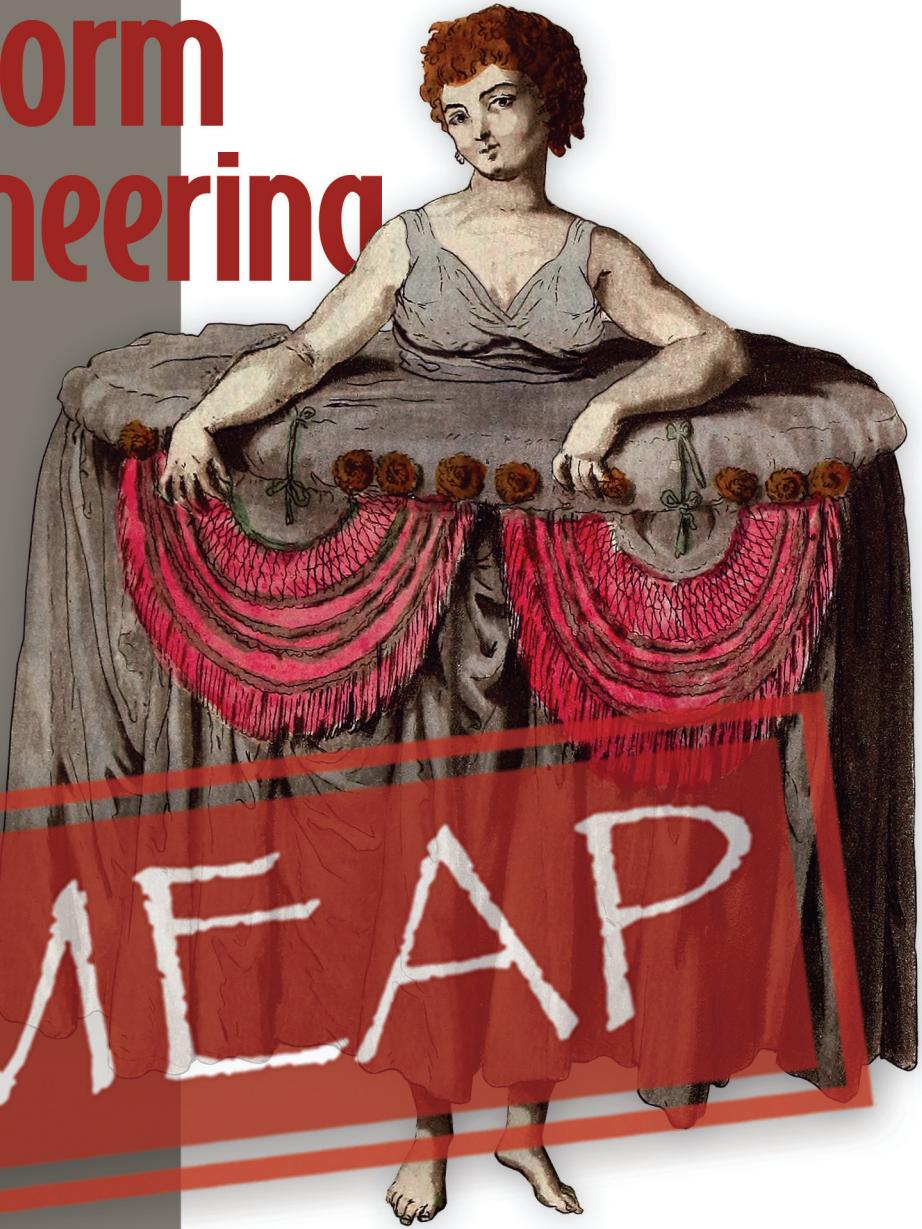


Sponsored by



# Effective Platform Engineering

Ajay Chankramath  
Nic Cheneweth  
Bryan Oliver  
Sean Alvarez



MANNING



# The leading observability platform that puts you in control

In this cloud native world, there's too much observability data to know what is most valuable, or how to control skyrocketing costs and quickly remediate issues.

Chronosphere is the observability platform built for control in a modern, containerized world.

We empower customers to focus on the data and insights that matter, to reduce data complexity, optimize costs, and remediate issues faster. Our observability platform **reduces data volumes and associated costs by 60% on average**, while saving developers thousands of hours.

Chronosphere was built from the ground up for cloud native environments by expert engineers, for your engineers.

## How Our Observability Platform Works



Want to learn more?



This book is in MEAP - Manning Early Access Program

---

## What is MEAP?

A book can take a year or more to write, so how do you learn that hot new technology today? The answer is MEAP, the Manning Early Access Program. In MEAP, you read a book chapter-by-chapter while it's being written and get the final eBook as soon as it's finished. In MEAP, you get the book before it's finished, and we commit to sending it to you immediately when it is published. The content you get is not finished and will evolve, sometimes dramatically, before it is good enough for us to publish. But you get the chapter drafts when you need the information. And you get a chance to let the author know what you think, which can really help us both make a better book.

MEAP offers several benefits over the traditional "wait to read" model.

- Get started now. You can read early versions of the chapters before the book is finished.
- Regular updates. We'll let you know when updates are available.
- Contribute to the writing process. Your feedback in the [liveBook Discussion Forum](#) makes the book better.

To learn more about MEAP, visit <https://www.manning.com/meap-program>.





**MEAP Edition**  
**Manning Early Access Program**

**Effective Platform Engineering**  
**Version 5**

**Copyright 2025 Manning Publications**

For more information on this and other Manning titles go to [manning.com](https://manning.com).

# welcome

---

Thank you for purchasing the MEAP edition of *Effective Platform Engineering*!

We're thrilled to have you on this journey as we explore the world of building and optimizing developer-centric platforms. To fully benefit from the book, you will need experience in software development, DevOps practices, and cloud-native technologies like Kubernetes. If you're familiar with continuous integration, delivery, and observability, you're in the perfect place to take your skills to the next level.

In *Effective Platform Engineering*, we create platforms that empower developers and reduce operational complexity. You'll learn how to build self-service platforms that enable teams to focus on delivering business value without getting bogged down by day-to-day operations. We share our collective experience of transforming organizations with real-world strategies that can drive developer autonomy, improve efficiency, and accelerate digital transformation. From understanding foundational platform principles to incorporating advanced features like Generative AI, this book will guide you every step of the way.

We've also made sure to weave practical exercises and real-life scenarios throughout the book, giving you hands-on experience in solving platform engineers' daily challenges. You will encounter case studies based on real organizations demonstrating how platform engineering can be a game-changer for businesses. These exercises will take you through scenarios such as optimizing cloud infrastructure, reducing cognitive load for developers, and aligning platform capabilities with business goals. This approach ensures that by the end of the book, you'll understand the theory and have practical tools and techniques to apply in your work.

We truly believe that platform engineering is more than just a technical discipline—it's a way to unlock creativity and collaboration across your teams. Our goal is to provide you with actionable insights and techniques to help you design platforms that are not only powerful but also sustainable and easy to use. We hope this book becomes a practical companion for you, equipping you with the knowledge and skills to build and refine platforms with confidence.

We're excited to hear your thoughts and ideas as you progress through the book. Participation in the [liveBook Discussion forum](#) is welcomed and crucial for the final product's success. Your feedback and questions will help make this book even better. Together, we can create a vibrant community of platform engineers sharing experiences and learning from one another.

—Ajay Chankramath, Bryan Oliver, Nic Cheneweth, Sean Alvarez

# *brief contents*

---

## **PART 1: GETTING STARTED**

- 1 What is Platform Engineering?*
- 2 Foundational Platform Engineering Practices*
- 3 Measuring your way to Platform Engineering Success*

## **PART 2: BUILDING ENGINEERING PLATFORMS**

- 4 Governance, Compliance and Trust*
- 5 Evolutionary Observability*
- 6 Building a Software-Defined Engineering Platform*
- 7 Platform Control Plane Foundations*
- 8 Control Plane Services and Extensions*

## **PART 3: SCALING ENGINEERING PLATFORMS**

- 9 Architecture Changes to Support Scale*
  - 10 Platform Product Evolution*
  - 11 Generative AI in Platform Engineering*
- Appendix A. Developer Expectations*

# **1 What is Platform Engineering?**

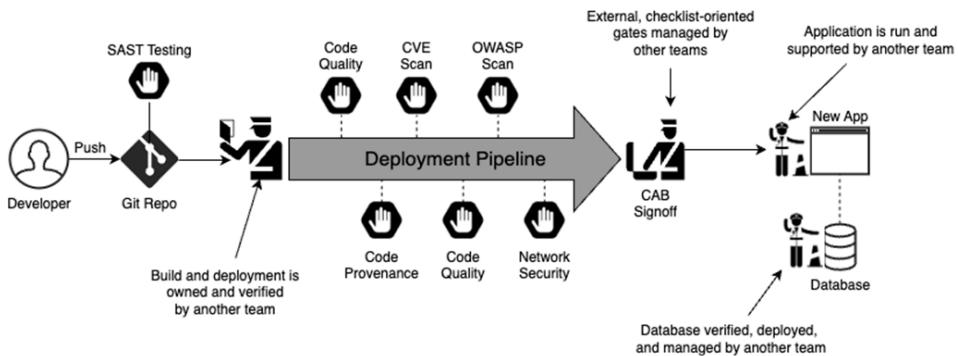
## **This chapter covers**

- Definition and outcomes of platform engineering (PE)
- Why should organizations build and use platform engineering?
- Mental models and core principles of platform engineering
- How platform engineering differs from DevOps, SRE and Developer Experience (DevEx)?
- Impact of GenAI on platform engineering

Throughout this book we will imagine working for a company called PETech that is facing problems with inefficient practices deploying and operating software to production, which will be similar to situations we have seen in many real companies. PETech's problems are significant, and platform engineering practices will be shown to dramatically improve operations. The problems at your company may be similar, and by following the journey of PETech, hopefully you will be able to see how using platform engineering practices and implementing an engineering platform can improve things!

## The Background of PETech

PETech is a retail company that has been in business for a long time, and the business was initially built on a monolithic application. As more and more teams have spun up in the last few years, the company has moved to a microservice architecture with teams owning what they build. As more and more services are added the development teams are starting to see problems with releasing and supporting software efficiently. When new features are requested, it can take weeks to deploy any changes once development is complete. In figure 1.1, we show the manual inefficient deployment process at PETech which may not be too unlike many similar organizations.



**Figure 1.1 The PETech deployment process is very manual and requires multiple handoffs to different teams. There are also many different verification steps that result in a release process that can take weeks to complete once a code change is pushed. The developers have to understand and account for all of these changes because any failure will result in a rejected deployment being pushed back.**

With any change, there are multiple handoffs and stages because different teams are responsible for deploying and verifying any code change before it is allowed into production including security, compliance and governance tests. Engineers are drowning in the complexities of these controls because they know any change that fails a test will result in a rejected deployment, adding even more time to release. Once those features are released, because running the software is owned by another team any bugs and fixes take quite a while to resolve, requiring time consuming back and forth communication.

When we talk to software engineers both in situations like PETech, or in situations that have started to improve from this but still have room to grow we ask them: what if you could build and deploy a new service in an hour? What if you could build software rapidly and allow for the same degree of control and governance required by the enterprise? What if it were easy to test a change with only a portion of the users before rolling it out to everyone? If an issue occurs in production, what if the developers knew about it immediately, or even before it occurred and could fix it quickly? In short, what if we could make creating, deploying, and running new software changes a *non-event* in your workflow, speeding up delivery and making development *fun* again?

By implementing the principles and practices of Platform Engineering at clients ranging from scaling startups to large enterprises, we have repeatedly seen that this can be achieved for engineering teams of all skill sets and sizes. This book aims to empower readers to embrace and implement platform engineering practices within their own organizations by following PETech's transformative journey, achieving rapid and reliable software delivery, streamlined deployment processes, and the enhanced ability to minimize and recover from production issues. This approach ultimately leads to customer delight, success in your organization's mission, and engineers doing what they love to do, focusing on writing software.

## 1.1 What is Platform Engineering?

Platform engineering (PE) is an exciting evolution of DevOps culture and practices that is revolutionizing the ability of many organizations to quickly and reliably deliver software to production. Though there is an internal cultural aspect to DevOps, far too often it just becomes a Team. Product developers tell DevOps what they need and DevOps creates it; Frequently with a complex combination of manual and automated infrastructure-as-code practices. Over time and at scale, the general pace and complexity of IT work ends up not sustainably improved. Improvements in quality are often used to justify the effort. While the boost in quality is valuable, it's important to remember that DevOps originally emerged to keep up with the faster development cycles that the market demands. This remains a challenge for most organizations.

If we start thinking of DevOps as a product team that builds and delivers an infrastructure product, and we design and engineer with this goal in mind, we open the door to a completely different outcome.

it allows teams to operate in a more autonomous way during the development process leading to faster delivery of features. As teams start to operate their own software in production, PE practices provide a way to do so with less cognitive load and complexity leading to software stability and quick issue resolution.

To begin our journey, let's imagine an end state for PETech: Setting up for success was like opening a portal to a realm of efficiency. It took half a year to roll out a change in the previous state, facing hurdles at every step. However, with the changes we implemented, it defied the odds, slashing the timeline from months to a mere hour. The bottlenecks that once plagued the process – security concerns, auditing nightmares, compliance headaches, change control dilemmas, and spiraling costs – are now gracefully overcome. But how was PETech able to accomplish this feat? The secret lies in dismantling siloed functions and optimizing the elements that traditionally bog down the deployment process. Security, deployment, and other critical governance aspects are streamlined and automated, allowing for an efficient workflow that catapults the company into a new era of technological prowess.

As we embark on this learning journey about platform engineering, it's crucial to understand the principles underpinning this radical transformation. From defining efficient ways to develop, test, deploy, and monitor the code to addressing security and governance concerns with a finesse that incorporates modern software techniques, PETech has implemented a comprehensive and exhaustive list of principles that make deployment efficiency a goal and a reality.

The essence of platform engineering is intentionally addressing the problems like PETech was seeing and delivering the outcomes of engineering efficiency. Still, deployment efficiency is merely the tip of the iceberg. Platform engineering, as showcased by PETech, is about unlocking possibilities, breaking barriers, and reshaping the future of technology to create more efficient and effective ways to build and deliver software products. Together, we will redefine the landscape of platform engineering.

This book shows how to effectively start using platform engineering practices in your organization to build and deliver engineering platforms that will enable:

- Fast onboarding of new teams and services
- Increased deployment frequency to production
- A more remarkable ability to minimize and quickly recover from production issues.

In this chapter, you will learn:

- What platform engineering is?
- When using these practices will provide the most significant benefit?
- What the core platform engineering principles are?
- How best to start using platform engineering principles to deliver an effective engineering platform?

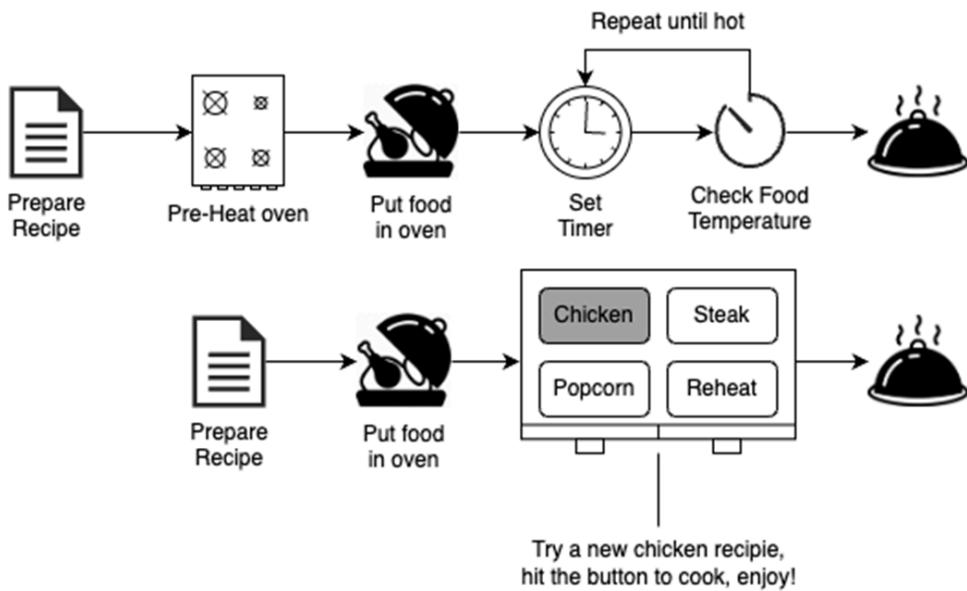
The number of articles, posts, and conference tracks on platform engineering has increased dramatically over the last few years. Yet, these sources can have very different ideas about what it means, why it matters, or what *good* looks like. Why is it so hard to define?

Platform engineering seems hard to define because it is a strategy targeted at a central challenge in modern software development: How do I rapidly and sustainably deliver software experiences to customers, given the constantly evolving technologies, the extensive governance and operational requirements, and the critically essential security challenges? Platform engineering is an approach to solving this challenge.

We define *effective platform engineering* as a craft:

- Composed of the architectural, engineering, and product delivery disciplines
- Applied by dedicated, development teams with broad domain knowledge and product ownership
- Delivered as an engineering platform that provides internal software development teams access to the tools and technologies they need to innovate, create, release, and operate their software
- Providing self-service and seamless access to all platform functions
- Minimizing the need for non-development tasks, and cross-team engineering delays
- Decreasing the cognitive load required to meet all security, governance and compliance
- Succeeding as measured by clear business goals that can be regularly reported against with observable metrics.

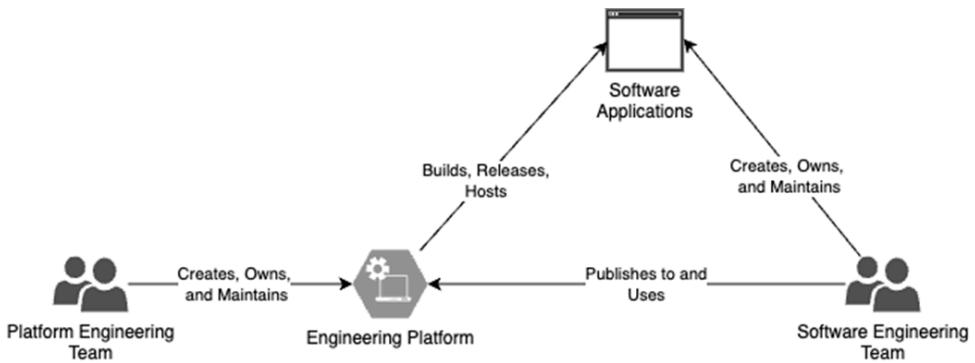
Think of this as analogous to an oven with pre-programmed buttons for different types of food. In figure 1.2, we show an analogous workflow with preparing food.



**Figure 1.2 An oven with pre-programmed food choices would make experimenting with new recipes much easier. You wouldn't have to worry about knowing the right temperature or how long to cook a new meal. You could simply put the ingredients together, hit a button and enjoy! Building an engineering platform should provide developers a similar experience for deploying new software.**

If a company designed, architected and manufactured an oven that could cook any recipe with the push of a button (a microwave maybe?) it would be much easier to experiment with new recipes. The company would need to staff chefs that have enough experience to know what temperature and time is needed to cook many different types of food perfectly. This way without having to worry about these things you could simply put together ingredients, hit the right button and enjoy your creation! However, even with these push-button options you may want to cook a recipe that uses ingredients that weren't accounted for when the oven was made, like seafood. In this case you will need to know at what temperature and how long to cook the food, and check it periodically to make sure it's reached the right temperature but isn't overcooked. That way you will know it is hot and free of anything that could cause illness. With every new recipe you need to repeat this process as well. you can still use the manual processes, it'll probably just take longer. So how can we use platform engineering to make something analogous to this magical oven? That's where engineering platforms come into play.

As described in the definition above, a modern *engineering platform*(EP) is the product delivered by a platform engineering (PE) team to enable development teams to operate more efficiently. An EP can provide teams with a ‘paved road’ to deploy and operate software from build to production release that will remove cognitive complexity and get started from day 1 of a new effort releasing to production. Figure 1.3 shows the relationship between a platform engineering team and the consumer software product development teams.



**Figure 1.3 The relationship between a Platform Engineering team and the Engineering teams it serves is through an Engineering Platform. The PE Team provides a platform that defines the tools and processes needed to be secure and compliant, and teams use it to publish and operate their software.**

An EP is a collection of the technologies, tools, and knowledge needed by the majority of teams within the organization to deliver their software quickly to production, safely experiment with the features and capabilities the software provides to customers, and operate it over the long term, all while remaining secure and compliant. This could include CI/CD systems, pre-defined build and release steps that include compliance and security best practices, reusable infrastructure definitions that can be easily applied, a hardened runtime environment such as kubernetes, and many more capabilities. The components of the EP should be defined and implemented as part of a single, unified product roadmap that is prioritized according to the needs of its users, engineers. The goal is that when using the EP, teams should be able to continue to deliver value to the organization without many of the common engineering friction points and delays usually experienced.

Most of the time, if you ask a stakeholder what is responsible for that engineering friction and delay, they will point to those operational, security, and governance *requirements* as being the unavoidable cause. Systems must indeed be healthy and secure, protecting customer and business data. Laws and regulations must be followed so that a business can be in compliance and maintain customer trust. So, how do you engineer and deliver a platform product that removes the friction yet still meets the requirements? That is the professional goal of platform engineering. Platform engineering skills are how an effective engineering platform is delivered.

