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