.

Burning Out on OSS Operations

As we predicted in Part I, this approach to platform delivery wasn't scaling. Each OSS system had a large feature surface area, and the complexity of operating that as a company foundation led to constant operational strain. Even with two pager rotations, the number of high-severity incidents a week was much closer to 50 than 5, and this was only somewhat ameliorated by the fact that rotations could follow the sun. The DRE team had reached the limits of efficiencies that could come from automation alone.

Unfortunately, the application teams weren't seeing this pain; indeed, they were happy with the flexibility provided by the OSS systems' extensive feature set. Their primary demand was that the DRE teams should expand their portfolio with more offerings. But when the DRE leaders tried to explain that they needed twice as many engineers to sustainably manage the existing workload before they could even think of adding more offerings, they were met with disbelief. The DRE team could continue to grow only by agreeing to support more systems and configurations, which they knew would quickly arrive at the same unsustainable scaling point.

Trying (and Failing) to Change the Game

This impasse prompted the team to make several attempts at reducing the complexity they had to manage. The first attempt to change the model was trying to get out of the game altogether and move to vendor "hosted open source" IaaS implementations, where application teams would own their own operations. In another company, putting the operational load on the vendor might have solved the problem. But due to the multicloud requirements of this company, the differences across vendors made

3 See *Database Reliability Engineering* by Laine Campbell and Charity Majors (O'Reilly).

it too hard for the application teams to operate themselves. This was only truly appreciated after the vendor offerings were rolled out, when the DRE team was still constantly being paged to handle acute operational issues because the application teams lacked the depth to debug them. At this point, they realized "get out of the game" really meant "stay in the game, but as an operations team."

The next tactic was to take a page from the SRE book and try to improve things through SLA documentation. The idea was to provide clear documentation of what the team could and could not support within its SLA; this would help application teams understand that their bespoke configurations couldn't be supported by the DRE team alone and lead to a shared operational model. The DRE team saw this as a rational compromise, balancing customers' needs for customization with their ability to manage these at scale. But it sounded like a lawyerly abdication of responsibility to those customers, who saw the team as using rules and processes to position themselves as advisors and evangelists rather than owners. Ian brokered a top-down handover for one of the biggest customers, forcing trade-offs on both sides, but it was clear that this approach wasn't going to be sustainable without constant conflict and politics.

The team turned next to a software-based approach: full encapsulation. We described this in Part I: creating a service API layer that fully encapsulates the open source APIs, which allows the platform team full control over what they support/operate. The users were not particularly interested in giving up their direct access to the data stores in favor of this option, but the team hit on the idea of tying this to a feature supporting multiregion reads and writes with simple (key, value) semantics. Early customers were satisfied, but as the platform team looked for the next tranche of potential customers, they realized that no one else was interested in the multiregion use cases in the short term, and instead they wanted a full SQL interface. So the team started creating a plan for adding secondary indices, with a view to eventually supporting SQL semantics. At this point Ian stepped in, noting that the idea of building their own in-house global SQL database was less thinking big and more of an impossible dream.

Shadow Platforms Force a Reset

During the time these three attempts were being made, frustrated application teams had started building shadow platforms to get the features they needed, such as MongoDB for document support and FoundationDB for transactional write semantics that could horizontally scale. These had the usual characteristics: when the application team was growing rapidly they were happy to own all the operations, but as the initial engineers moved on and the operational load kept growing, the application teams were eager to offload them onto the platform team. Adding more OSS to the portfolio was the opposite direction of where the team wanted to go, but there was

something here that the company needed, and Ian challenged his team to figure out how to meet those needs.

At this point, the team did a reset by bringing in some managers who had experience building product-oriented infrastructure. These managers started by doing product discovery, looking across the collection of offerings and, using their prior experience to guide them, investigating what the application teams actually needed from these systems. That is, they attempted to discover what the teams specifically *required*, as opposed to just preferred, in Cassandra, MongoDB, PostgreSQL, Kafka, and FoundationDB. With this understanding, the platform team sought to identify a narrower surface area of common needs that could be used to remove one or two of the broader offerings. Through this effort, they found two major opportunities:

Simplification
: There was a lot of demand for a cross-application configuration platform for which (key, value) semantics were fine. Here, the team leveraged the shadow platform investment in FoundationDB to power a managed service focused only on solving this use case.