## 1.2 Why should I care about Platform Engineering?

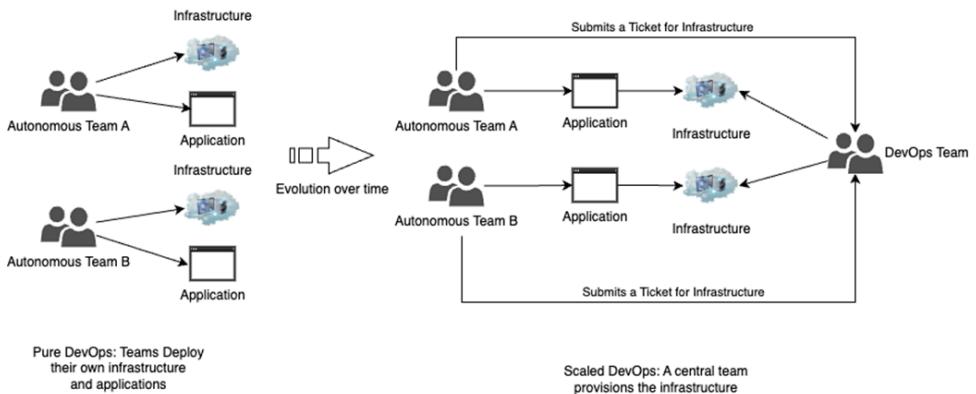
This section highlights the challenges and limitations that can arise with traditional DevOps practices and explains why shifting to platform engineering can provide significant benefits. By traditional DevOps we mean: a development approach that emphasizes collaboration and communication between development (Dev) and operations (Ops) teams, and that values automating the configuration of infrastructure. This is not a bad value statement. But better collaboration and communication has always been a visible goal (or challenge). What changed was the increased emphasis on using software to manage infrastructure. And as with software development in general, it is the way in which those communication and collaboration boundaries function that will define the issues and possible solutions.

Understanding these issues and solutions is important because it demonstrates how platform engineering can enhance efficiency, standardize processes, and improve overall software delivery by providing self-service tools and automated guardrails for development teams.

DevOps, as a practice, has revolutionized software delivery by giving teams the cross functional skill sets needed to manage the creation and delivery of their software end-to-end. However, we are now seeing issues with the practice that require a new way of thinking, and you may be seeing some of these issues in your organizations.

- As DevOps is adopted, the increased autonomy to deploy can lead to significantly increased costs as more cloud resources are provisioned for each team's software lifecycle environments.
- Required skill sets in cloud technologies, infrastructure as code, and networking requirements (among others) that can be hard to find and staff.
- The growth in complexity and the difficulty in finding the right skills results in DevOps Teams beginning to appear, responsible for ensuring infrastructure is provisioned when requested (in a compliant way), accounts are set up, permissions for access are set appropriately, and so on, switch requests queueing up much the same as traditional IT.
- Whether called an Ops, DevOps, or Platform team, quickly this team can become overwhelmed and unable to keep up with the volume of tickets sent in, resulting in lengthy fulfillment times that slow down onboarding, innovation, and, ultimately, releases to production.

As DevOps teams expand in companies, another common issue arises: technology becomes messy. Figure 1.4 demonstrates how the DevOps teams are evolving over time.



**Figure 1.4** In a pure DevOps culture teams will deploy their own infrastructure, but this leads to sprawl in the IT estate. To combat this, many organizations will create central DevOps teams in an effort to enforce standardization. This leads to bottlenecks at scale, when the team becomes overwhelmed with tickets.

Each team can choose how they set up their software (especially in the cloud), which leads to a need for more standardization. Some teams use serverless tech, others prefer Kubernetes, and some mix things up. This makes it hard to share knowledge across the company and slows the start of new teams because they have to decide on their tech setup from scratch. Solutions like architecture review boards or central architecture teams can help, but in the midst of sprawl the effect is still more delay. Security and compliance rules become significant concerns as teams become more independent and release software more frequently. To mitigate this, security teams make lists to ensure everyone is careful when setting up their software. This leads to new processes like Change Advisory Boards or security reviews, creating delays.

One way to tackle these issues is to task the very open-ended DevOps *cultural* goals with the very specific *product experience* goals of platform engineering (See Figure 1.3). This experience is a measure both of the time required, and equally importantly, the actions required. An effective *product* does require interactions with the builder of the product in order to use. This means creating self-service tools that allow teams to handle problems instead of waiting for help. By giving teams access to standard patterns, tools and autonomous ways to use them, we can start dealing with the mess of tech problems that have built up over time. This also lets us focus more on making things better for developers so they can work on what matters most: building great products. In a very real way, platform engineering is another way of saying product engineering. It's not enough to ask, are we engineering good infrastructure implementations, but rather, are we creating a good consumer experience, a good product experience for the developer using our infrastructure platform?

By providing an efficient path to production and allowing teams to start a new service without needing to make infrastructure architecture, security and governance decisions, they can get started while being able to deploy to production quickly, even on day one. As an engineering team grows and requires capabilities for internal team management, the ability to deploy and manage new infrastructure, and add additional capabilities for continuous integration and delivery (CI/CD), providing self-service APIs can make these functions available for immediate use with no waiting time for another team to execute. With platform engineering, automated guardrails can be provided as part of the paved road, with security and governance teams moving from manual compliance verification to developing automation that can do the same verification with every deployment. The organization can quickly gain confidence that all processes are followed, and teams can become more efficient by focusing only on what is needed to make their new features work.

For all of these reasons, more and more organizations are seeing the benefits of platform engineering practices.

### **1.3 When to use Platform Engineering principles?**

We recommend introducing platform engineering concepts and the ensuing team very early in an organization's journey. This might mean different things to different organizations.

For an established organization without explicit platform engineering thinking (product thinking), the challenges will be greater. Internal organizational structures can present barriers, the difficulty of change can be harder to predict, and time needed before experiencing the benefits is likely to be longer. For large enterprises, the time to begin platform engineering is after they first identify the most strategically valuable software development initiatives. It is often the case that Enterprises struggle to know which development efforts are returning value, which have the greatest potential for value, versus those that are merely routine maintenance. It is not unusual for a large enterprise to first create a digital division, sometimes even as a whole new company, in which to gather the strategically valuable work and invest there in PE.

For a digital native startup, organizations that started their journey using digital techniques to build the products as opposed to adopting them later on in their lifecycle, platform engineering may not be top of mind as you work on translating some critical concepts into products. However, we strongly recommend that the strategy around platform engineering is considered part of your overall business and technology strategy, even if you cannot establish a dedicated team to do the work. In organizations like this, we see this as an evolutionary path where the first step will be to build the most valuable technical capabilities as part of your domain-aligned team and then find your inflection point closely associated with your need for scaleups. For teams that are deploying and operating software in a similar way this may include reusable modules for common CI/CD pipeline tasks, reusable infrastructure definitions, or alignment on tooling to use across the delivery ecosystem.

In reality, most organizations have a standalone DevOps team that might execute software delivery functions as an activity that could be more cohesive from the developers building the product capabilities. For these organizations, platform engineering techniques will provide the maximum benefit. Chapter 3 will discuss in detail setting up the Key Performance Indicators (KPIs) first and determining the absolute need for building Engineering Platforms in a platform engineering paradigm.

## 1.4 When do these principles not apply?

As you learn about the concepts of platform engineering, we ask you to consider them as principles that can be applied in your work, whether or not you have an Engineering Platform at your organization or an initiative to create one. One may at first think that “Platform” implies large-scale or big organizations. Still, the concepts and principles of well-built engineering platforms and platform engineering practices apply to any organization or team of any size.

For example, consider that self-service principles may be less valuable to a tiny organization or team. Still, the principles of CI/CD, decoupling lifecycles, and creating repeatedly deployable environments are practices from which any organization of any size can benefit.

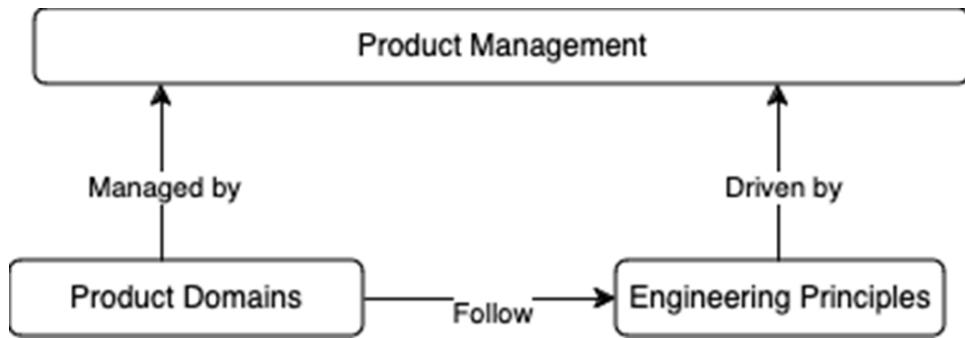
If you work at a large organization and are considering building a platform engineering practice, this book is for you. If you work at a tiny organization or startup, there are still lessons to be learned here even if the investment required in a PE team may not make sense until the company reaches a larger scale. When you're just starting a business and working on your first products, don't stress about creating an engineering platform. It's when your startup grows bigger that you'll start thinking about ways to make things more efficient and standardize common tasks thereby bringing about economies of scale. That's when you'll realize you need an engineering platform. In this book you will still find a subset of principles to adopt to help your company grow and experience less overall scaling pain as your organization and customer base increase.

## 1.5 Foundational Concepts in Platform Engineering

A platform engineering team will work to enable practices in the organization that will increase efficiency while enhancing and maintaining the practices that ensure the -ilities of software delivery such as

- Maintainability
- Security
- Scalability
- Reliability
- Extensibility
- Recoverability

Now, let us look at a high-level mental model of platform engineering in figure 1.5 and the ensuing explanation.



**Figure 1.5 below shows the high-level mental model of platform engineering. It has three essential components: product domains, underlying engineering principles, and product management.**

Platform engineering is a discipline orchestrated by the seamless collaboration of three key elements: Strong Product Management surrounding the product the platform engineers are building, an emphasis on finding and maintaining the implementation Domains within the product, and the consistent applications Platform Engineering Principles. At its core, Product Management acts as the visionary architect, shaping the purpose and trajectory of the platform. It forms the backlog of building an engineering platform and prioritizes according to customer needs. This strategic blueprint guides the specialized efforts of Product Domains, each contributing their expertise to construct a cohesive technological ecosystem. Engineering principles can include

- Identity and Access Management (IAM)
- Networking
- Security
- Compliance and Governance
- Cloud Runtimes (i.e. Kubernetes)
- Observability

It's important to understand the relationship between engineering principles and product domains. Engineering principles embody our overarching goals, such as maintaining loose coupling and enabling independent releases within and between platform teams. The specific practices implemented to achieve these outcomes, such as Domain Driven Design (DDD), act as the applied methodologies. For instance, preserving flexibility in technology and implementation choices aligns with the practice of evolutionary architecture. This interconnected relationship between principles and domains underscores the strategic alignment of overarching goals with the tactical application of specific practices to achieve them.

However, the platform engineering team itself will need a good understanding of these principles so that they can be embedded in the design and execution of anything produced. This book will describe these foundational principles and how they fit into a well-designed engineering platform.

### 1.5.1 Platform Product Management

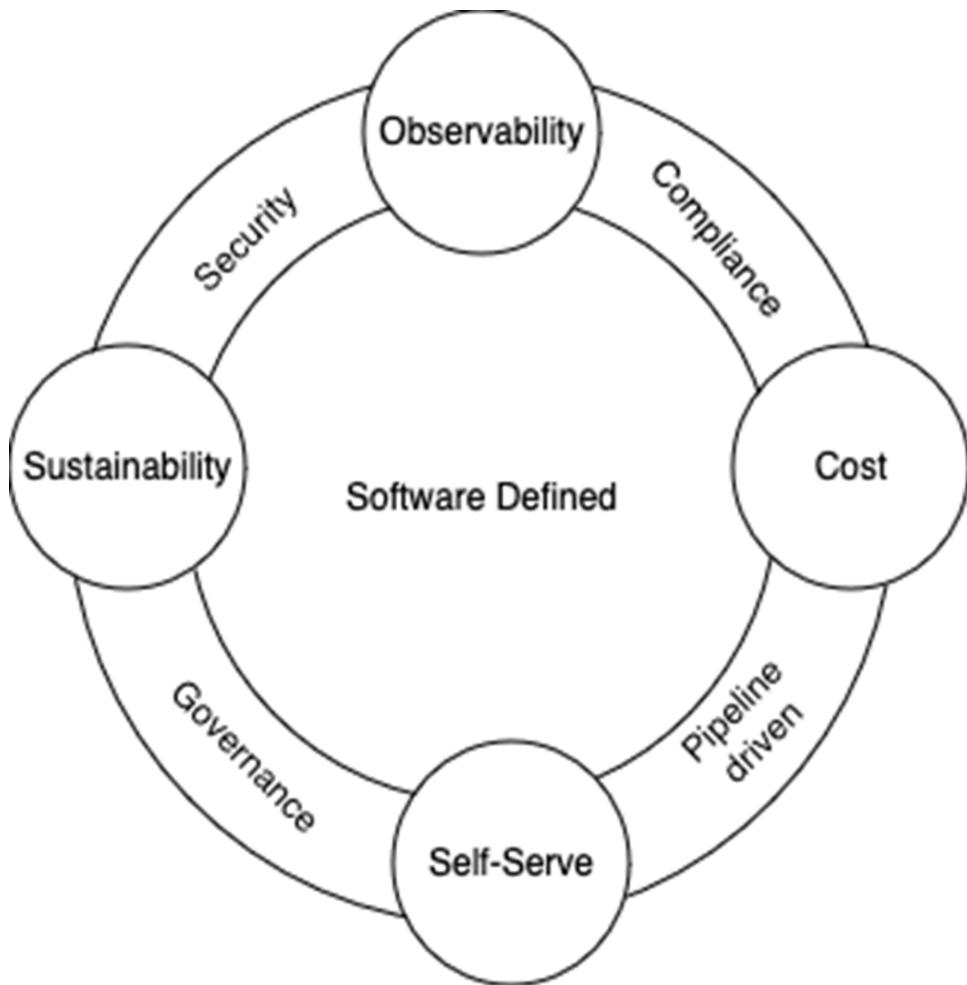
All platform engineering is done with the rigor of *platform product management* with a well-defined product lifecycle. This is typically referred to as a platform product. The product management lens ensures that the platform you are building does not turn out to be an incoherent collection of automation that is difficult to manage and scale. Platform Product capabilities built by a platform engineering team are usually used by the Software developers responsible for the functional code and practicing the DevOps culture. However, *Site Reliability Engineers* (SREs) will also use these capabilities because of the natural progression where the SREs will maintain the code and improve its reliability. By using the same practices to deliver products to external customers, delivering and managing an engineering platform as a product for internal engineering teams will help ensure that the needs of those customers are best being met, increasing adoption and, ultimately, value.

The Engineering Platform product is divided into eight *domains*. Each of the domains is built with the *six unique engineering principles*. It is important for us to understand the principles before we talk about the domains. In the next section, we will talk about these principles and then talk about the domains in the subsequent section.

### 1.5.2 Platform Engineering Principles

This section introduces the essential principles of platform engineering, which include observability, continuous deployment, self-service functionality, compliance and governance, cost and sustainability, and security. Learning about these principles is crucial for ensuring that platform engineering practices enhance the efficiency, reliability, and scalability of software delivery within an organization. Figure 1.6 shows an expanded view of the principles.

The platform engineering team is assumed to execute like any other development team within the organization. Practices such as “everything defined as code” and continuous testing will be assumed across all practices. In executing platform engineering in the true sense of software engineering, several core principles should be followed.



**Figure 1.6 Expanded View of the Principles involved in the Platform Engineering Mental Model.** As described in the previous figure, this figure shows the specific domains and principles that constitute the platform engineering mental model.

Software-defined exists in the middle because it is core to every other concern. The rest of the principles are circular however, and that is because each of these should continuously be revisited as the others evolve. For example, as you evolve self-service functionality you'll likely need to look at the observability of that not just for function but to ensure runtime costs haven't gone up too high as a result of your users being able to deploy more. That may lead to looking at governance and compliance to keep those costs down. By keeping every one of these principles in the platform team's core ways of working, the chances of success for any initiative will go up significantly.

### 1.5.3 Observability

Observability in software development is generally considered a way to measure the state of your system's internals by inferring what is visible externally. There are times in which developers and teams conflate monitoring and observability. While monitoring refers to the system's health, observability focuses on functional correctness. Similarly, while monitoring tells you if the system is operational at a given time, observability tells you whether the system could become unhealthy. While monitoring is key for any system, software, or otherwise, it inherently tells you about a failure and a suboptimal end-user experience as they would undoubtedly have encountered the issue. Observability, on the other hand, is about being proactive. It achieves that through its inherent nature of using inferences to develop insights and actions to ensure your end users do not experience the issues.

In our definition of observability for engineering platforms, we look at it far beyond typical observability for applications and infrastructure. Instead, we look at it across multiple axes such as portfolio of applications, platforms, cloud health, incidents, service health, and most importantly, business operations.

### 1.5.4 Continuous Deployment via Pipeline (CI/CD)

Continuous Integration/Continuous Delivery (CI/CD) is a set of practices and techniques in a typical software development life cycle that is used for integrating the software written by multiple developers and teams frequently and continuously with the idea of generating faster feedback loops and ensuring that the software written by individual developers work together as a product functionally.

While these terms (CI & CD) are usually referred to together, the functions refer to two parts of the software build and delivery process. CI focuses on integrating the code changes in one repository, while CD refers to delivering software to various environments with appropriate gating as defined by the development teams. **Continuous deployment**, the automated process of releasing the changes to production, is often referred to in this context.

### 1.5.5 Self-Service Functionality

Self-service functionality should be an intrinsic trait of all capabilities the platform exposes. In a small startup organization, all individuals have access to and the ability to change/modify/deploy any app or environment. In this case, decoupling engineers from systems has no value because there is no ticket system we are trying to get away from. The decoupling becomes valuable as teams and responsibilities mature, and the org evolves into submitting requests or tickets to an emerging "DevOps-like" team. Self-service **APIs** should be developed to eliminate roadblocks to usage by autonomously executing teams.

### 1.5.6 Compliance and Governance

Compliance & Governance in platform engineering is approached differently than standard enterprise. One of the goals of platform engineering is to enable teams to work autonomously so that the platform team does not become a bottleneck on their ability to deliver software. As one might expect, this goal directly opposes the goals of your compliance, security, or governance team.

Thus, when we discuss Compliance and Governance here, we will approach it from an enabling perspective, always looking for ways to align the goals of these teams (platform and compliance) and reduce friction for the platform's developers and users.

We do this by instrumenting automated capabilities in the tools and systems provided to teams and users of the platform. Teams should have complete control and administrative rights over their process and how they self-verify and govern their software. The platform is not responsible for compliance and governance while writing software. This allows the Platform team to focus on compliance verification as opposed to prescribing how compliance is done for each team. **Compliance at the Point of Change** (see Chapter 6) allows the Platform team to verify software as it enters the environment and removes that same team from the processes and procedures with which individual teams developed that same software.

### **1.5.7 Cost and Sustainability**

Cost & Sustainability in platform engineering are the principles that ensure individual developer responsibility is enabled at all possible steps within the software development lifecycle. This starts with provisioning the resources to get your application developed and being integrated and built, followed by getting the application tested and deployed. This should be a matter of time about optimizing the costs after building and deploying your applications instead of having the developer access to the right capabilities to build it right in the first place.

Cost considerations in platform engineering are applied both from the ideas around **cost optimization** (referred to as “cloud cost optimization” these days) as well as **FinOps** (which breaks down the silos and reduces friction in using the cloud resources in a cost-efficient manner to improve the ability of the business to scale), around the principles set forth by the FinOps foundation. We expect this to be applied in all the possible domains (as described in the next section) and do not see it as something stand-alone and independent of your platform engineering ecosystem.

Sustainability is a closely related principle to cost management but focuses on sustainable and environmentally responsible practices when choices are made as the developers go through crafting their products. By providing environmentally responsible sustainable options through platform capabilities, the developers can choose them as their sensible default, reducing the need to use less viable options inadvertently. All developers will make a more responsible choice when given a choice. However, this must be a choice exposed through platform capabilities as the architecture of the product they are building and the related scaling and performance requirements eventually dictate the right decisions.

### **1.5.8 Security**

Security is approached similarly to how we have outlined Compliance and Governance. Teams developing on the platform are responsible for continuously verifying their code is secure while developing and deploying it. The Platform team is then only responsible for verifying that work has been done via Compliance at the Point of Change and that the software added to the environment is secure.

The platform team applies the same principles to development teams regarding the platform's security. This means applying security checks to the Platform team's pipelines that allow for continuous security verification during the writing of new platform code and verification of that security at the boundary of the environment or deployment of the changes to the platform.

While network rules are helpful in specific situations, they need to provide a complete security strategy and reliance on them leads to friction between development and platform teams.

## 1.6 Platform Product Domains

We now outline the importance of applying domain-driven design (DDD) practices to engineering platforms, emphasizing the creation of effective boundaries to ensure low-friction integration and interaction among different domains. Understanding these principles is essential for developing a cohesive and efficient platform that can evolve independently while maintaining a high-quality user experience and operational efficiency.

Like any other software product, domain-driven design (DDD) - a software development approach that emphasizes understanding the core business domain, enabling software that closely aligns with business needs through collaboration between technical and domain experts - practices can benefit an engineering platform's delivery. What are the domains within an Engineering Platform? There is not a single, universal answer to that question. Given that the measure is in the boundary experience between the domains, it is possible that more than one way of describing the features or capabilities of the platform can result in effective, loosely coupled boundaries.

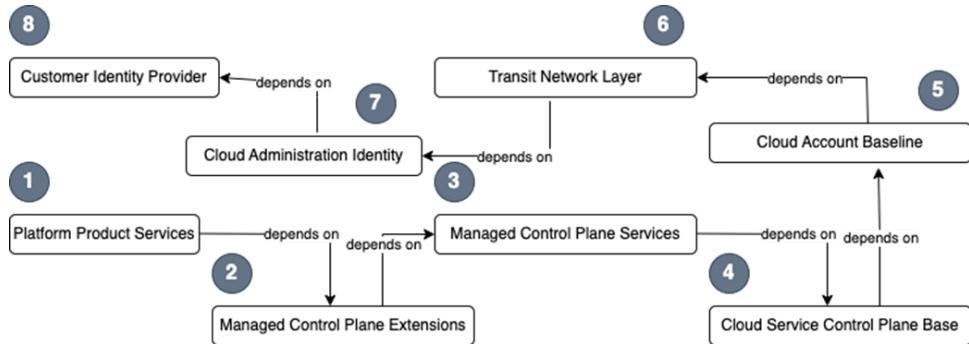
There are also many seemingly intuitive ways of setting boundaries that are anti-patterns in practice. Remember, an effective domain boundary is not merely anything you want it to be, nor is setting boundaries along technology lines (traditional functional team structure) likely to be correct as a default. The above boundaries result from years of trials (and errors) of many different definitions in both Fortune 500 companies and well-financed startup settings. While these may not be the only effective boundaries, they are highly effective. Functionality within each domain is capable of low-friction integration and interaction with the other domains. High-quality SaaS options exist across nearly all these domains that can be integrated in a healthy domain manner should you choose to use them. Self-serve, loosely coupled boundaries can be maintained where different domain teams are created among the top-level domains as well as many effective subdomains within each.

Effective low-friction boundaries between even these domains do not exist by default. They must be carefully created and maintained if the user experience is to be sustained and the investment in creating the Platform returned.

The engineering platform described in this book will be divided into eight domains, each developed using the abovementioned principles in the previous section.

When discussing platform product domains, we often refer to domains as related to the cloud. This approach reflects the standard practice in today's engineering, but it's important to note that these aspects are relevant outside cloud-based systems. We see the cloud less as a specific place and more as a standard set of principles and ways of operating in the contemporary world of software development.

Figure 1.7 shows the eight distinct domains. These domains should be visualized as dependent on each other and following a logical progression, with foundational elements, administrative identities, and control planes forming the basis for more advanced functionalities and services. Understanding these dependencies is crucial for effectively planning, implementing, and maintaining the overall system.



**Figure 1.7 Expanded View of the Domains involved in the Platform Engineering Mental Model.**

By separating our platform into these distinct domains we can provide a bounded context for each platform concern, which will allow independent codebases and deployment pipelines to develop. When we talk about the dependencies, we should recognize that an initial deployment will need to respect these. For instance, the cloud service control plane can't be deployed until a transit network is available. Once we get past that however, each of these domains can evolve independently unless there is a breaking change of some sort.

### 1.6.1 Customer Identity Provider

This domain covers identity and authorization framework implemented for the Customers of the engineering platform. This domain is kept separate from the administrative identity defined above to create a cohesive experience across all platform capabilities, regardless of the underlying provider. This strategy is similar to many other products you likely use. For instance, Netflix hosts their service on AWS. When you stream a movie or TV show on Netflix, the file that contains the movie exists on a physical device where it is assigned access permission based on the AWS IAM roles and permissions. However, as a user of Netflix you are not assigned an AWS IAM identity nor is there a direct connection between you and the role used to access the file. When you subscribe to Netflix, you are assigned an identity within the Netflix customer identity system. When you stream a show, the service validates your permission to watch the show and then accesses and streams it on your behalf. This allows Netflix to provide all of its services without concern for the end user where they are hosted, what 3rd-party services are being used, etc. We want to provide the same experience for engineering platform customers.

**NOTE** You may think of cases where a development team requires access to the underlying provider services to debug workloads or validate configurations. Part of addressing these concerns falls under the platform team principle of observability. Another way to manage this will be by teams providing their backend services and *connecting* them to the platform. Both of these practices will be discussed.

### 1.6.2 Cloud Administration Identity

The obvious foundational starting point for creating a hosted product. The engineering platform product team must be the administrative owner of the cloud accounts that host the product. As we will demonstrate later, platform customer identity is a completely separate domain. Administrative identity will cover things like:

- How cloud vendor accounts are provisioned
- How corporate SSO is integrated with the account
- How service accounts (machine users) for platform automation are supported
- Service account access policies

### 1.6.3 Cloud Account Baseline

While different cloud providers use slightly different terminology, each provides a mechanism to sandbox organizational units, teams, software lifecycle environments, and workloads. This mechanism can be called accounts, subscriptions, projects, etc.; here, we will refer to this capability as ‘accounts’. This domain will ensure an account strategy that will enable security between deployment environments while minimizing the ability for development teams to unintentionally deploy in-progress code to the wrong environment or inadvertently cause corruption between environments during runtime. Account baseline configuration implements requirements that are organization or account-wide in nature, applying in every setting and in a sense, neither blocking nor enabling with regard to the account purpose. Examples include:

- Network flow and audit log aggregation
- SIEM or SOAR automation
- Account-level security scanning
- DNS hosted zones and zone delegations when using the vendor DNS service
- Organization or product level network-as-policy configuration

### 1.6.4 Transit Network Layer

All deployed platform services and capabilities will require network connections for communication, the setup and management of which are handled in a dedicated domain. Networking can encompass many modes of communication, including:

- Access to and from the public internet
- Inter-service (i.e., serverless functions to databases)
- Traffic across availability zones with a region

- Traffic across global regions
- Traffic between cloud providers
- Traffic between a cloud provider and a private cloud/data center

By handling this in a dedicated domain, we can ensure a strategy that covers all use cases optimally and minimizes conflicts that could surface between communication modes.

**NOTE** This layer also introduces a shared responsibility model that will be required through all the following domains. This enables the platform team to provide services without defining how the engineering teams will use them other than enforcing guardrails for usage. In the case of networking, this could mean that the platform provides a mechanism for teams to expose their workload to the public internet in a secure and compliant way, but teams are required to configure this and monitor for issues.

### 1.6.5 Cloud Service Control Plane Base

This domain defines the platform's core control plane that enables and orchestrates all infrastructure components of the platform. Kubernetes provides a highly effective and extensible control plane API in addition to the compute orchestration engineering teams the functionality to deploy and run their workloads. In this book, we will provide an example that uses Kubernetes for this layer, which has quickly become an industry-standard way to deliver these services efficiently. Each of the main cloud service providers offer fully managed Kubernetes control planes with a shared security model wherein they assume full responsibility for the secure configuration of the control plan at industry standard benchmark levels covering 99% of corporate and government users requirements all for a negligible cost. These implementations are guaranteed upstream compatible Kubernetes and are by far the most effective approach in a cloud context. This domain is specifically labeled as Cloud and Base to indicate that the scope should be limited to only those elements that are fully cloud vendor managed in the operational sense. Later domains include a Managed label but managed in this case means managed by the engineering platform team rather than the cloud vendor.

Today, most enterprise engineering platforms rely on distributed service architectures, and therefore Kubernetes will be available to act as the platform's control plane. However, Kubernetes isn't the only way to build a control plane. If your organization isn't using Kubernetes, you'll need to set up a control plane through other means. Cloud vendors API is a good starting point. But unlike Kubernetes, those APIs don't come with built-in extensibility, which means you'll need to create the service interface yourself. This can be a heavy lift, so it's worth considering Kubernetes to manage infrastructure resources, even if your users don't specifically need Kubernetes.

Cloud vendors are continually enhancing services, like AWS Control Tower, making it easier to create a control plane when extended with serverless functions. Alternatively, you can explore solutions like HashiCorp's Terraform Cloud. When combined with Sentinel and well-defined account permission boundaries, these tools can help you achieve similar results.

### 1.6.6 Managed Control Plane Services

The control plane will have several services that the platform team *manages* that are either used by the platform team for management purposes or are utilized broadly by platform customers. These are not custom services created by a Platform team but rather open-source or other third-party services commonly integrated with the control plane, which may include direct or indirect interaction with user applications. We make a distinction between control plane *services* and control plane *extensions*. As you will see in the discussion below, there may be capabilities that blur the distinction. Yet, there is operational and domain value in maintaining the basic distinction between control plane services and control plane extensions.

Control plane services have the following characteristics:

1. They collect, analyze, and return or forward data about the state and activities of the cluster or applications running on the cluster.
2. They do not provision or configure other resources inside or outside the cluster except where a non-customer-facing activity.
3. They can include cloud vendor-specific implementations of the Kubernetes API.

An example of number two is the cluster-autoscaler. True, this service provisions and removes nodes based on deployment requirements managed by the Kubernetes orchestrator. Yet, while Users of the cluster benefit from dynamic capacity management, they do not directly interact with the capability.

Assume number three means storage class and gateway API services that, though generally available, are not quite fully incorporated in the vendor's delivery process for Managed Kubernetes services. For example, there was a period where the AWS efs-csi was technically available as a customer-managed option, even though it was being developed to be a fully managed EKS add-on at some point. While it was customer-managed, treating it as a control plane service was more effective than an extension since it was soon to move to the cloud control plane base domain.

There are many examples in this category, a few of which are listed below to provide some idea of the scope. The next section defines extensions and will further discuss the value of the distinction.

<ul style="list-style-type: none"> <li>· metrics-server</li> <li>· kube-state-metrics</li> <li>· datadog-agent</li> <li>· event-exporter</li> <li>· opentelemetry-operator</li> </ul>	<ul style="list-style-type: none"> <li>· cluster-autoscaler</li> <li>· karpenter</li> <li>· cloud vendor csi services</li> <li>· kubecost cost-model agent</li> <li>· gatekeeper</li> </ul>
---	---

### 1.6.7 Managed Control Plane Extensions

Services that extend the capability of the Kubernetes API, enabling administrators or customers to create other services or artifacts are managed as part of this domain. These capabilities normally require a higher level of care as compared to control plane services. By having these in a separate domain the platform team can carefully control upgrades and maintenance that could produce side effects to workloads or the platform itself. These are typically open-source or third-party services but also include custom operators or controllers with the following characteristics:

1. It extends the Kubernetes API by enabling Customers to provision and configure a resource that is not part of the Kubernetes API definition.
2. The resource may exist within the cluster but generally is external to the cluster, such as an external infrastructure component.

When attempting to provide Platform customers with the capability of deploying and managing non-cluster native resources, you are more likely to encounter organizational challenges. A functional boundary is very likely to be crossed, and for this reason alone, it is worth maintaining this distinction at the implementation level. Several examples are listed below. Each item includes the description of the resource a user can provision or manage using the extension.

- cert-manager: request certificates from an external certificate authority.
- crossplane: provision and manage other cloud infrastructure components.
- external-dns: creation and manage external DNS records.
- istio: connect services to ingress points, configure traffic routing rules, define circuit-breaking parameters
- Vault-agent: dynamically import application configuration from external source

### 1.6.8 Platform Product Services

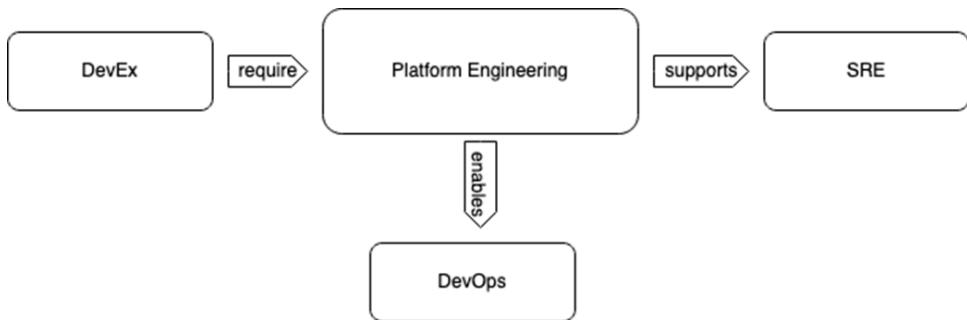
A broad domain that includes the more full-featured third-party applications provided to customers and maintained by the Platform team, such as ArgoCD, Prometheus, or Kiali. Beyond that, it is the domain home for the technology elements of what the industry is now referring to as Developer Experience or DevEx. This will include capabilities such as Backstage, Sonarqube, language starter kits, and many other touchpoints.

In addition, Platform Product Services include the custom or *management* APIs created by the Platform team to tie together the entire platform experience.

## 1.7 Platform Engineering Enablers

In this section we will clarify the roles and interconnections of DevOps, SRE, and platform engineering, emphasizing their complementary nature rather than viewing platform engineering as a replacement for DevOps or SRE.

To maximize adoption and usage, developer experience must be at the core of all capabilities produced. DevOps and SRE practices built up in the industry are core to enabling the platform team to provide this, as shown in figure 1.8.



**Figure 1.8 Platform Engineering Enablers.** This view aims to ensure clarity regarding whether platform engineering is a replacement for DevOps or SRE. In our definition, this accessible reference tells us how it all fits together.

However, rather than being considered enablers, we often hear somewhat inflammatory and misleading statements about platform engineering replacing DevOps and SRE in an organization. This is due to a need for more precise definitions of these terms and how they are applied in a given context. To make it easier to understand our point of view, throughout this book, we will consider these to be practices that enable platform engineering practices with concrete scopes of definition.

### 1.7.1 Developer Experience

Developer Experience (DevEx) refers to the techniques and tools used to improve software developers' experience in developing and deploying their software. This scope covers setting up the development environment, provisioning the resources, writing the code, building it, testing it, and deploying it to various environments, including the production environment, in a secure, scalable, and observable manner. The goal of DevEx is not only to improve the experience but also to measure the improvements and achieve the targets needed for organizational success through reduced developer friction.

### 1.7.2 DevOps

DevOps is fundamentally a culture of bringing together development and operations by using the underlying best practices and principles of software development. DevOps, as seen in various organizations, should not be a team of people (DevOps Team) or a job description (DevOps Engineer). Creating a separate role to execute parts of the software development lifecycle with a different set of skills creates yet another silo and unnecessary handoffs that reduce the effectiveness of an organization. It is our strong belief and opinion, as firmed up by several years of seeing how it works across the industry, that the best DevOps model for software development is to have the same set of developers design the software, write the code, build, tests, deploy, monitor and support the code in production.

### 1.7.3 Site Reliability Engineering

Site Reliability Engineering (SRE) on the other hand, predates the concept of DevOps and refers to a field of software engineering where development teams practicing true DevOps principles, as explained above, can successfully hand over more mature products to an independent and separate set of people, for the software to be made more reliable, scalable and resilient. We do not expect all the products and services built by an organization to be supported through the principles of SRE. Unlike DevOps, the term SREs also refers to a team of people usually part of a centralized pool of resources, allocated to specific products based on whether the products have reached the level of maturity to be handled by a set of people with the ability to manage the functional code and the infrastructure needs in equal measure. While SREs do minimal product design and architecture, they maintain the application, database, and infrastructure, make appropriate changes around bug fixes, and help improve the reliability and resiliency of the overall product as needed.

## 1.8 Impact of Generative AI in the PE space

Given the tremendous growth in interest and innovation around large-language models and other forms of generative AI, it is important to note how this is currently impacting the approaches for platform engineering and building platform experiences.

We think of AI's purpose as complementing and augmenting the proposed goals of platform engineering as defined in section 1.1 (reducing developer friction and improving the experience) by helping the PE solutions scale with increased usability, reliability, and extensibility, with security, observability, and sustainability considerations embedded in them.

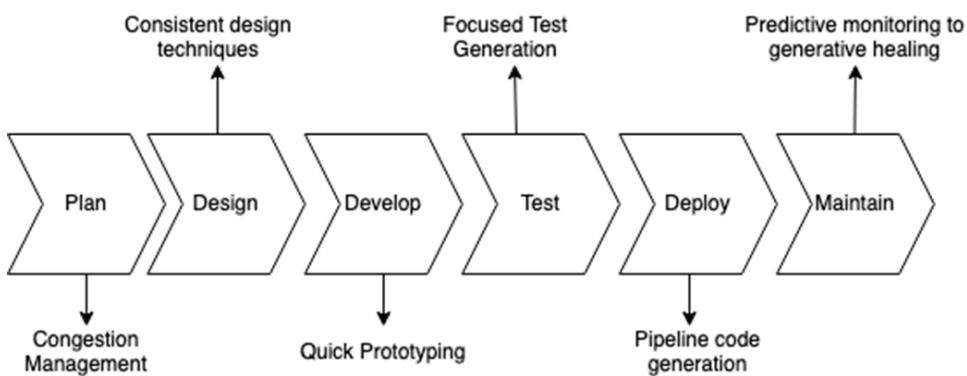
Predictive AI (leveraging AI techniques to answer various What-if scenarios in Platform Engineering to make the decision-making process more straightforward) and Applied AI (leveraging AI techniques to solve multiple SDLC challenges) have always been part of the platform engineering space. A critical approach to predictive AI becoming more prevalent in recent years is the increased usage of decision trees and models to hone in on a specific approach when there are many choices. An example of predictive AI helping this is when you are trying to look at your application pattern and figure out the appropriate runtime configuration. As of the writing of this book, some public cloud providers provide nearly 20 different options for using virtual machines, serverless, PaaS, Kubernetes, Batch, Step functions, and other ways to run your applications. This number will increase as technologies evolve, requiring a codified approach to making the right decisions, most probably through your engineering platform. These can be highly complicated for the developers and architects to recommend. Moreover, inconsistent usage of the solutions to the architectural patterns can create more operational challenges for all the downstream teams. Applied AI, on the other hand, has always had a significant impact whenever there has been a data-driven decision-making process. This is where you can apply the learnings from prior executions of similar solution patterns of a problem to the current scenario under consideration. The critical concept of self-healing is an example of using Applied AI. Both of these should be considered relatively new under the renewed interest in Generative AI.

We expect and are already seeing Generative AI being able to generate complete or partial solutions to the platform engineering problems with sufficiently clear prompts. The challenge in the space here is the availability of appropriate Large Language Models (LLMs) to help generate the solutions that are appropriate and contextually aware for a given organization.

We had earlier defined the product strategy for a platform in section 1.5. While describing the need for the reader to understand the overall vision and strategy, we also clarified that strategy creation would not be the primary task of many of the readers. However, executing a specific roadmap and actions around it requires a clear understanding of the underlying strategy. In this context, we must consider the impacts of Generative AI, as several tools in the industry will help you in that process. But suppose you start thinking about a Generative AI strategy tool available in the market. In that case, you have to understand the areas of focus, the need to evolve the strategy, and, ultimately, the techniques required to develop the strategy.

### 1.8.1 Identifying areas of focus

The best starting point to identify the areas of focus for Generative AI will be to create a first-pass strategy followed by a detailed path-to-production analysis, which will tell you the areas within the SDLC where you have the most inefficiencies. Once you have that, you should look at the places you plan to invest within the following phases under the general categories shown in figure 1.9.



**Figure 1.9 Focus Areas for Generative AI in Platform Strategy.** This is a rapidly evolving space, and focus areas should be identified using the same product strategies discussed throughout this chapter.

From a strategic point of view, you should first identify the focus areas and then break down the problem into multiple steps in the evolution.

### 1.8.2 Evolving Strategy with GenAI

From a strategic point of view, you should first identify the focus areas and then break down the problem into multiple steps in the evolution. Let us look at this for each focus area we discussed in the previous section.

**Plan:**

As part of the planning, you would typically conduct empathy interviews and examine the available data exposed using a detailed path to production analysis. You will also examine some lagging indicators, like DORA metrics, to better understand the problem space. However, could you feed all the data into specific usage patterns that can provide you with a comprehensive set of requirements? Could that be further prioritized using the exact predicted value for each condition?

Generative AI can be effective in analyzing the output of the planning process to identify priorities. One critical element of prioritization is the ability to understand conflict with inconsistent requirements, which is another easy consideration for Generative AI techniques.

Another area of exploration is the ability of Generative AI to turn planning and acceptance criteria analysis into actual user stories. If not completely, textual assistance can accelerate the effort of business analysis and product owners by reducing some of the toil in packaging prior work into various forms based on the audience.

## **Design**

By now, we have all seen the tools Generative AI provides to generate architectural diagrams, data models, and deeply descriptive visualizations for your designs. These translations occur with the power of the knowledge of similar design patterns in the models used with the singular focus on improving the design experience and the product's usability and quality.

Tools have been available in the industry that continue to automatically create ERD for your design specifications and provide you with options to select from a variety of selections driven by either the design criteria set forth before you embark on the process or, in some cases, enabling an iterative process. These options generated by the AI models can tell you the potential cost of implementation and the performance considerations, which makes your decision-making process more solid.

Anytime you are putting together a strategy, the biggest challenge is figuring out how soon you can prove your hypothesis. Addressing this challenge requires you to build proof-of-concepts that can give your end users and decision-makers a better feel for what you are proposing to develop. Simply showing them a document or a spreadsheet with numbers will go only a limited way in aligning them with your thinking. Generative AI can be a potent tool in this space by providing a streamlined way to generate your prototype quickly.

## **Develop**

Most of the terrifying stories we have been hearing about since the advent of ChatGPT 4 have been about whether GenAI will replace the developers. While we have concluded that this is still a far-fetched idea, the fact that using AI will make good developers better and more productive is inevitable. Specifically, this includes facilitated code generation based on the design with all the necessary documentation, smart code reviews that are more timely for faster feedback, and debugging to ensure functional equivalence. Another bane of the developers is the ability to develop unit tests that can help your continuous integration go through seamlessly. Some of the most popular tools currently around assisted coding provide developers with great real-time insights into making the development process more efficient and enjoyable with clear outcomes.