Coupling multiple primitives
: There was still a demand for something schema-aware, but product discovery revealed that this requirement was as much about caching and search as it was about ACID transactions. The team realized if they combined PostgreSQL with searchability and caching functionality, they could offer a more limited SQL system that would satisfy most customers.

They also realized that if they could succeed on these two projects, then both a platform (Cassandra) and a shadow platform (MongoDB) could be sunset.

Executing on the Reset

To create these new platforms, the team collaborated closely with application teams that had pressing demands for the offerings. The result was rapid development of platforms that, while scrappy, satisfied these customers and turned them into advocates for the work throughout the company. Thanks to the early successes, a long list of application teams signed up to onboard as the platforms matured.

In all, it took about four years of iteration to identify the right product offering that would meet the major application needs while limiting the complexity for the platform team, meaning their growth could be controlled without ceding all future feature development to shadow platforms.[4] You may be wondering whether a process that involved three false starts and a multiyear migration is really a success. But this

4 We would be remiss not to be honest about the cost of this change: at the time of writing, the team is in the second year of a five-year plan to fully deprecate the MongoDB and Cassandra offerings.

is the reality of delivering platforms that manage your company's unique complexity: they will evolve, trade-offs are hard, and sometimes the best solution isn't even viable at the time you identify the problem. Don't be afraid to keep iterating.

Wrapping Up

Alignment and trust are challenging but achievable goals for your platform team. Managing complexity, on the other hand, is a North Star that you'll use to guide your organization, but it's unlikely you will ever fully accomplish this task. There are things you can do to detect complexity, and practices you can use to help control it, but complexity will always be there. That doesn't mean you should throw your hands up and give up on the task entirely. When you find complexity in excessive human coordination or shadow platforms, see it as an opportunity to develop new ways to automate, simplify, and understand the needs of your customers. As your platform organization develops and matures, use this North Star to remind yourself that too much growth too fast can make it that much harder to keep complexity in check, and that the iterative process of product discovery is critical to finding the simplest scalable solution among the sea of more complex options. The more time you are able to spend thinking about and driving down complexity for your users, the more mature your platform will become.

About the Authors

Camille Fournier is a technology executive with leadership experience ranging from early-stage startups to Fortune 50 corporations. She was a founding member of the CNCF Technical Oversight Committee and currently serves on the board of *ACM Queue*. She has published two other books with O'Reilly, *The Manager's Path: A Guide for Tech Leaders Navigating Growth and Change* and *97 Things Every Engineering Manager Should Know*.

Ian Nowland has been in the software industry for 25 years, most recently spending 4 years at Datadog, where he was the SVP of core engineering. Prior to this, he was at AWS in their early days (2008–2016), where he was the lead engineer on the launch of Amazon EMR and the leader of the first five years of the EC2 Nitro project. He is currently a cofounder at a stealth mode startup.

Colophon

The animal on the cover of *Platform Engineering* is a marbled newt (*Triturus marmoratus*), a striking amphibian native to Western Europe, specifically France and the Iberian peninsula. Its name is derived from its distinctive appearance: its dark brown or black body is adorned with irregular, marbled patterns of green. This coloration provides excellent camouflage in the newt's natural habitat of woodlands and meadows. Female marbled newts also have an orange stripe running along their back. While females are larger, these newts range from about 5 to 6.5 inches long.

The marbled newt's diet consists mainly of insects, worms, and other small invertebrates, which it hunts both on land and in water. Though this animal is primarily terrestrial, as an amphibian it requires access to water for breeding and tends to remain in ponds during colder times of year. During breeding season each February, males develop a striking, feathery crest on their back and attract mates by spreading pheromones through a tail-lashing motion. Female newts deposit eggs individually, first smelling and inspecting the leaves of aquatic plants before selecting a leaf to wrap around the egg. Scientists have determined that marbled newts rely on celestial clues such as geomagnetic fields and constellations to locate familiar breeding ponds.

The marbled newt is listed as vulnerable by the IUCN due to habitat loss. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Jose Marzan, based on a black-and-white engraving from *Lydekker's Natural History*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro and the heading font is Adobe Myriad Condensed.