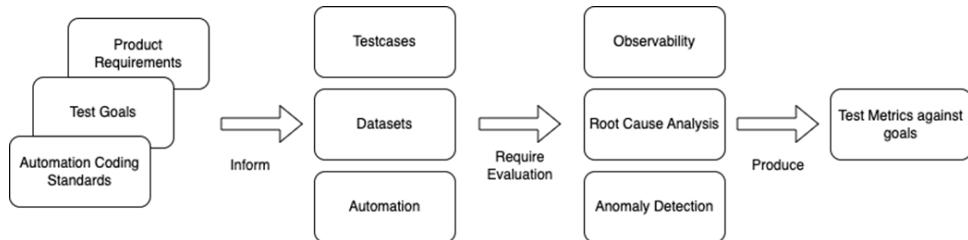
If you are not a startup or a scaleup, you are bound to have a significant amount of legacy code architected in a suboptimal manner. While the models still need to be better to make a tightly coupled architecture broken down into manageable modules, many tools are coming into the market for migrating mainframes and codebases that are difficult to find developers for. Considering these in your platform strategy will help determine the proper investment and focus levels.

**Table 1.1 How GenAI can help improve development velocity**

Condition	How can GenAI help?
Better analysis of complex coding issues	LLM-generated solutions can be effective in resolving difficult-to-solve coding challenges
Improved test coverage	Certain categories of tests can be generated, reducing coding time
Earlier feedback on static code analysis	code complexity and maintainable issues can be discovered sooner, and possible solutions implemented more quickly

## Tests

Testing has the highest potential of all the opportunities to improve the SDLC. The scope of this work typically centers around the following areas. In figure 1.10, we show the typical test evolution from a GenAI point of view.



**Figure 1.10 Test Evolution from a generative AI point of view. GenAI techniques can help perform efficient root cause analysis by quickly evaluating observability data.**

As shown above, a more holistic impact can be expected within your testing by focusing on test case generation and generating more automation to conduct these tests. Once the tests are executed, the generative AI techniques will help you do much better root cause analysis by allowing you to go through your metrics, logs, and traces and ensuring a heavy lens of observability is brought into the ecosystem. At that point, you will also need the ability to conduct the anomaly detection.

## Deploy

Generative AI will play a huge role in releasing your software for use across various environments and improving the customer experience. The obvious ones are security checks based on the changing usage patterns and the known changes in the compliance space.

## Maintenance

Using GenAI techniques can improve Software support and maintenance. Here are some areas to consider when developing your strategy for software maintenance capabilities as part of your platform buildout.

1. Improving end-user experience: Consider the usage of an NLP and deep learning chatbots to create more real-time responses to the most common questions one might have as part of using the system
2. Alerting and resolution: Continuously identifying drifts within your system against sensible defaults and acceptable ranges can lead to informed alerting and insights that can lead to automated remediation.
3. Feedback cycle: GenAI can rapidly discover usage patterns that can guide roadmap prioritization decisions.

The impact of generative AI as a tool in the evolution of your platform strategy should be evaluated in the same way as any other product strategy. Much of the current analysis continues to show that ML or LLM-generated ideas are wrong more often than they are right [\[1\]](#). Yet, they can also produce very valuable insights. Use Platform Value Measure techniques to continually assess whether the return on investment from incorporating GenAI is justified.

## 1.9 Summary

- Platform Engineering is a craft composed of the architectural, engineering, and product delivery disciplines applied by dedicated engineering teams in an Engineering Platform's ideation, creation, delivery, and evolution.
- Effective platform engineering teams will work to deliver engineering platforms that provide internal software development teams self-managed and seamless access to the tools and technologies they need to innovate, create, release, and operate their software without the usual toil, delays, and cognitive load.
- Platform engineering principles and practices should be adopted in an organization's evolution as early as possible.
- Platform engineering teams are software engineering teams that deliver internal products to stakeholders and users throughout the organization.
- Platform engineering requires a strategic approach with a product mindset to differentiate it from developing automation that can improve productivity
- Platform engineering principles should be applied during the development of platform products and enabled via automation and guardrails for engineering teams who use these products.
- The development and delivery of engineering platforms should follow domain-driven design principles.

- Implemented correctly, platform engineering is neither a buzzword, nor a replacement for the cultural paradigm of DevOps or the principles of Developer experience or the practice of Site reliability engineering. Instead it is an integral component of the ecosystem that brings together all of these related concepts to make developers lives easier and improve an organization's ability to deliver their products more efficiently leading to successful business outcomes.
- Generative AI helps identify critical areas for platform strategy improvement (planning, design, testing, etc.) and accelerates these phases through automation and prediction. Generative AI tools can generate code, design documents, prototypes, and test cases, freeing developers for more strategic tasks. The most significant impact of Generative AI will be in testing, where it can automate test case generation, anomaly detection, and root cause analysis.

### 1.9.1 Chapter Footnotes

[1.5a]

1. <https://www.thoughtworks.com/en-us/perspectives/edition6-product-innovation>
2. Sinek, Simon. 2011. *Start with Why*. Harlow, England: Penguin Books.
3. Cohn, Mike. 2004. *User stories applied*. Redwood City, CA: Addison Wesley Longman Publishing Co., Inc.
4. Osterwalder, Alexander, Yves Pigneur, Patricia Papadakos, Gregory Bernarda, Trish Papadakos, and Alan Smith. 2014. *Value Proposition Design*. New York, NY: John Wiley & Sons
5. Lencioni, Patrick M. 2002. *The Five Dysfunctions of a Team*. J-B Lencioni Series. London, England: Jossey-Bass.

[1] <https://www.abtasty.com/blog/1000-experiments-club-ronny-kohavi/>

## ***2 Foundational Platform Engineering Practices***

### **This chapter covers**

- Delivering a platform using the principles of product management
- Cloud-Native Technologies and how they impact platform engineering
- Software Defined Platform and its relevance
- Evolutionary Platform Architecture
- Domain-Driven Platform Design

Building an engineering platform without changing organizational structures, decision-making habits, and engineering culture is a strategy destined for failure. However, simultaneously shifting an entire organization's mindset, structures, and culture is a complex and challenging task that requires careful planning.

As your team is forming and defining the goals of your platform, you will start to think about how you can enable yourselves to build a successful engineering platform in an efficient and well-designed way. Some of your teammates may wonder if there should be certain linters - a set of tools required to improve the quality of the code - in place for the infrastructure-as-code you will write. Others argue about using Git as a truth source for deployments, or if some form of artifact registry is needed. In another meeting, you debate the merits of CI/CD systems, with some preferring a more traditional system like Jenkins, with others who want to go full cloud-native with modern Kubernetes-enabled delivery tools. Your team starts to wonder if you'll ever get started on doing any actual work because you're too busy arguing about the semantics of how to do the job.

When a platform team has disagreements, one way to sort them out is to remember a key point: platform engineering is an exercise in software development. You should use the same methods to develop platforms you would use for building any other application, and the team should run and test their software using a special version of the same platform tooling as everyone else. This practice is often called "dogfooding," which means using our own platform to help us improve and build it.

In this chapter, we'll talk about what it means to build a software-defined platform in the product mindset we introduced in chapter 1 using principles from *evolutionary architecture*, like how to design and write software that is easy to change and scale, and applying them to our software design. Let us also define what a platform product is. A **platform product**, first introduced in chapter 01, is a technology solution designed to serve as a foundational layer or a "platform" upon which other products, services, or systems are built, extended, or integrated. We'll also talk about how we can use these principles to draw circles around the types of people that operate with our platform and use this to make decisions about the APIs and services we build. Some of the questions we will answer are:

- What is the model for delivering a platform product?
- How are the advances in cloud native technologies helping platform engineering?
- How do we use the platform to help us build and evolve the platform?
- What are the domains and boundaries of the systems and services in an engineering platform?
- How do we evolve and iterate on the platform using a software-defined approach?
- What are the differences between a software-defined platform and an off-the-shelf platform?

## 2.1 Product Delivery Model

In this section, you will understand the foundational concepts of infrastructure, developer tools, and product delivery approaches which are essential for effectively creating and managing an engineering platform, which continuously evolves to meet customer needs and organizational goals. This foundation is crucial as we delve into the roles, strategies, and practices necessary for successful platform engineering.

In many enterprises, not all, when it comes to infrastructure, developer tools, and other technologies needed to create and operate custom software, they will treat these as separate, one-off projects. Usually, someone who isn't a software developer decides which tool or technology to get and which vendor to buy it from. They negotiate the purchase like it's a typical capital investment, expecting it will be used as-is for years to come, and then pass it off to another non-user to set up, all accompanied by a detailed project plan that comes complete with a Gantt-style project plan of weekly milestones.

To do platform engineering right, you need a product delivery approach. For product delivery to really work, we must move away from the old-school IT delivery methods.

**ENGINEERING PLATFORM** An Engineering Platform is a product designed to empower software developers to independently (self-serve) create, test, launch, manage, and monitor custom software solutions in an ongoing improvement cycle

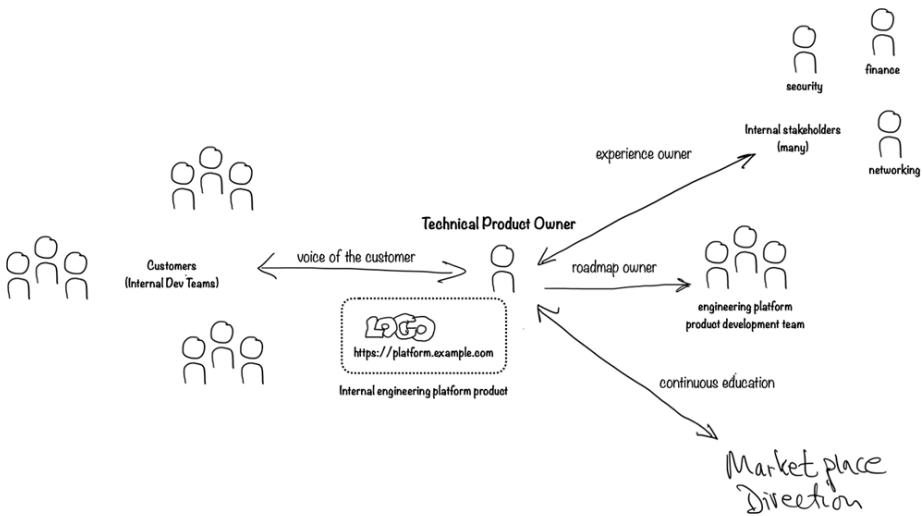
Products are meant to last a long time, constantly improving and evolving to keep up with new conditions, technologies, and the features customers want. Change isn't just a one-time project; it's how software operates. A product delivery model is all about the continuous, ongoing development of the product. Adding features that aren't self-serve messes up the product delivery model. Building or implementing technologies in a way that makes change too expensive also breaks the product delivery model.

Things you need to know to be effective as a platform engineer working with the engineering platform product team include

- Is there actual Technical Product Ownership of the Platform?
- Is the experience of using the platform being targeted to the customer or merely to meet stakeholder concerns?
- Are the architecture and engineering practices optimized for building a product?
- Is the Platform being delivered in rapid incremental steps, starting from a minimum valuable set of capabilities?

### 2.1.1 Technical Product Ownership

Figure 2.1 shows the relationship between various entities while establishing the platform product ownership. Technical Product team, led by a Product Owner, delivers unique capabilities, focuses on the relevance of the product capabilities for the skilled technologists, and adapts feedback approaches for internal users.



**Figure 2.1 A Technical Product Owner (TPO) serves as the voice of the customer while understanding the technical needs of internal engineering teams. This role will manage the product roadmap to ensure that features are developed that will meet customer needs and provide the intended value.**

A true product team takes full ownership of what it delivers. It's the only team in the organization providing a specific set of capabilities, and it's led by a Product Owner who has the responsibility and authority to set the team's product roadmap.

The product owner is often called a *Technical Product Owner* (TPO) for an Engineering Platform. This can be a helpful distinction as it clarifies within the organization that an engineering product's users are skilled technologists with very different needs and expectations from general consumer products. The users are internal, with different approaches needed for developing effective feedback loops. However, many of the fundamental activities remain the same regardless of the complexity or technical nature of the product.

- They are the voice of the customer. They deeply understand why customers use the product and what provides value. You might want to think of the customers in this scenario as your end-users.
- They find ways of meeting stakeholder needs without impairing the customer experience.
- They are responsible to know and understand where the marketplace is heading.
- They prioritize the backlog, deciding on the order of release for platform capabilities, engineering improvements, or the refactoring of Platform technical debt.
- They change the product roadmap when observations of actual usage or customer experiences indicate the expected value is not obtained.

Most organizations initially misunderstand the importance of the TPO role and assume traditional IT processes can be used to deliver an engineering platform product. This assumption is destined to fail. Whether you are shipping skateboards or an engineering platform, a lack of actual, empowered product ownership will produce a product far different from the initial vision with significantly reduced value.

### **2.1.2 Customers vs Stakeholders**

When we talk about who really uses an engineering platform within the organization, it's key to notice the differences. Some folks, like software developers and engineers, might use just about every feature almost every day. Others might only have a subset of the platform's features they use regularly. Then there are those who only use the platform occasionally or in specific situations, whether they're using many features or just a few. Remember, the real customers are the ones who actually use the product. When deciding who to focus on first, start with impact and value they bring to the organization. The same goes for choosing which features to add or beef up—think about who will use them, how often, and the overall value they'll bring.

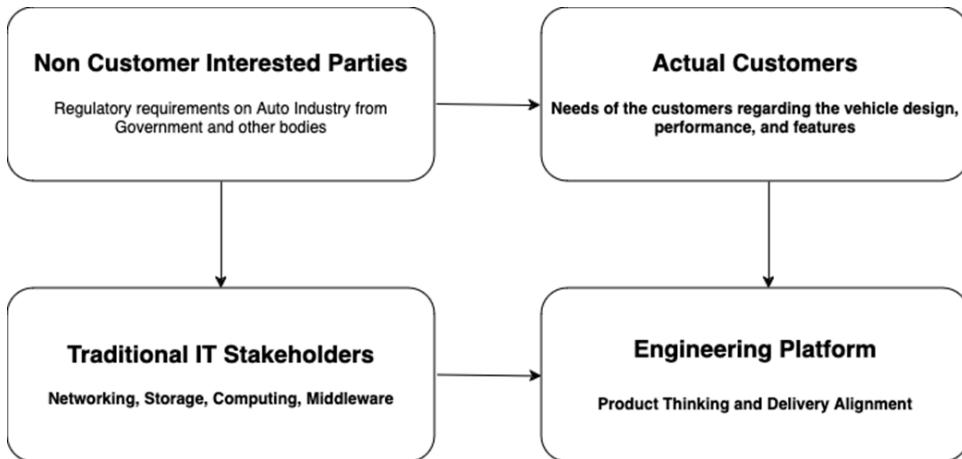
But then there are also plenty of influencers. These are people who aren't actual users but, thanks to their roles as leaders or decision-makers in other parts of the company, have a big say in shaping or nudging the platform's requirements. They're invested in the outcome and influence things from the sidelines—they're not the customers. They are the stakeholders.

Think about the automotive industry as an example. Car companies start life building only a certain kind of vehicle. Initial success comes from offering a single category, such as a high-end performance sedan. Ferrari is an excellent example of this. Ferrari built their first passenger sedans in 1947 and stuck to building only sedans, not trucks, SUVs, or long-haul buses. Eventually, in 2022, came out with their *Purosangue* line which ventured into the SUV market. Ferrari was able to maintain their premium fast car brand first due to the prioritization and optimization of what was perceived to be the most significant potential customer pool for their core product.

There are also many non-customer interested parties. Take state and federal governments, for example—they have a laundry list of requirements ranging from safety to taxation to insurance. Imagine if Ferrari decided to prioritize these government demands above all when designing their cars. They'd end up with a vehicle that probably nobody would want to buy. If the government set requirements so strict that they left no room for creativity or flexibility in car design, automakers would likely move their sales to markets free from those constraints.

When you cater to stakeholders at the expense of your actual customers, you risk losing those customers. In the context of engineering platforms, this means you could miss the mark on your platform's goals, not deliver the intended value, and see the proliferation of workaround solutions. Can you say "shadow IT"?

In the figure 2.2, you can see how the four different components - non customer interested parties, actual customers, traditional IT stakeholders and the engineering platform - relate to each other



**Figure 2.2 A four way relationship between the key components that shows how the customers and stakeholders drive the engineering platform requirements**

Many Enterprise stakeholders, including finance, security, legal, and project management, will expect to have a say in the platform's delivery. Traditional IT stakeholders may expect to own the technology capabilities found within an engineering platform through the siloes of networking, storage, computing, operating systems, middleware, Active Directory, and the like.

If the expectation is that the legacy silos can remain and behave in legacy ways while we somehow wrap an engineering platform around them, you will not succeed. If that were possible, adding the DevOps team would have been the solution. But it's not about creating a wrapper; it's about introducing a whole new way of working.

For an engineering platform to achieve its potential, everyone involved needs to agree on how we think about and deliver products. All stakeholders must align around the model of product thinking and product delivery. We will explain how this alignment happens through a simple scenario below.

What is the starting point for these stakeholders in this new product model? The most basic way to summarize this way of working is that Stakeholders must provide either:

- a service interface (API) to the capabilities they own that enables all internal customers who need the capability to obtain it in a self-serve and self-directed manner (e.g., as much of the service as they need, whenever they need it) or
- well-defined outcome standards to which all impacted teams within the enterprise can self-solution.

Take penetration testing as an example of a service interface. Typically, this falls under the Security Team's umbrella. To fulfill the *service interface* requirement, the Security Team needs to set up a self-serve system. This way, any developer in the organization can grab the credentials they need, run scans on the endpoints they're working on, and meet the security requirements—all without having to sync up or even talk directly with the Security Team. One might argue that this is a less-than-ideal outcome if the developer does not care about secure code. However, in reality, the developer is not only educated that writing secure code is required but can also do it in the easiest manner possible.

In practice, a security scan will become part of the platform's reusable pipeline code library. No matter how the security scan service interface ends up being used, offering it as a self-serve option ensures the security team meets the requirements. Plus, this feature can be easily integrated into the Engineering Platform or any internal product that needs it.

The consistent application of this practice is one of the main reasons building an engineering platform on a cloud infrastructure provider, such as AWS or Google Cloud, is almost a requirement. Every service they offer is accessible via an API and designed to support independent, multi-tenant user patterns. While modern private data center vendors like VMware and Cisco do offer products with comprehensive APIs, and there are open-source solutions like OpenStack for creating private clouds, many enterprises struggle to implement these in a way that supports this level of usability due to their traditional IT operating models.

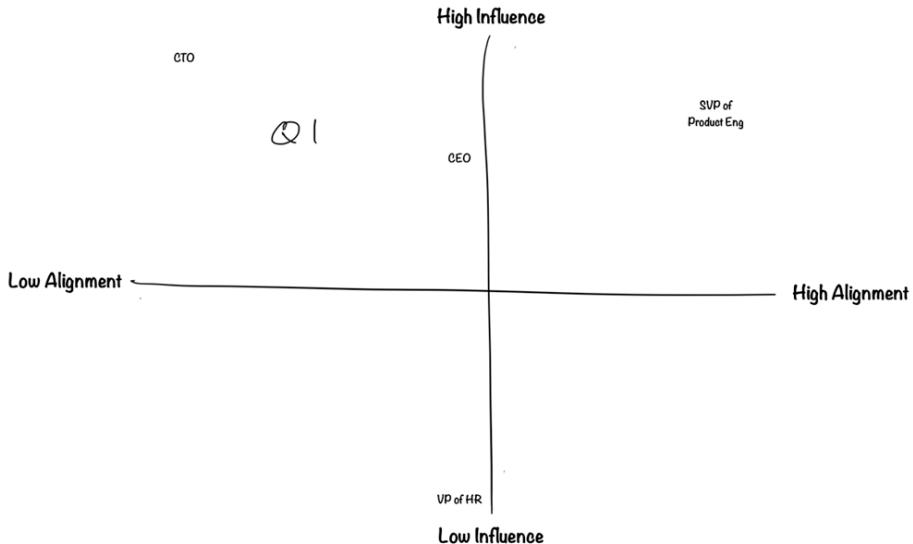
The second bullet is equally important as it is not typically the case that all stakeholders will have the resources or expertise to provide a service interface to their domain. In that case, they provide a well-defined and testable set of outcome requirements. A good example would be a security stakeholder defining corporate standards as 1) requiring a single source of truth for internal authentication and 2) the internal product security standard is Oauth2.[\[1\]](#).

An internal identity domain team could independently select and implement the single-source authentication service and provide self-service API access for teams building anything that requires internal authentication. But, if the existing solutions did not fit their product goals an internal product team could self-solve an authorization technology and implementation so long as it meets the Oauth2 standard.

So long as stakeholders follow one or both of these approaches, other internal teams are never blocked waiting for the stakeholder to perform work on their behalf nor do they have to create high-friction solutions for their customers.

When you've assembled a fresh platform engineering team, it's smart to sketch out and maintain a stakeholder map from the start. Track all the stakeholders who impact the platform so you can identify situations where a stakeholder's current or planned solutions or processes will not work with the Platform.

Use a diagram like the one shown below to determine where the various stakeholders currently stand regarding their influence and alignment.



**Figure 2.3 A Stakeholder Map to evaluate a stakeholder's influence over the backlog against their alignment to platform objectives. Ideally, high-influence stakeholders should be highly aligned to ensure platform success.**

First, how much influence does the stakeholder have or want to have over the engineering platform? In the example diagram, the VP of HR is neutral in aligning with the platform's goals and has meager influence, as you might expect. The CEO is highly influential, though typically delegates their responsibility to others and attempts to reduce their direct influence beyond the broader corporate strategies. In both situations, the neutral alignment may have no negative impact. However, the CTO is highly influential and much more likely to be directly involved in setting operating models or budgets that directly influence an Engineering Platform.

Are they highly aligned, providing the platform engineering team with API access or outcome requirements? Or are they unaligned, requiring the platform team to open tickets for needed configuration or conforming to existing operating models, whether they fit the needs of the engineering platform or not?

People can disagree for all sorts of reasons. It could be due to a real difference in strategy, or maybe someone just needs a little more time or resources to get in sync with the overall vision. An accurate and maintained stakeholder map is a highly effective way of tracking delivery risk from internal stakeholders and processes. Quadrant 1 (Q1), of course, is where to focus. For each stakeholder in Q1, list their objections or constraints. Find the source of the constraints. Sometimes, a constraint imposed by one stakeholder results from a constraint placed on them by another stakeholder. At some point, a solution will be needed for each of these issues. Failing to do so will result in a miss in some aspect of the product vision. Each miss is a reduction of value, whether small or large. Enough misses, and the entire investment is at risk.

It's really important to spot and keep an eye on any misalignments early on. This can really save you from a lot of criticism later on, especially when you're diving into something as complex as deploying an engineering platform. Introducing this platform means embracing a whole new way of thinking and working, which can sometimes bump up against the existing culture and processes. Making it clear what's changing and how much change is happening helps prevent people from wrongly blaming the goals when things don't pan out as expected. Being upfront and transparent about these mismatches early on is crucial to sidestep any later criticisms of adopting a product-centric mindset in such a tough project. To paraphrase G.K. Chesterton: It's not that engineering platforms have been tried and found wanting. It is that they have been found hard and not tried [2.4.b].

### **2.1.3 Exercise 2.1 Build a Stakeholder Map for Your Organization**

Assume your organization has decided to build an internal engineering platform. Perhaps you are already building one. As you think about the current challenges, there are probably many areas where product thinking would face or is facing challenges from the interaction of various stakeholders within your company.

Create a stakeholder map and populate the map with several relevant stakeholders. Pick one stakeholder from Quadrant 1 and create a list of their objections or constraints.

Describe how the capability owned by the stakeholder could either be

- provided as an API that provides the platform engineers with a self-serve experience or
- defined by a standard that would enable the platform engineers to solve for the desired product experience while meeting the needs of the stakeholders.

### **2.1.4 Optimize for a Product**

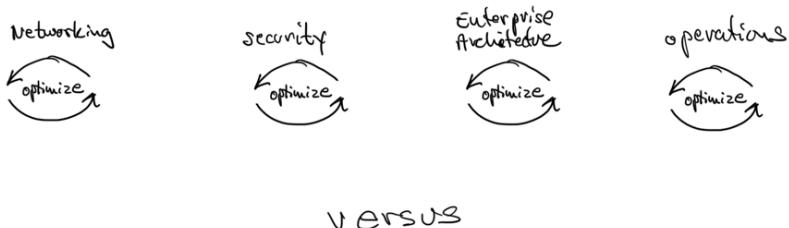
Deciding to build an internal engineering platform is, in large part, a response to the challenges and shortcomings of your enterprise's current setup. Your current setup results from your current organizational structures, how decisions are made, and the overall engineering culture.

As a platform engineer, how do you know which aspects of your organizational structures contribute to engineering and product quality and which detract? There isn't an easy answer. However, after building engineering platforms in dozens of national and multinational enterprises, three key practices have repeatedly been demonstrated as leading indicators.

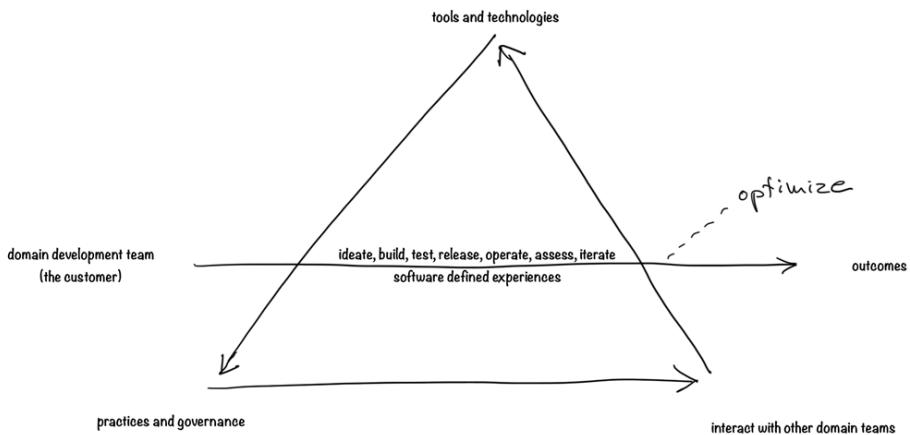
- Is the entire end-to-end process for building, delivering, and operating software being assessed and engineered for quality and success against the product strategy?
- Are all features or capabilities of the product delivered as a service interface first?
- Are commitments to architectural and product experience decisions made based on results of actual implementations and observations of customer usage?

## END-TO-END ENGINEERING OPTIMIZATION

Each silo optimizes for its own needs, budgets, values, opinions about 'who is the customer', and understanding of priorities.



Engineering quality and architectural strategy impact measured against the entire end-to-end flow.



**Figure 2.4 Look at all the activities, from ideation to operations, and assess engineering quality and impacts from organization or architectural decisions. This Developer Activity model is inspired by Yrjö Engeström's human activity model. [2.4c]**

The developer activity model shown in figure 2.4 provides an effective means of determining the engineering platform's actual impact and, therefore, the scope of the assessment for e2e engineering effectiveness.

The goal is to fully empower independent, domain-focused development teams. These teams will come up with and try out ideas specific to their domain. They'll build and test software to create measurable experiences and value. Then, they'll release the software to their customers and manage it, ensuring quality user experiences while keeping an eye on the results. Are customers using the product as expected? Is it delivering the intended value? Is it resilient and performant?

This range of activities outlines what the development teams need to do. They must handle these tasks efficiently and with minimal delays, focusing on actions that directly support these goals. This entire set of activities defines what the teams need to be enabled to do. They must excel at these tasks and minimize time spent on anything not optimized for achieving these goals. Regardless of how much the engineering platform handles at any given time, quality and results should be measured against this entire scope. In other words, design quality should be evaluated from the start, considering the whole workflow.

Historically, IT organizations have been structured around functional tasks. When optimizations happen, they occur within these functional silos, each with its own measures of success, budgets, management values, and working methods. What might seem like an optimization within a silo can actually increase the overall time and reduce quality when you look at the development process as a whole. This means there must be a willingness to change organizational structures, decision-making methods, engineering culture, and anything else needed to support effective product delivery. If you don't change the organization, then you can't expect to get different results.

## **SERVICE INTERFACE FIRST**

All features of an engineering platform should be designed, built, and delivered as service interfaces (APIs) first. You'll include additional user touchpoints, like CLIs and UIs, as part of the overall user experience. But no matter the type or number of touchpoints, the core capability is accessed through an API. This architectural choice is the key to ensuring product flexibility, quality, and longevity.

## **BASE ARCHITECTURE AND TECHNOLOGY COMMITMENTS WITHIN THE PLATFORM ON REAL-LIFE EXPERIMENTATION**

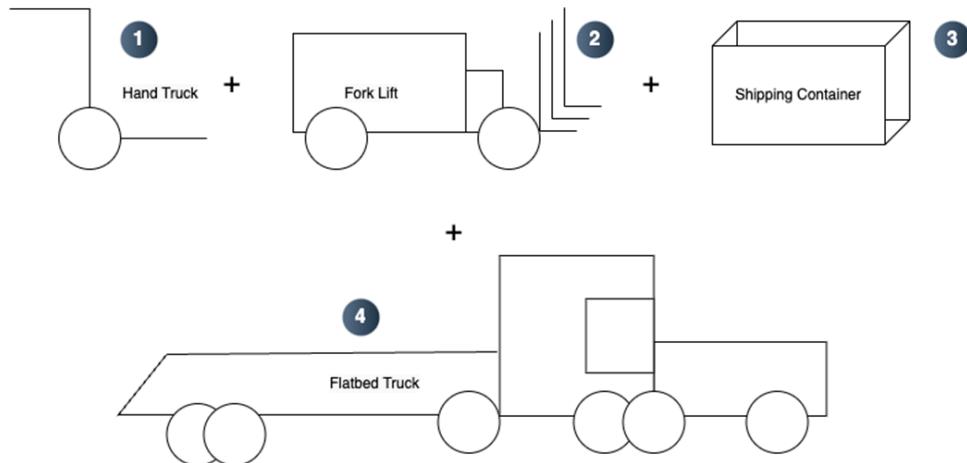
Commitment here means the final decision to build and release a production feature or capability. These commitments should only be made after experimenting with the architecture or technology options in a real-world setting. This doesn't always have to mean live production, although that often gives the best results. It can also involve working proofs of concept where platform customers test things out with their actual use cases. And in every case, include observations of how real customers use the feature. Customers routinely use features in ways we cannot anticipate or overestimate the importance of a feature before it is available.

**WARNING** When deciding on implementing a durable message queue, an anti-pattern would be for the engineering platform team to independently assess available solutions and attempt to find something with the broadest range of features and capabilities, ostensibly to try to future-proof the decision against future requirements. However logical this may sound, in practice you simply cannot anticipate future requirement and bloated, do-everything technologies underperform and are routinely more costly than a handful of smaller, optimized solutions

### 2.1.5 The Importance of a Minimal Valuable Product and Early Adopters

An engineering platform needs to evolve based on actual customer usage from the very start. In product development terms, this means identifying the minimum valuable version of the product and releasing that first. It's not always clear what functionality the first version needs for customers to find it valuable, and opinions within your team or company may vary. Whatever you initially decide on, treat it like a hypothesis. Carefully analyze the behavior of early adopters, look for evidence that shows whether the initial features are effective, and be ready to change priorities if the evidence doesn't support your initial ideas. This approach applies to both the MVP and every new feature you release.

Probably everyone working in or around software development is familiar with the Minimum Viable Product (MVP) delivery diagram. It starts with a skateboard, which evolves into a bicycle, then to a motorcycle, and finally a car. This *physical-object* analogy, however, can be confusing from a user's perspective. Sure, a motorcycle is more useful than a skateboard, but if I believe I need a car, how do these steps help me? Here's an alternative way to think about it. Here is an alternative way of thinking about it.



**Figure 2.5 shows the Progression of an MVP delivery process. A fully functional product does not need to be delivered at the start, but all phases should deliver some incremental value to the customer.**

Imagine you need to move goods from your factory out to your customers. Suppose the solution progresses from hand trucks to a forklift to loading cargo containers onto flatbed trucks, with each stage being an additional component. In that case, you can see how each offers value and expands the capabilities. This model works well for an engineering platform because the platform team isn't usually the direct developer of most capabilities; they integrate various capabilities and services. There will be many evolutionary stages to cover the full scope of developer activities. You can start with the smallest valuable element and keep expanding until you achieve complete coverage. Each new feature will go through the same process of experimentation, proof of concept, and real-world usage before it's fully implemented in the platform. This approach will exemplify the idea that every capability in the platform is built with a product mindset and a product lifecycle, ensuring easy replaceability and maintainability.

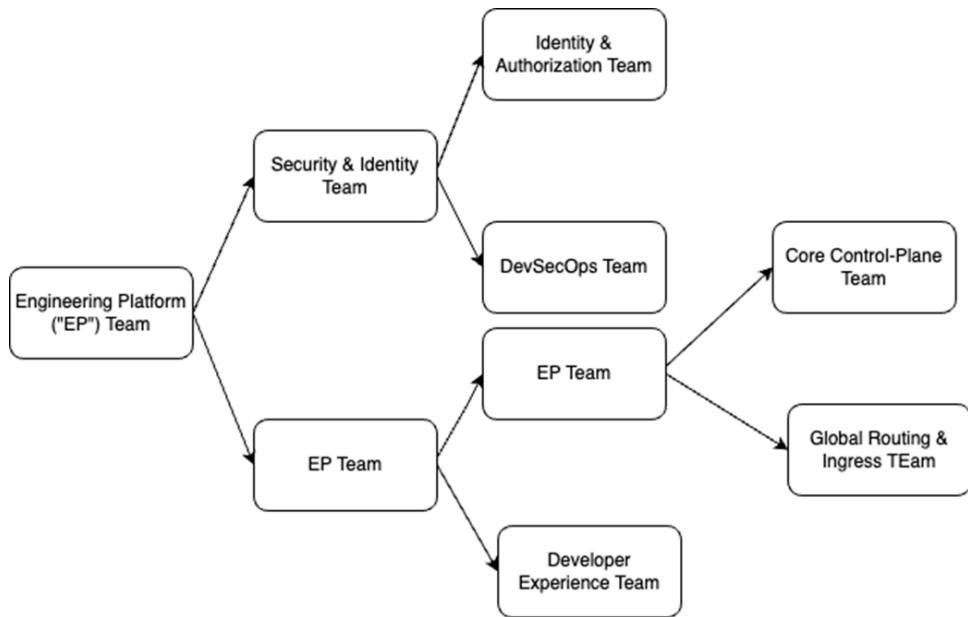
## DELIVERY CAN BEGIN WITH A SINGLE TEAM

"A complex system that works is invariably found to have evolved from a simple system that works."

- John Gall[2.4b]

Gall's insights really hit home when you look at engineering platforms. He pointed out something crucial, especially true for a platform: to build something complex and mature, you have to start with something simple that works and provides measurable value to at least a small set of users. That starting point is the MVP. We normally recommend that this initial offering be built by a single team and architected for a future roadmap where many teams are involved in delivering the Platform. However the absolute requirement is for a single, unified architecture roadmap and a single TPO maintaining this unified product vision. Over time, if you need to speed things up, you can go from one delivery team to several, each with its own product owner but all reporting back to the single, top-level TPO.

What does this evolution of teams look like? There is no single evolutionary path that will fit every enterprise situation. However, more than once, we have seen the following progression:



**Figure 2.6 shows the evolutionary path of platform development teams. Often, while an EP starts as an effort driven by a single team, multiple teams spin up over time to handle specific aspects of platform functions, with all development efforts under a top-level product owner.**

Starting with a single team allows a platform to start quickly, with a high-quality architectural design, and on a small scale to prove the functionality and value that could be generated. Over time, certain capabilities are so heavily used and have such broad impact that a dedicated team can be justified to manage it. Developer-experience teams is a common area, as is identity and DevSecOps. These areas quickly become important enough that a single team cannot manage those areas and at the same time continue to develop new capabilities for the platform as a whole. Eventually, even the core EP team begins to split as globalization is introduced, requiring dynamic routing between clusters to be supported by a dedicated effort.

### 2.1.6 Exercise 2.2: Define the feature set of an MVP engineering platform

Create a detailed list of the capabilities you believe are necessary to form the MVP for an engineering platform. For each, describe why the absence of the capability renders the MVP definition of the product unsuccessful.

## 2.2 Cloud Native Technologies

This section focuses on how to deliver the expected value from an engineering platform by developing a scalable and extensible architecture, leveraging cloud-native technologies to control costs and facilitate growth. Understanding this will help you navigate the complexities of technology choices, ensuring your platform is flexible and easy to change as it evolves.

To deliver the value expected from the engineering platform, we need to be able to show value quickly as we begin to execute the strategy and realize the value model. To do that, we want to develop an architecture that is scalable and extensible so that we can start small and grow over time. We also want to be able to control costs as we grow, so where do we start? How should we think through the vendors and technologies on the market? We can start with cloud-native technologies, which allow us to spin up quickly and grow over time.

### **2.2.1 How Do Cloud Native Technologies Help PE?**

Returning to our original storyline at PE Tech, you're starting to get a sense of Product Strategy and the mindset you will need to build your platform. But as you start digging into the vast world of Platform Engineering, you quickly realize you are in over your head. There are so many technologies to choose from, architectures to adopt, and SaaS platforms for sale that it's overwhelming. How do we even begin to solve this problem?

Right before you let that overwhelming feeling take control, you remember a quote from your trusty ole Pragmatic Programmer book:

"A good design is easier to change than a bad design"

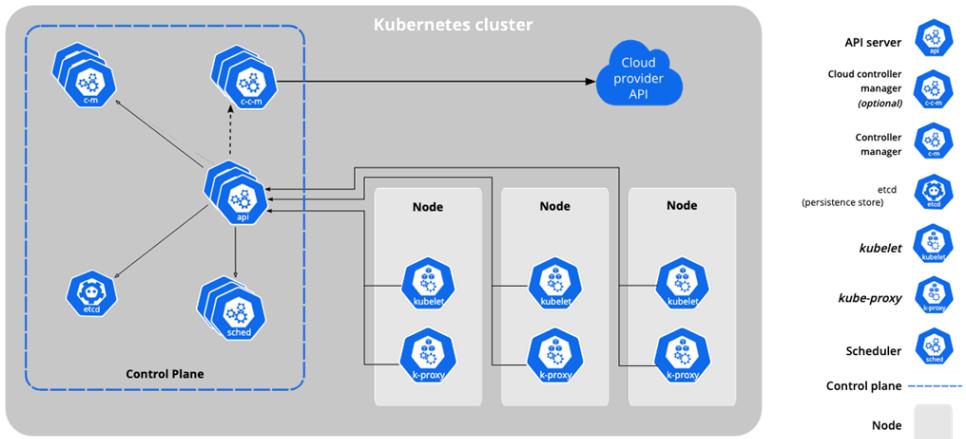
*Thomas and Hunt, Pragmatic Programmer*

To recall, what the authors are talking about here is that when faced with difficult technology decisions, we should always make the choice that is easiest to change later. This is true for building products and extremely true for building engineering platforms. As we'll talk about later in the book, Engineering Platforms are made of evolutionary building blocks and are constantly changing. If you made choices for your platform that were harder to change later, you would quickly find yourself in long development cycles and slow feature turnout.

So, how do we avoid all of this and stick to our principles? Well, this is where our focus on Cloud Native comes in. There are many arguments as to what Cloud Native (the term) is. This book centers around what Kubernetes has done to the tech industry, which is *standardization*.

Kubernetes, as you may know, is an orchestration engine for applications. However, Kubernetes also represents a standardized and agreed-upon set of APIs. No single cloud vendor is controlling this project. Many committees and Special Interest Groups govern it with rotating chair policies to prevent leadership decay. They have vetting processes for new APIs and features, code reviews for changes to the project, proposal standards to make changes, and lively debates and discussions surrounding the project's future that any member may participate in. Much of the cloud native ecosystem is built around Kubernetes.

The basic components of a Kubernetes cluster are pictured below



**Figure 2.7 The components of a Kubernetes Cluster. Cloud PaaS offerings will usually serve the control plane as a service, leaving node management to the user. In “serverless” cloud offerings, worker nodes will also be offered as a service. Courtesy: <https://kubernetes.io/docs/concepts/overview/components/>**

In brief, a Kubernetes cluster will have a control plane containing the services used for cluster operation and management and a set of worker nodes used to run applications. The control plane includes the services to perform CRUD operations on cluster resources, manage service discovery, orchestrate scaling operations, and more. The worker nodes are used to run user services and applications. These include the applications and runtime services deployed to the cluster by both the PE and engineering teams using the runtime. Most cloud PaaS offerings will include a managed control plane, with the cloud provider assuming responsibility for updates, high availability, and scaling operations of that layer. Worker nodes are generally the user's responsibility, with options ranging from full control of the instances used to run worker processes all the way through “serverless” options in which the worker nodes are managed for most administrative tasks.

As the PETech engineer, you now think: “I've heard a lot about this native cloud stuff, but I'm unsure where to start. I know I'm not going to get everything right the first time, so I need to make choices that are easier to change later. Right now, I just want to get a foundation going for my Engineering Platform”

How would you solve this problem?

1. Buy an engineering platform off the shelf
2. Install an open-source engineering platform
3. Build a backlog and start writing your platform from scratch
4. Choose an option that doesn't lock me into any one cloud or way of working

If you immediately thought of the last option, you may know where this is going. The lesson to be learned from Kubernetes, and many other successful large-scale projects like it, is that the diversity and distributed nature of the governing body lead to the creation of a project that is flexible, extensible, and provides a foundation that is easy to change.

However, if you immediately thought, “Install Kubernetes!” (option 2), you missed the point. We are looking for a foundation that gives you flexibility, extensibility, and avoids vendor lock-in. We like to use Kubernetes as an example because it provides an API that does all of these things:

Flexible? Check - it can be installed quickly and easily on many platforms and devices

Extensible? Check - the Kubernetes APIs can be heavily extended to provide custom features to your users

Avoids Vendor Lock-in? Check - while you may use a vendor-managed version of Kubernetes, all vendors are required to pass the Open Source Kubernetes Conformance checks on a quarterly basis, meaning they cannot deviate from the standards-defined version of Kubernetes as published by the community if they want to advertise the product as managed Kubernetes. This means you, the platform user, can easily shift your workloads to another managed Kubernetes on somebody else's cloud with an acceptable level of effort.

### **2.2.2 Cloud Native Technologies**

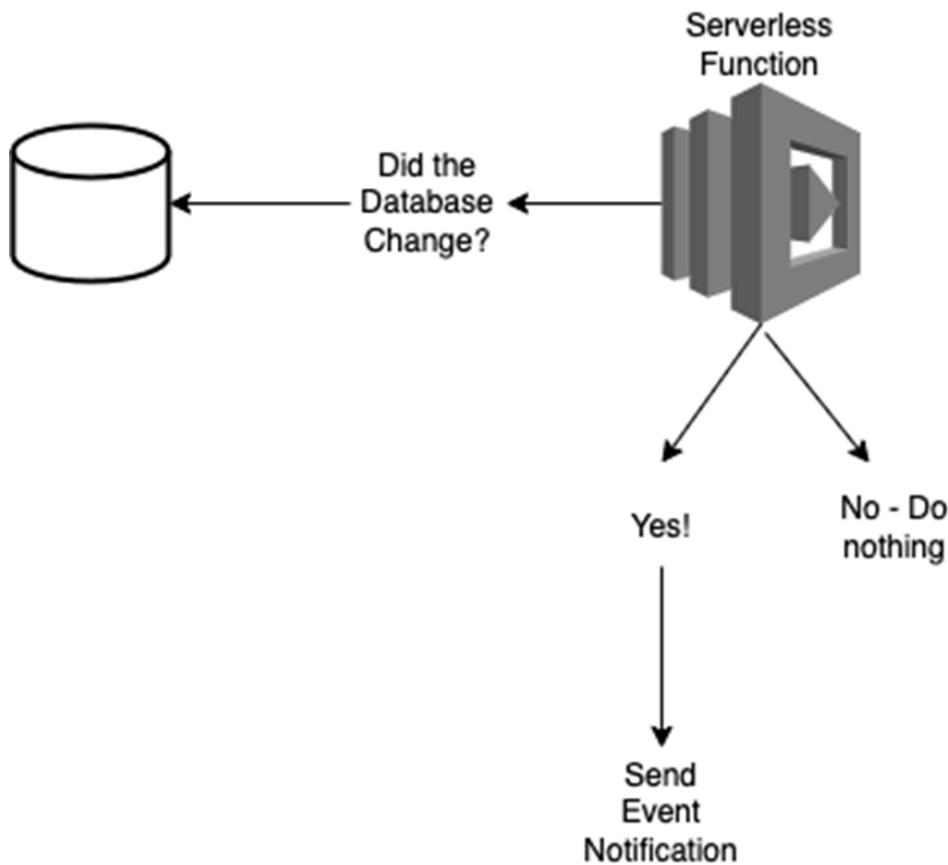
Cloud Native technologies give us a very powerful approach to building Engineering Platforms. The key difference between the ability to build platforms now versus 10 years ago is that vendors now agree on API design and specification, as we discussed in 2.3.1. Remember that we said each major cloud vendor implements and conforms to the upstream releases of Kubernetes. They must run conformance tests to certify and prove to the community that they offer a validated project version. This, coupled with the extensible API model of Kubernetes, provides a unique foundation for building engineering platforms that are not limited by the vendor.

Before this evolution, the ability to provide a platform offering was largely limited by the feature set of the tools purchased and the degree to which those tools provided an API. Even when they did, the platform team would largely feel “locked in” once they invested their time in providing an extended ecosystem around purchased tools.

The evolution of Cloud Native, however, has driven even proprietary tooling to take a different approach to extensibility. We now see foundational tools, even from a specific vendor, that help provide flexibility and avoid some level of vendor lock-in. They do this by providing interfaces we can abstract and build on top of instead of consuming them point-and-click.

To understand how important governed extensibility can be, consider the tradeoffs of Serverless versus Managed Kubernetes as a case study. Serverless platforms are generally defined as vendor-provided and managed offerings that allow one to deploy lightweight functions (usually written in Javascript, Golang, Python, or Java) that do one or two things well and fast. This is a very powerful offering. It allows developers to quickly deploy a function to the cloud and see it work with relatively little effort.

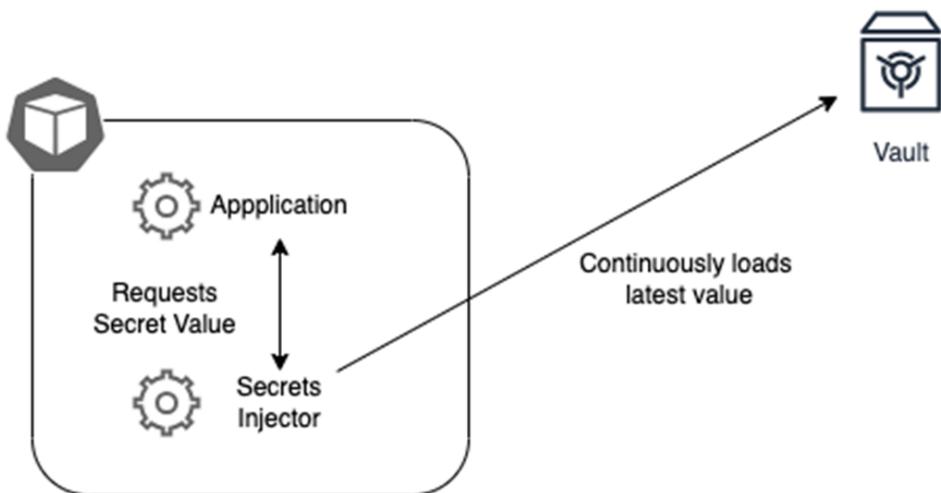
This figure shows an example of a good use of Serverless functions.



**Figure 2.8 shows an example decision tree for when to use serverless functions. Serverless functions are good for lightweight, simple workflows that serve a single purpose.**

When deciding whether to use serverless functions, a good target is a lightweight, simple workflow that serves a single purpose and is not expected to need the more complete capabilities of an engineering platform. That way, the function can execute quickly, ideally without any side effects on other parts of the system, allowing you to take the most advantage of cloud pricing models for this service.

Now, let's consider a few (not all) requirements of an engineering platform. We'll need secret injection, code signing verification, static code analysis verification, traffic routing, and identity verification, just to name a few.



**Figure 2.9 shows an example of a pod sidecar. This container runs a process alongside the application container to enhance functionality. In this example, a sidecar continuously loads a new secret value from a vault so that the application will always get the latest updates when secrets are requested.**

In the Kubernetes context, most of these requirements are relatively straightforward. We can use the Mutating Admission Controller API to automatically add capabilities to every service when deployed, such as adding sidecar processes for retrieving secrets or performing authentication and authorization decisions before the request is passed along to our primary workload. We can use the service-to-service networking model to provide routing and load balancing to services or use one of the open-source service meshes to provide more complex routing requirements and even circuit-breaking capabilities. We can use the Validating Admission Controller API to verify our code is signed at deployment time.

In the serverless context, these extensible APIs do not exist...by design! All those additional capabilities are the developer's direct responsibility and must be managed during the deployment.

The serverless platform is designed for simple, single-purpose functions. However, we often see companies trying to build more complex engineering platforms using these serverless functions. And they either buy off-the-shelf tools or reimplement the features that an open and extensible platform like Kubernetes already provides.

The important distinction to draw from this section is that when we talk about Cloud Native, especially in the context of Platform Engineering, we are talking about using tools and technologies that are extensible, flexible, avoid vendor lock-in (where possible), and integrate well into our platforms and environments.

### 2.2.3 Build vs Buy in Cloud Native

Build vs Buy is a problem as old as software. And that will probably never change. In Platform Engineering, remember that the goal of a successful platform is to provide a product offering to your users. Product offerings require the ability to change and evolve dynamically and rapidly. So when considering whether you should build or buy a tool for your engineering platform, ask yourself if that choice is going to give you the flexibility you need to provide new features to your users as well as the extensibility you need to continue to add new features after the initial adoption of this choice. Now, there is a danger that the company that built the product you chose will soon go out of business. This is yet another reason why you might want to build your platform product with the right level of composability and replaceability so that you can swap products out easily without a large-scale system redesign.

When looking at technologies for your engineering platform, we want you to go back to that quote from earlier and make the decision that will be easier to change later. When considering the serverless versus Kubernetes tradeoffs, which one will be easier to change and extend? Maybe the original answer is Serverless, but as your platform's needs grow and evolve, you quickly realize you need a more extensible platform to facilitate rapid growth and customization.

### 2.2.4 Exercise 2.3: Evaluate the vendor market

#### Exercise:

Research and find other platforms in the Cloud Native Ecosystem that meet the criteria in 2.2.1.

Hint: They don't necessarily have to be open source to meet these criteria!

## 2.3 Software-Defined Platform

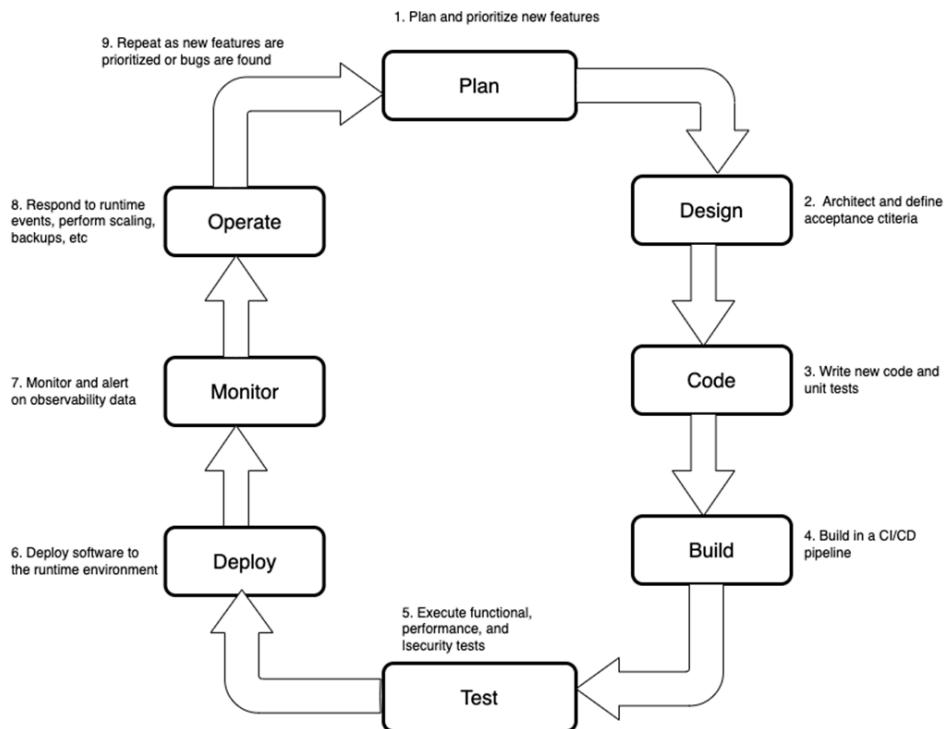
Understanding the foundational concepts of software development practices is crucial for building an effective engineering platform. This section will explain how to apply software engineering principles to increase the stability, scalability, extensibility, and maintainability of your platform, ensuring it delivers ongoing value to your organization.

When creating our platform, we must think about it like any other software product developed within the organization. In chapter 1, we learned about using product strategy and practices to start defining our backlog. Now we need to think about the software development practices we should use for the development, build, and release cycles for the platform. In our experience, many platform teams will start with engineers that have an operations background, because they have more infrastructure experience than most. In traditional operations roles, simply developing scripts or IaC modules and executing them manually have been generally accepted ways of working. As we move to platform engineering, we want to introduce more rigor into the SDLC process, using software engineering principles to increase the stability, scalability, extensibility and maintainability of what is developed. Some of these practices are probably already familiar to most experienced engineers.

### 2.3.1 Development Practices

Solid engineering practices help to develop and release repeatable and maintainable software. As we define those practices, it's helpful to create a document that the whole team agrees on so that we can be confident everyone will follow the standards in the same way from the start. This document can also be a team charter of sorts to make onboarding new team members easier in the future. There is an example of this type of document in the Chapter 2 folder of the GitHub repo for this book called "Team\_Charter.md".

### 2.3.2 The Platform SDLC



**Figure 2.10 A Base Development SDLC covering all stages from feature planning through running in production. This can be used as a standard across the team to increase code quality for all releases.**

Throughout this book, all code development done on an engineering platform follows the SDLC seen in figure 2.10. We will extend some of the sections of this as we move forward, but for now, this serves as a base SDLC process that we can document and expect the team to follow for every change made to the platform.

#### Plan

- Each feature will be planned to determine how it fits into the overall platform roadmap, what dependencies it may have, and how it should be prioritized.

## Design

- Before beginning development, we will design the feature with architectural diagrams and acceptance criteria. This is not to say that we expect the design to be final and unchangeable. We need to keep the ability to pivot as we learn new information.

## Code

- Once we have a design in place and know what the acceptance criteria are we can start coding including writing unit tests.

## Build -> Test -> Deploy

- All deployments will go through an automated pipeline that runs tests before deployment.

## Monitor

- All deployments are monitored according to monitoring requirements written during the plan phase

## Operate

- The platform team is assumed to operate autonomously, meaning that they will also support what they deploy in production.

## Repeat

- Cycles can occur in any part of this process, and because we are delivering the platform as a product, it is never considered “done”. Every release informs the next release as we learn how the system is used and we fix bugs.

One thing we should note here is that to be successful, we need to have a focus on Observability (Monitor and Operate) for the entire development lifecycle.

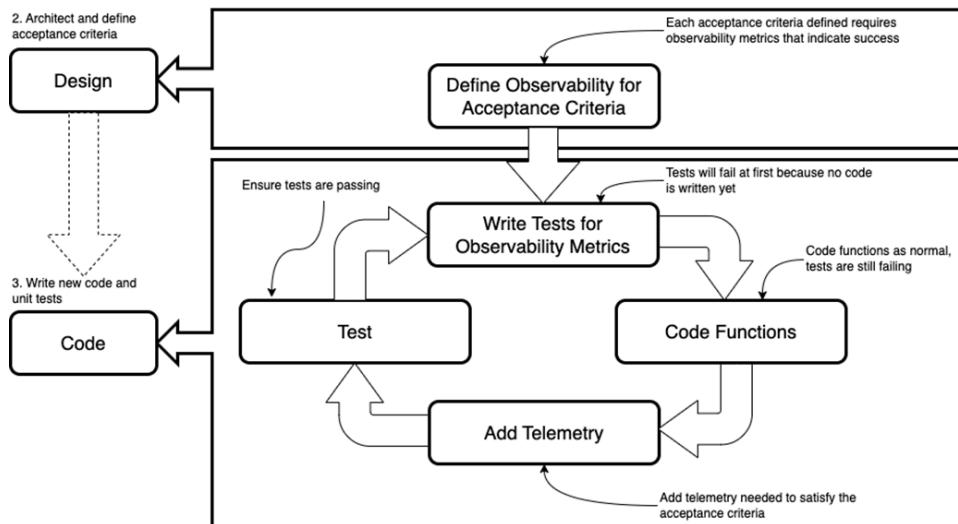
### 2.3.3 Observability-Driven Development

Once your engineering platform is in place and developers start using it, your team is releasing new changes regularly using all of the software practices and the full SDLC described above, and you can see that all tests are passing. However, one engineering team reports that some of their services have intermittent communication issues. They've tried to diagnose the problem by looking for unhandled exceptions and HTTP error codes and can't find any issues in their software, so they've become convinced it's a problem with the platform they can't see and want to file a bug report.

Your team spends a couple of hours trying to track down the problem as well by looking at the metrics being produced from the runtime and can't find anything either, so you're convinced it's a problem with the other team's software and try to work with them to find it. Eventually, someone on the team suggests activating more granular network logs and trying to reproduce the problem once the new logging is in place. Analysis shows that a network policy in the base platform was throttling traffic because the throughput generated by the new services exceeded the threshold for a new DDOS policy that was only expected to apply to external sources. A simple fix to the configuration is made and deployed, resolving the issue.

The team lost a full day of work due to the issue. They spent three hours unsuccessfully debugging, reported the bug, and waited an hour for the platform team. The platform team spent 2 hours checking logs and another hour debugging the new services. Once network logs were activated, it took an hour to parse them, and the fix took just 10 minutes to code and deploy.

All of this could be mitigated with Observability-Driven Development (ODD). ODD is a technique that guides the definition and development of new software by determining what observability data (and possibly alerting) is needed to ensure that a feature is not only working, but working *correctly* and delivering the intended results. To do this, the design and code phase of the SDLC described above can be extended with a new cycle:



**Figure 2.11 An Expanded SDLC that includes Observability-Driven Development (ODD) Practices. ODD can help ensure that a delivered feature is not only working but also working correctly and returning the expected value to the organization and its customers.**

## EXTENDING THE DESIGN PHASE

We begin by including a step in the design phase to figure out which observability data can prove that the features we release are not only functioning but also performing well and delivering the expected value.. As an example, imagine we want to add a new feature to the platform based on the scenario we went through at the beginning of this section:

- Protect the platform from external DOS attacks by throttling traffic above 1000 requests per second (RPS) to a single service from the same source

We need to determine what observability data we need to make sure this is working as intended by stopping traffic above the threshold. Still, we also need to think about what we need to know to determine it's functioning correctly and give us the value we want, i.e. platform protection from malicious attacks. Data to know if it's working correctly should be pretty straightforward (you may think of other data points)

- RPS arriving at a destination address from a single source

If the feature is working as intended this should always be at most 1000. Next, think about what data we need to show that this returns the intended value, protecting the platform from external DOS attacks. (Note that most network hardening will want to also protect against insider threats, but we won't consider that for this example). To show this, we want some additional data:

- Source of the request
- Destination of the request
- An indicator of whether this is an internal source
- How often requests are blocked from a given source
- Because this is a system security feature, alerts are defined to notify the platform team that DOS protection has been fired so that it can quickly be determined if further action needs to be taken

With these data points defined, we can now begin the development cycle with the following steps:

- Tests are written to show that the observability data needed is being produced
- Functional code is written, along with unit tests, to show the correctness
- Logging and telemetry are added until the observability tests pass

At a high level, the observability-driven design principles are the following.

1. **Instrumentation as Code:** Embed telemetry (metrics, logs, traces) directly into the system, ensuring observability is integrated with the development process.
2. **Contextual Data and Tracing:** Collect rich, contextual data and implement distributed tracing to track system behavior across services, enabling quick identification of issues in complex architectures.
3. **Actionable Metrics and Logs:** Define key, actionable metrics, and structured logging that provide insights into system health, performance, and errors, making monitoring and troubleshooting efficient.
4. **Automated Alerts and Self-Service Monitoring:** Enable teams to configure monitoring, dashboards, and proactive alerts, empowering quick responses to issues without centralized dependency.
5. **Continuous Improvement via Feedback Loops:** Use observability data to focus on root causes, drive continuous system improvement, and balance innovation with reliability through error budgets.

If you know about test-driven development, this might seem familiar because it follows a similar approach. The twist here is that we go beyond just writing tests to check if things work. We also ensure that the data itself can confirm everything is correct and that we're actually getting the value we expect.

If we deliver this feature with the observability data we defined and if the scenario we started with were to happen, the platform team would get an alert right away that traffic was being blocked. They could quickly see that the traffic was coming from an internal source and that something was misconfigured with the policy. It could be that the definition of internal addresses is wrong. In addition, because the team saw intermittent network failures and not persistent blocks, the time a detected source is blocked is likely too short. The platform team could now notify the application team that a problem with the platform was detected, and they're working on fixing it, saving hours of effort all around.

### **2.3.4 Exercise 2.4: Observability-Driven Design Requirements**

Practice defining the observability data needed for a feature using ODD principles, listed in the previous section by considering the following feature request:

*Allow teams to quickly expose a deployed service to the internet by creating an API function that will accept a service name and location and:*

- Create a DNS entry for the service at service-name.petech.com
- Only allow TLS-encrypted traffic on port 443
- Route traffic to the service location

Think about these two questions and define the observability data we might need to be able to show.

- Is this API function working correctly?
- Is this API returning the expected value to the team or organization?
- Do we need any alerts for this feature?

## **2.4 Evolutionary Platform Architecture**

As you begin to design your software defined platform, you may start to struggle with design choices. Is the X tool or Y tool better for security scans? Should the teams be managed in a centralized API or GitHub? Should we use an open-source tool or work with a vendor? Is using Infra-As-Code to create our services and resources appropriate, or should we use a managed infrastructure service? You could even think about the familiarity of the developers with architecture choices, and how difficult it could be to learn them.

All of these questions are good ones, and they have different answers in different contexts. The most important thing to remember is something we discussed in 2.3.1. When you have difficult technology decisions that can reach the same result, we should always pick the one that is easier to change in the future. This principle also applies to the architecture of our platform. When we're making design choices about our Engineering Platform, we can't possibly know how the platform will be used a year from now. So, all we can do is make design choices that are flexible and malleable.

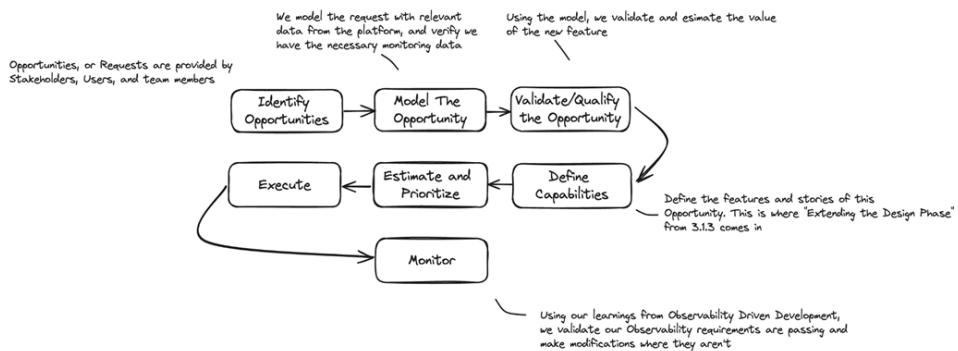
### 2.4.1 Backlog Management for Incremental Design

As we introduce the concepts of platform engineering to the business, the ideas start flowing. Everyone in Product, Ops, Marketing, and leadership has an idea for the platform that we "simply must do". It's good to capture those high-level ideas as large buckets, ideas, or epics if you use some form of iteration management tooling. But, as we capture the backlog and ideas from our stakeholders and leaders, the team begins to get stuck. It's hard to imagine if we'll need a platform UI for the dev teams because we don't have any dev teams using the platform yet! We might be able to get some nods of approval when we bring up the idea of a CLI to talk to the APIs, but until we get users hands-on, it's just an assumption.

"In practice, master plans fail—because they create totalitarian order, not organic order. They are too rigid; they cannot easily adapt to the natural and unpredictable changes that inevitably arise in the life of a community. As these changes occur...the master plan becomes obsolete."

*As quoted in Domain Driven Design, pp 497, Evans*

We don't want to implement massively complex features that nobody ends up using. Getting the platform to a minimum viable usability allows us to quickly switch gears into a process of software development that enables the platform to grow based on the real needs of its users.



**Figure 2.12 Platform Development Iteration Flow. Using a flow during iteration will focus development on how the short-term lifecycle fits into the bigger picture.**

When you try to look too far ahead into your platform's life, you forget to focus on the short-term lifecycle of the work that needs to be done in the immediate future. Instead of concentrating on the work that needs to be done within the next 1-4 weeks, you can ensure that the work being done, and about to be done, is identified, modeled, validated, and properly defined. This provides less confusing and more consistent execution and measurement of your outcomes, allowing you to pivot and evolve your platform much more fluidly.

We also highlight again the importance of monitoring data in this workflow. Note how we refer back to our learnings from observability driven development at multiple stages in this process. At step 2, "Model the Opportunity," we are modeling the requested change and validating that we have the necessary data to do so and the necessary data to build the feature. And as we proceed to define the new capability, we are making sure to build changes for new data points so that our platform's new capability can be monitored and alerted on properly. Going back to our example of the traffic blocking alerts, if we don't account for making sure we are collecting network data during our "Define" phase, we'll be missing the necessary tests and infrastructure to capture these data points, rendering our tests useless.

Let's say a stakeholder comes in and says "My team needs a user interface to push deployments to the platform." Our PE Tech team has some debate about this once it's time to prioritize it for the platform. Some members think that very few engineers will use it.

Following our incremental change process, let's see if we can answer some of these debates. From stage 1, we have already identified the opportunity, but let's put it into words:

*As a developer, I need a User Interface to push deployments to the production cluster.*

Great, now let's Model and then Validate it. Given that this is a user experience feature and not something we can directly validate with monitoring data, we'll need to model this feature with data given to us by our users. As a team, we assume that most of the developers will prefer to use the CLI in a deployment pipeline. But we need data to model this assumption and validate it. So we'll want to start with a small 2-question User Survey and send it out to the engineering group:

Question 1 (select multiple): In your day-to-day role, do you prefer to use:

1. CLIs
2. APIs
3. GUIs
4. All of the Above
5. Other

Question 2: If given a choice between the current CLI in a pipeline and a click-to-deploy GUI, which would you deploy to the platform with?

1. CLI
2. GUI

We send this out as a simple Slackbot 2-part question, making it as easy for our engineers to answer it and return to their jobs. We don't want to inundate our users with lengthy or wordy surveys; it's best to model these new capabilities with questions that are most likely to get as many responses as possible. Keeping it short ensures we'll at least get a majority (50% or higher) response rate.

Our results come in, and we get:

50 overall responses

**Table 2.1 Shows the responses to the two questions that were posed above**

	Q1	Q2
Responses	35 CLIs, 30 APIs, 25 GUIs 10 All of the Above, 5 other	30 CLIs, 20 GUIs

Surprisingly, there are more users than we expected who want to use a GUI to deploy to the platform! Had we just used our assumptions and not modeled and validated them, we never would have developed this feature, leaving 40% of our users using a set of features they would prefer to do in a different interface.

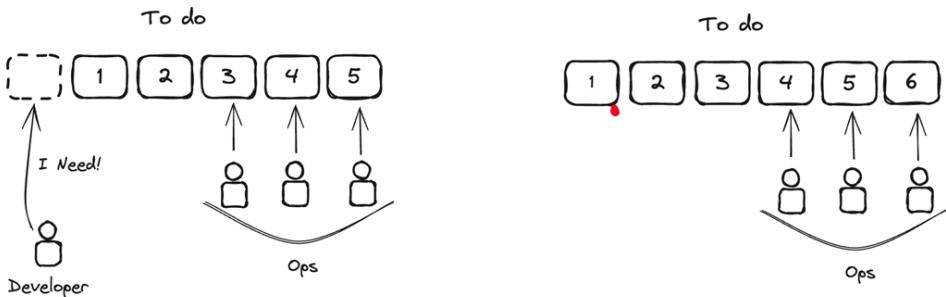
This model and validation outline the importance of due diligence regarding new features and opportunities for the platform. We also reinforced the importance of simplicity. In our experience building platforms, we've noticed that when we start complex dialogues, open conversations, or lengthy RFCs, the engagement is a meager overall percentage of the developer population. It's important to remember that every individual in the engineering organization has a role to perform and daily duties to focus on. By making our survey short, sweet, and in the medium our teams are already using (say a 2 part in-channel slack question) we reach a much broader audience and get a proper validation of our assumptions.

Now that we know we will build this GUI for our users, what sort of data do we need to think about in our Define stage? Remember again our learnings from Extending the Design Phase; we want to think about what data we require to drive functional automated tests for the platform. Given this is a UI for our development teams, we'll want to ensure they are having an excellent overall experience, including latency, low error percentages, and fast page load times. We'll also want to write tests and automation that ensure we are properly collecting this data; for example, error percentages could be perceived as low if we aren't collecting the proper stream of errors.

As we move on to the following stages of the iteration, including execution and monitoring, we want to look for a similar percentage of adoption as we saw in our survey. If we had 40% of users saying they would like to use the GUI of our platform, this is our target adoption percentage as we roll out the new feature. We can write observability data to track this, and if we don't meet it, we can do follow-ups with our users to find out what's going on, making changes to fit their needs and to meet our acceptable adoption percentage.

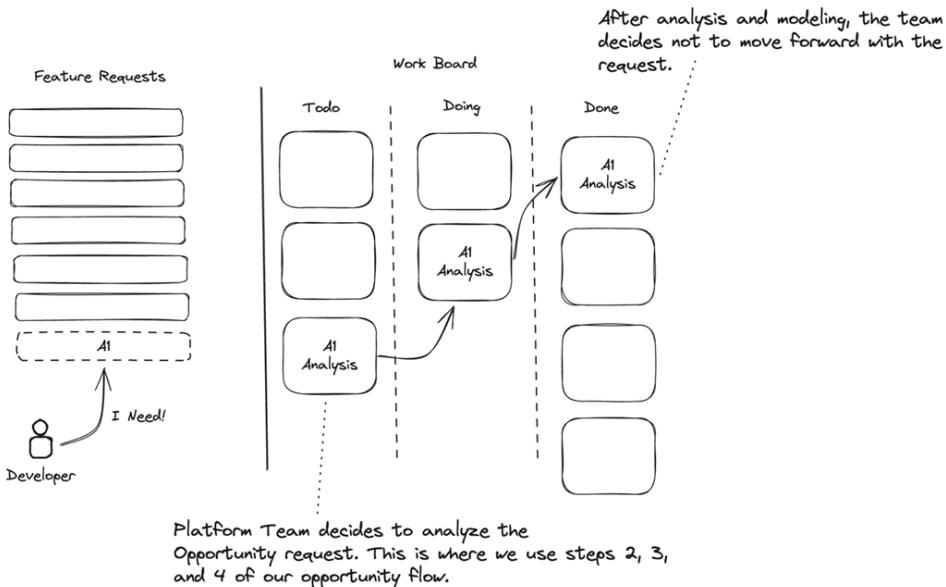
#### **2.4.2 Capturing User Interactions and Feedback**

At this point, you might be wondering how to identify the opportunities you need to build a platform. To answer that, there are several ways to measure the usage of our platform, discover what our customers need, and elicit feedback directly.



**Figure 2.13 A standard Operations Team Ticket Queue. Developer Requests are immediately placed into the Todo pile.**

The first method is the most direct and obvious; as a centralized team, you will probably have a ticket system. This fact is unavoidable in most organizations. However, as a product team, your ticket system is treated quite differently from the standard operations team. In the typical operations workflow, it is assumed that when a ticket request is put in, there is an SLA on when it will be completed. But...there's an even bigger assumption we just glossed over. And that's the assumption that it will be done at all! Think about a DevOps or Identity team. When tickets come into their queue, it's assumed that most of these requests will be done at some point. This is not the case in the Product operating model of building our platform, because not all these requests will get prioritized. This highlights the problem with assuming that "DevOps" is a team. As we mentioned in Chapter 01, DevOps should be a culture.



**Figure 2.14 The Platform Development Ticket Queue. These get treated as product feature requests, to be analyzed, accepted and prioritized.**

When building an Engineering Platform and using the product operating model, it's important to remember that all requests, except for bugs and outages, are treated as product requests. This means that the team needs to carefully review and analyze each one. Customers of the platform have chosen to use the platform product, and that means they cannot expect to demand features be made, and certainly not with an SLA! If teams outside of the platform team were able to demand new changes all the time with an SLA, then our Platform team would slowly decay into being only a DevOps team, and it would lose focus on the self-service features that make our platform a functional product.

This doesn't have to mean that you don't need a request queue though. In fact, a queue can turn into your platform backlog! By applying a bit of marketing, instead of calling it a request or demand queue, we might want to call it a Platform "Idea" queue or Platform "Feature Request" queue. By changing the wording, we change its meaning, and teams will understand that requests can (and will) be denied if they don't fit within the Platform as determined by the product team building it.

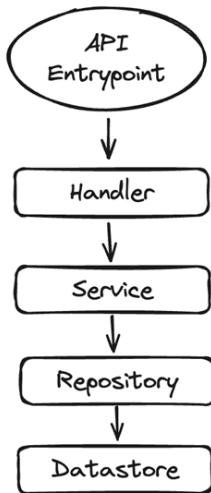
So, how else might we capture feedback and new needs of the platform? As you are building the platform at PETech, you realize you need monitoring and observability tools for the customers deploying applications to your platform. You may not realize that you, the Platform team, need those same tools. Automated measurement of the platform's usage is a key indicator of how the platform is being used, helping us to know which changes are liked and what new features we should prioritize. We'll talk much more about measurement in Chapter 4.

Lastly, you should get feedback from your customer base directly. There are numerous methods to accomplish this. When providing CLIs to your customers, you can include anonymous metrics gathering. You can send out surveys and conduct 1-1 interviews to find out what features people like, don't like, and don't have. It's important to come back and gather this type of data from your users regularly, and connect with them. Weekly or bi-weekly demos of new platform features help build engagement, trust, and interaction with the platform's customers. We consider this to be an invaluable component of the platform development process because there is no better way to make your product better - by getting feedback on what you have built so far.

### **2.4.3 Architectural Fitness Functions for an Engineering Platform**

As we are defining the APIs of the platform, the topic of databases comes up. Many of the APIs we will create for the Control Plane of our platform will need to store their state in a resilient database. One of the senior developers on our Platform Team at PE Tech points out that our platform will support many regions, maybe even a Global topology where developers will be interacting with our control plane from multiple continents. So our control plane must be highly available, fast, and replicated across many regions. So as a team you start thinking about globally available database services from your cloud provider. But, another team member then points out that we also need an easy to use development experience for platform engineers, and a highly distributed global database could make the local development experience very complex. And then another team member adds that while services like DynamoDB meet our requirements in AWS, we just bought another company and their entire infrastructure is on GCP. They've recently asked us to start exploring supporting the Engineering Platform on their cloud as well!

First, how do we reconcile all of these concerns? Lets return to the fact that we have a software defined platform. Good software design includes an architecture that decouples hard dependencies (like databases) and allows for change over time. To handle all the different data needs, we've decided to separate the database details from our platform's APIs. We're setting up a service layer and a repository layer. The repository layer will use a Datastore interface, which can work with various database technologies. As long as these databases use the same functions, we won't need to change the code in the repository or service layers at all.



### Example, Teams API

```
// Teams API
teamDS := datastore.NewPostgresDatastore("team")
teamRepo := repository.NewTeamsRepo(teamDS)
teamService := service.NewTeamService(teamRepo)
teamHandler := handler.NewTeamHandler(teamService)
```

**Figure 2.15 An example of defining the necessary APIs for a Datastore in an engineering platform (Teams API)**

You might have heard about Fitness Functions before. They're well-explained in many books. Simply put, an Architecture Fitness Function is a tool that helps objectively measure how well certain aspects of a software's architecture are performing. This concept is neatly summed up in "Fundamentals of Software Architecture" by Richards and Ford.

Then, to ensure we always meet this pattern, and keep these layers decoupled, we would write a **fitness function** that verifies the service layer only ever imports the repository layer, and the repository layer only ever imports an implemented Datastore. We'd write another fitness function that ensures all of our Datastore implementations adhere to the standard Datastore interface. These Fitness functions ensure our control plane api logic doesn't ever change when we decide to implement a new database, be it local to one developer's computer or globally distributed.

You can think of them as a sort of Unit test that asserts the architectural patterns and decisions remain intact as you are making changes.

```

// Teams API
teamDS := datastore.NewPostgresDatastore("team")
teamRepo := repository.NewTeamsRepo(teamDS)
teamService := service.NewTeamService(teamRepo)
teamHandler := handler.NewTeamHandler(teamService)
...
// Fitness Functions
// pseudocode - Modified example from Richards and Ford Page 87
layeredArchitecture()
    .layer("Datastore").definedBy(..datastore..")
    .layer("Repository").definedBy(..repository..")
    .layer("Service").definedBy(..service..")
    .layer("Handler").definedBy(..handler..")

    .assertLayer("Handler").notAccessibleByAnyLayer()
    .assertLayer("Service").mayOnlyBeAccessedByLayer("Handler")
    .assertLayer("Repository").mayOnlyBeAccessedByLayer("Service")
    .assertLayer("Datastore").mayOnlyBeAccessedByLayer("Datastore")

```

Here we can see that each layer is only consumed by the next, and we enforce this with a fitness function. This makes sure that down the line if we try to skip creating a datastore layer for a new database (choosing instead to call datastore functions from our service layer) our Fitness function will fail, stating that we must use the Repository Layer to interact with our new Datastore. You can see more examples of this pattern in action in our reference in the Github repository for the book.

Another thought that may cross your mind for your platform at PETech is this Fitness Function practice feels a lot like the sort of thing the developers have to do for their applications, writing tests! And you would be right. Remember, at the start of this chapter, we said that Platform Products are also software to be developed using a software SDLC, and to build a scalable and successful engineering platform, we have to treat it with Software Principles. This includes fundamental architectural principles, like Fitness Functions, and writing tests that we continuously verify and trust.

Take a look at the repository for Chapter 2 to see more examples of engineering platform fitness functions.

#### **2.4.4 Exercise 2.5: Fitness Functions**

At PETech, while we are going to build the Engineering Platform on AWS, we know that the merger with AllTech is pending completion. AllTech is 100% on Google Cloud, and they don't even have an AWS account. We know that when we build the platform for PETech, we have to focus on our immediate customers but make architectural decisions that allow us to change and adapt the platform, such as potentially for other clouds in the future. One area of high importance is our custom Platform APIs.

How might we write a fitness function that ensures our platform APIs are implemented in a cloud-agnostic way?

1. Using the sample API provided in the (C3 Repo)[Todo, Link] - Write a fitness function that ensures our API keeps cloud-specific features and operations in isolated interfaces that don't affect our service logic
2. How might we expand this fitness function to work for all of our Platform APIs, not just this one?

After some debate amongst the team, we've decided that test-driven development is a rule we want to adopt and use across all of our platform's custom software.

1. Write an Architectural Decision Record that captures this decision. Include reasons, alternatives, and details.
2. Write a fitness function that will fail if someone checks in a new Service Layer without tests associated with it.

As we've seen throughout this chapter, Observability and Monitoring Data are at the core of every decision we make. Consider how we might write ADRs and Fitness Functions that capture this.

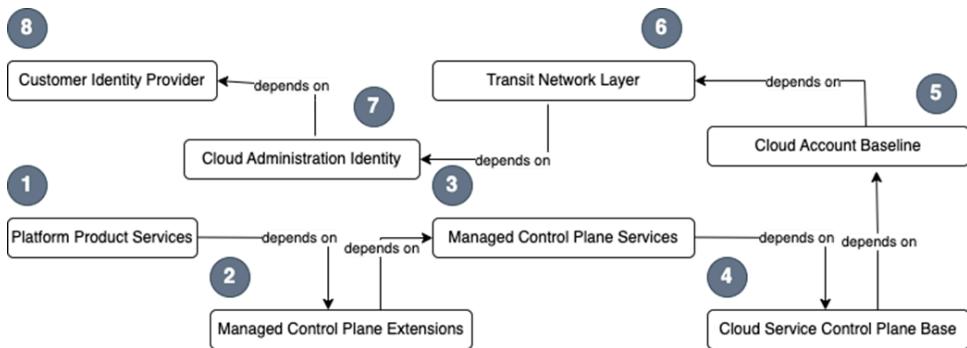
1. How might we write a fitness function that would fail if a new API Feature gets checked in without any monitors? Feel free to use a specific observability tool to write your answer and then compare it against the answer in the back for similarities.
2. Consider how a data-driven approach might change the dynamics of the team's interactions with other stakeholders and executives at the company. What tactics can you use to debate the merits of a new feature request using our ADRs, Fitness Functions, and observation-driven decision-making? Consider how these techniques remove emotions and assumptions from these sorts of debates.

## 2.5 Domain Driven Platform Design

Recall in 3.2 our quote from Eric Evans: "**master plans fail.**" Assuming at PETech we accept this as true, what sort of high-level plans can we make that help guide us and nurture our platform, ensuring it moves in the right direction? One area that helps teams align on a common vision and shared understanding is to adopt the principles of domain-driven design. Popularized by Eric Evans (in the book named the same) in 2004, Domain Driven Design has given engineers and business leaders a common language and framework for designing and planning software build to implementation.

At PETech, you were already familiar with domain-driven design. Your CTO handed out the book to everyone at the company four years ago during the "transformation" and popped around the office with random quizzes to ensure everyone read it. But in thinking about platforms, you start to wonder, "What are the domains of an engineering platform?"

Recall that in 1.8, we talked about the eight domains of an engineering platform.



**Figure 2.16 Expanded View of the Platform Engineering Mental Model first introduced in Chapter 1 is reproduced here for easy reference.**

If you need to review each of the 8 domains (right side column) refer back to section 1.8 of Chapter 1.

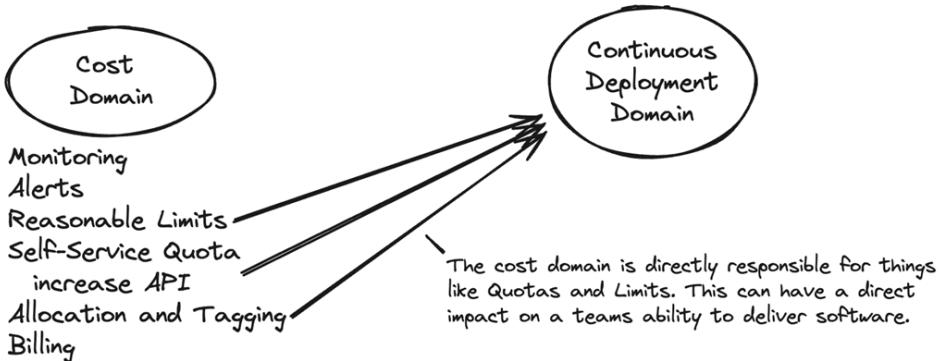
By using these domains in our software practices, as described in this chapter, they become part of our planning and incremental design. Let's give some examples.

Imagine that you are having a strategy session at PETech, and one of the platform stakeholders has stated that things are getting too expensive. One engineer states that we should use cost-tracking software, like IBM Apptio. Another suggests we tag and annotate our tooling and create automated alerts. A third engineer says we should track and implement price automation approvals for teams.

The tricky part about these wide-ranging discussions is that everyone uses different terms and concepts. Let's focus on the issue of cost. What can we actually do within our platform to manage costs? How can we set up models to track and improve our cost management? And what exactly are our goals related to controlling costs?

By flipping the conversation to be domain-bounded, we move away from proposing random solutions using disconnected terms and concepts to focusing on what the problems are, and how we will measure them, and putting together ideas using our shared understanding of the problem. We might write the following (simplified) Architectural Decision Record:

*The cost domain is important to our engineering platform design. However, the domain of Continuous Deployment requires that we provide our developers with an experience that does not create friction in their deployment process. So we know we must provide cost solutions that fit our platform's other domains. We propose the following goal: 80% of all development teams leverage cost automation to give them notifications when a workload is underutilized.*



**Figure 2.17 An example illustration of an ADR for cost domain aligned to the continuous deployment domain**

What's important about this ADR, is the focus on how the Cost domain is interacting with the Continuous Deployment domain. Instead of just focusing on the reduction of cost, we focus on the reduction of cost within the boundaries and goals of our platform, realizing that there is value lost in creating more friction for development teams. While often difficult to correlate 1-1 to dollars, creating friction in the development process will slow down our teams ability to deliver new features to our customers, in the long-term resulting in a loss of revenue.

By focusing our conversation on the domains of the platform, as opposed to focusing on a random problem, we came to a solution that fits with the other domains of our platform and doesn't break the relationships we have with them.

### 2.5.1 Exercise 2.6: Writing a Domain-specific ADR

Using what we learned so far, write an ADR for our Compliance domain, documenting Compliance at the Point of Change with, giving the context of our compliance domain and how it interrelates with the Continuous Deployment, Security, and Self-service domains.

## 2.6 Summary

- A platform delivery model will require the concept of platform product ownership with an assigned role
- Bypassing careful planning with off-the-shelf solutions might not achieve the desired results in platform engineering.
- Every investment in a platform capability requires a clear understanding of the value delivered within the defined scope.
- Your existing platform strategy might prioritize starter kits followed by Kubernetes and observability, but stakeholders, prioritization, and user research need further consideration.
- The idea of a software defined platform introduces more rigor into the SDLC process, using software engineering principles to increase the product's stability, scalability, extensibility and maintainability.

- Building an engineering platform requires flexibility and avoids vendor lock-in. Cloud-native technologies like Kubernetes provide standardized APIs for building extensible platforms that are easy to change later. Serverless functions are suitable for single-purpose tasks but need more extensibility when building platforms. When choosing tools, prioritize flexibility and extensibility to adapt to evolving needs.
- Delivering platforms as products with a well-thought-out delivery model and technical product ownership as its backbone are critical for overall success.
- The product delivery model assumes continuous evolution based on customer use and changing needs.
- Customers should actively use and receive value from the platform.
- Stakeholders do have an interest in the platform but are not users themselves.
- Stakeholders should support the platform by providing service interfaces or well-defined outcome standards.
- The entire product scope should be assessed for delivery quality and success against the product strategy.
- Features and capabilities should be delivered as service interfaces first.
- Architectural and product experience decisions should be based on actual customer usage.
- Start with a Minimum Viable Product (MVP) and evolve based on customer feedback. Early adopters provide valuable insights for continuous improvement.

### **2.6.1 Chapter Footnotes**

[2.4b]

1. Gall, John. 2002. *The Systems Bible*, Third Edition. Walker, Minn. General Systemantics Press.

[2.4.3c]

2. Engeström, Yrjö. *Learning by Expanding. An Activity – Theoretical Approach to Developmental Research*. Helsinki: Orienta konsultit.

[11] <https://oauth.net/2/>

# ***3 Measuring your way to Platform Engineering Success***

## **This chapter covers**

- The importance of measuring key aspects of platform engineering, including platform value models and metrics-driven investments that support the platform's overall success.
- What are the models to consider when setting up the engineering platform scaffolding, with a focus on the product delivery model introduced in the previous chapter, now expanded to cover the entire path-to-production.
- Cognitive load management, its impact on various stakeholders in the SDLC, and strategies for reducing it to improve developer efficiency and experience.
- Organizational changes required for implementing platform engineering and establishing key performance indicators (KPIs) to measure progress and success.

So far in your journey at PETech, you have established a vision and strategy, decided that you have a business need for building an engineering platform, and have put together the foundational engineering practices to create it. You have also looked at software-defined platforms and the cloud-native approach, which simplifies the process. The last critical element of your planning process comes - measuring what you will build. The final chapter of part one will round out all the pieces required to ensure you are setting up the platform engineering journey to succeed not just in the short term but for scaling.

Measuring what you are going to build has several aspects. At PETech, we saw that the leadership had understood the vision and worked with you to establish a strategy. During this time, your conversations with the CTO and the SVP of engineering have given you many insights into mapping this strategy to their overall business strategy. Then, as part of establishing an operating model, you worked with the product managers of each product domain and created the prioritized set of capabilities required to build the engineering platform. So far, so good. You are now ready to produce the capabilities. But wait, there is one problem. How do you know that what you are building meets the success criteria? What exactly are the success criteria here? Is it just the revenue maximization? What if you maximize the potential revenues at the cost of employee morale? What are the specific approaches that you should consistently consider in this journey?

These are all the questions that all stakeholders at PETech can answer as we go through this chapter. By learning from PETech, you will also be able to replicate its successes in your organization.

## **3.1 Why do you need to measure?**

Understanding how to measure and improve platform engineering practices is crucial for ensuring your efforts lead to tangible, positive results. This section will explore the importance of measurement in platform engineering and guide you on what to measure to demonstrate the value of your investments and drive continuous improvement.

Success depends on an objective and subjective assessment of what you are doing. Primarily, without measuring your ability to answer the questions you are trying to solve, you are, at best, suboptimal. By showing results, you can demonstrate the value of your investments. This value measurement is what we discuss in detail in the next section. But there is a lot more to it than simply measuring value. How about improving everything that you are doing? Continuous improvement is a core tenet of every engineering practice, and improvements require measurement. Platform engineering activities are no different.

### **3.1.1 What do you measure in platform engineering?**

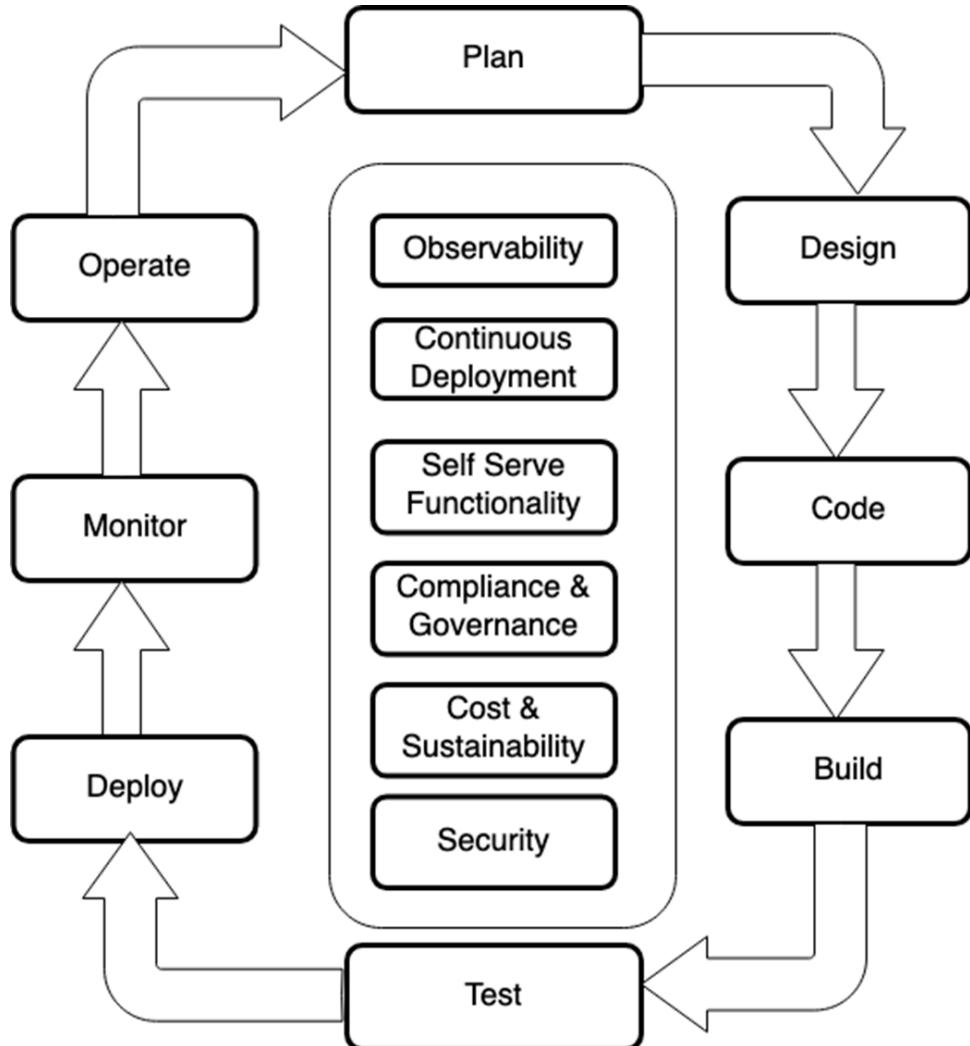
As you think about why measurement is important and explain this to your team at PETech, the next question to tackle is: what exactly should you measure? Should it be at the domain level, as discussed in Chapter One, or something else?

It's important to make sure whatever you decide to measure is straightforward to identify and set criteria for. This can seem complicated, which is why it's a good idea to look at your Software Development Life Cycle (SDLC) process to figure out what to measure. However, focusing only on the SDLC steps might make you overlook some details because you're seeing things from a very high level.

A better approach is to combine elements of the SDLC with the domains we talked about earlier. This way, you'll get a complete picture of what needs to be measured.

### 3.1.2 Mapping measurement to Core platform principles

Revisiting the discussion around the six core principles introduced in Chapter One and path-to-production analysis in the next section, identifying the areas of measurement around each of the six principles is an ideal way to start measuring the success of the engineering platform. Look at the illustration below. The lifecycle is a continuous process of improvement that is based on the feedback from the previous steps in the lifecycle.



**Figure 3.1** A high-level view of the SDLC in the context of platform principles. This figure indicates the fact that each of the platform principles is applicable for the steps in the SDLC.

As you look at the principles, there are several metrics you should track as you prepare for success.

**Total Cost of Ownership (TCO)** in software product development refers to the comprehensive assessment of all costs involved in acquiring, deploying, operating, and maintaining the product throughout its lifecycle. TCO includes direct costs such as licensing software or its use, development, and infrastructure and indirect costs like training, support, and downtime. Critical elements of TCO cover acquisition, implementation, operational, and maintenance costs and productivity losses. Understanding TCO allows organizations to make informed decisions by considering initial investments and long-term management expenses.

Here is a comprehensive list of activities to track

1. **Cost models for cloud infrastructure provisioning and usage.** While infrastructure provisioning may not always be on the cloud, standardizing on the baselines requires a set of services that can provide a relative view of infrastructure availability. Public cloud providers do a great job at this.
2. **Licensing models and agreements around the third-party tools.** Third-party tools can be SaaS, PaaS, or IaaS, depending on the organization's needs. Establishing standards, guardrails, and the acceptable numbers of the total cost of ownership (TCO) will include not just the licensing costs but also maintenance activities such as administration, installation, and upgrades, as well as the time saved by using these third-party tools.
3. The planning process should include planning for the capabilities to be built as part of a shared services stack. These should start with a **buy vs build analysis** that will again consider the total cost of ownership. Engineering platform capabilities that will look like a fascinating solution implementation may be better candidates to build at times if you can buy components of it and integrate them well with yet another basic tenet of platforms: replaceability. Just like improving the code base with changing client requirements, you will also see drastic improvements in the capabilities of the tools you might buy. Instead of having a vendor lock-in during such a time, which we see in every organization, by using a composable architecture, you will find it easier to replace them en masse without significant retraining, redesign, or re-platforming.
4. Establishing your **performance and scalability metrics** during the planning phase is critical. This is where your business strategy alignment, as discussed in Chapter 01, will come in handy. You need a clear vision of what your end users are looking for to decide what your platform should provide. For example, one of the requirements for PETech could be to deliver a targeted streaming service to millions of users around the globe every time one of their new features is released. This means that the metrics around specific high traffic across geographical considerations are important for cost-effectively delivering optimized content. That can be achieved through considerations around the capabilities of being able to scale to specific numbers that should be established around all aspects of your engineering platform.

5. **Security, governance, and compliance-related outcomes** are yet another one of the six core principles that are pervasive in every step of the SDLC as you map out your path-to-production. These include not only protocols to use but also measurements of the types of audits to be done and the vulnerabilities and risks involved. Modern application vulnerability scanning tools provide a significant amount of useful information about coverage, types of vulnerability, remediation time, and trends analysis. They also provide an overall score for the state of your system based on a set of algorithms that can give you and your application users a level of required confidence. Incorporating these into your engineering platforms and making them adaptive to the needs of the end-users will be an important consideration.
6. **Observability-related measurements** usually map to improved MTTD/MTTR (Mean Time To Detect / Mean Time to Respond) and the ability to generate actionable insights. These can be measured fairly easily by any observability tool you might be using. However, having a coherent strategy to expose this data across the SDLC is important as opposed to merely focusing on the application. For example, if you are instrumenting your applications, infrastructure, and business workflows, do you have measurements that assess the performance and scalability of the targeted entities? Would you be able to establish clear performance objectives?

DevOps Research and Assessment (DORA) initiative continuously identifies and categorizes various capabilities as you start your DevOps journey. This set of capabilities is also a good starting point for your platform engineering KPIs. As you build your engineering platform, your considerations typically include more than DORA recommends. Drawing inspiration from the former, we can look at a more comprehensive list of considerations if we map these capabilities to each of the eight steps identified in the SDLC diagram above. This mapping shows the most important capabilities for assessing your progress as you use path-to-production as the basis of engineering platform evolution.

The six principles introduced above should be viewed through the following three lenses.

1. What is the level of maturity (see section 3.3 below for a more detailed study on how to measure these)
2. How easy is this function for your organization?
3. What is the impact of doing this?

Each of these require addressing the cultural aspects of the organization, which we will discuss in section 3.3. Organizational culture, a transformational mindset, job satisfaction, a learning culture, and improving developer well-being summarize these aspects. We will keep returning to the ultimate goal of developer well-being, a.k.a. improving the developer experience throughout the rest of the chapter.

In the table below, we will first start by capturing the eight core tenets of each SDLC phase, starting with planning, designing, coding, and building the code.

**Table 3.1 Key capability categorization (Plan, Design, Code, Build)**

<u>Plan</u>	<u>Design</u>	<u>Code</u>	<u>Build</u>
Empowering teams	Loosely coupled architecture	Maintainability	Continuous integration
Flexible infrastructure	High cohesion	Trunk based development	Continuous delivery
Value Stream	Appropriate design patterns	Version control	IDE/Version control integrations
Working in small batches	SOLID principles	Readability	Environment configuration
Organizational culture	No overengineering	Consistency	Dependency management
	Intuitive design	Testability	Incremental building
User centricity	Single source of truth	Security	Artifact management
Project management	Consistency	Documentation	Package management

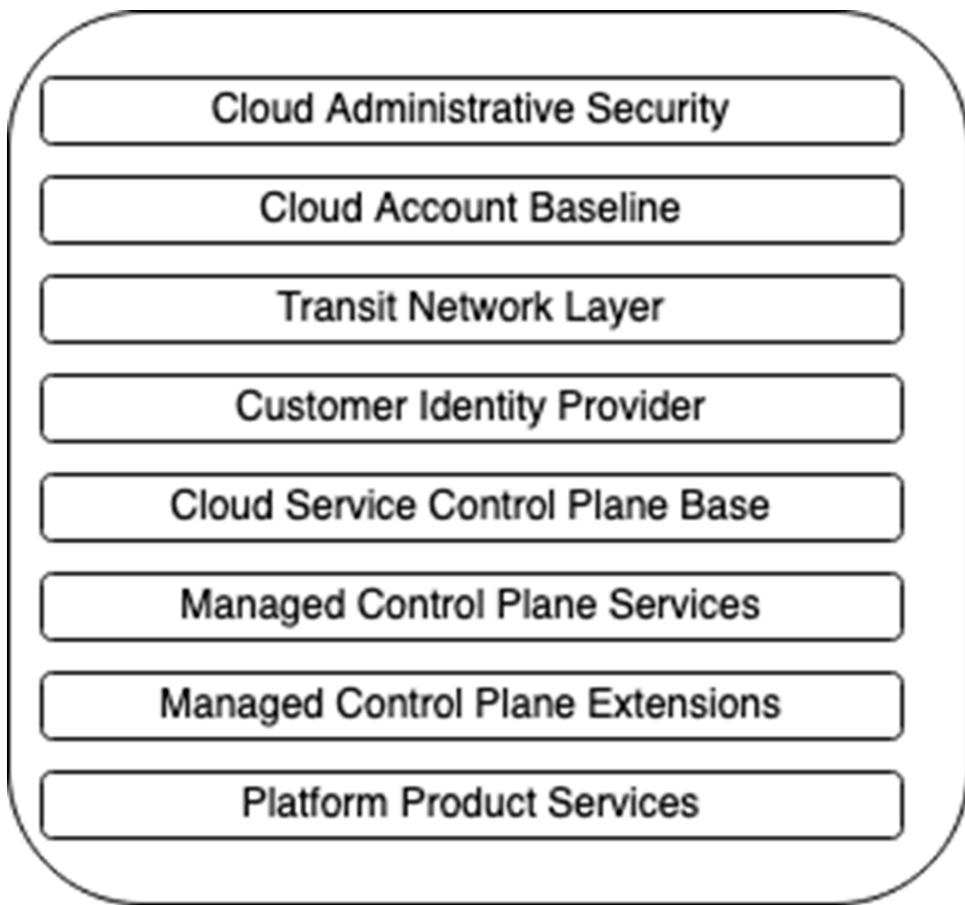
In the following table, we will continue the same exercise for the rest of the steps of the SDLC. Test the code, deploy it, monitor it, and operate it in production.

**Table 3.2 Key capability categorization (Test, Deploy, Monitor, Operate)**

<u>Test</u>	<u>Deploy</u>	<u>Monitor</u>	<u>Operate</u>
Test types	Deployment Automation	Monitoring strategy	Incident response and management
Test Automation	Change approvals	Observability strategy	Patch Management
Test Coverage	Scalability	Actionable insights	Failure Notification
Test Data Management	IaC integration	VSM visibility	Learning culture
Reporting & analytics	Config as code	Data Collection	Anomaly detection
Frameworks for technology stacks	Rollback	Analysis & Trend	Root Cause Analysis
Security	Feature flagging	Correlation	Database change management
Integration with CI/CD	Deployment Patterns (Blue/Green)	Dashboarding	Performance management

### 3.1.3 Mapping measurement to platform engineering domains

Let us now look at the engineering platform product domains we introduced in Chapter One. Each of the eight domains requires appropriate metrics to measure them individually, which will, in turn, tell you the areas of progress and improvements needed as you build your engineering platform.



**Figure 3.2 Engineering platform product domains. First introduced in Chapter 1 is reproduced here for demonstrating what should be measured for each of these domains**

These eight domains are crucial to understanding how to build your engineering platforms, whereas the principles referred to in the previous section focus on the attributes of what you build. Because of this, we have to look at both of these as we consider measuring the success of engineering platforms.

**Cloud administrative security (CAS):** This domain defines access control for the platform engineering team along with the security for service-to-service communication. There are several indicators to ensure the attributes in this domain are successful. They can be summarized as:

- Access control is the first thing that comes to mind in this domain. Ensuring that only the right people have access to the correct data and suitable access for these people done in a seamless, self-serve manner with automated compliance is the place for you to start. Your users should not need access to cloud resources directly, reserve this for PE team administration.

- Automated audits for regulatory practices should be implemented and measured relatively easily. These audits are done by comparing these practices against the baseline.
- Vulnerability identification and self-healing wherever possible while keeping track of the find/fix ratio
- Are you responding to the events fast enough, and are you resolving these? These metrics aren't that different from traditional MTTD/MTTR numbers, but purely in the context of CAS.

**Cloud account baselining (CAB):** As discussed in Chapter One, the ultimate goal for this domain is to avoid environmental corruption. Corruption happens because most providers use somewhat different terminology these days. We recommend focusing on measures around the following activities to ensure success in your efforts.

- Start by baselining these parameters with the expected behaviors around usage patterns, costs, traffic, or business-specific metrics. These baselines should be periodically revisited due to the dynamic nature of how cloud service providers update them.
- Use the data extensively to make your decisions dynamically. We recommend incorporating the baseline data and the actuals into your observability platform to understand the data better.
- Consider using predictive AI tools in this realm to generate the measurements. It would help if you used the ML/AI to improve the baselining's accuracy and make more efficient drift corrections.

**Transit network layer (TNL):** Our recommendation in Chapter One was to handle all the network connections for communication, setup, and management in a dedicated domain. The measurement considerations for this domain will be around the following:

- Reliability-related aspects such as packet loss, latency, DNS misconfigurations, and route flapping can all lead to unstable networks and degraded performance. Tracking the leading and lagging indicators for reliability is essential to improve overall resiliency.
- Security-related considerations should be tracked and measured. These are data interception, types of encryption protocols, and even the frequency of the updates and patches.
- Network design-related metrics are also possible considerations for tracking link redundancy and the overall fault tolerance.

The **Customer identity provider** domain covers the access for both platform users and machine users (service accounts) used by platform teams. Primary measurement considerations for this domain are usually around the following two measures.

- User management: Tracking and validating the user authentication metrics, multi-factor authentication (MFA) wherever applicable, user provisioning processes, and the regression testing of each of these based on the usage patterns you are seeing
- Configuration reviews: Misconfiguration of your identity providers often cause many vulnerabilities, not just for the platform but introduced through the platform and injected into the rest of your domains.

The **cloud service control plane** base defines the core runtime layer of the platform to enable the deployment and running of the workloads. The measurement considerations here are generally far more than the other domains due to the amount of user interaction and inputs you have as you go through the design of this layer.

- Performance, scalability, throughput, and operational efficiency can all be measured. Most of the time, the core platform implementation focuses on this runtime layer, which provides ample opportunities for you to track the capabilities needed for automation. Activities around API latency and usage metrics are both attributes to measure.
- Cost optimization is a significant consideration for tracking the success of your platforms due to the inherent leakiness of the services on the cloud. The optimization measures call for establishing the appropriate finops practices and democratizing that data.
- Security and compliance are critical pieces of this puzzle where you can track various access controls and monitor and measure any significant changes.
- Scalability around auto-scaling is something that most of your cloud-native products are capable of taking advantage of. Measures around the ability to do the scaling and understanding how the control plane is tracking the allocation

**Managed control plane services** measurements will be heavily contextual to your implementation itself. Based on the following, you can still create the necessary scaffolding for these measurements.

- Measurements around service performance based on the uptime, latency, and any SLAs established are great ways to see how you are measuring. One of the examples we gave in Chapter One was around Observability exporters. If you take that as a specific case, you can see that uptime, latency, and SLAs are all part of the equation as you measure.
- Another measure closely related to the autoscaling we referred to earlier in this section is the measure against workload distribution and the reliability around that.
- You want to remember the adoption and utilization metrics in this context, as most of the success of your platform can be attributed to the decisions you make in this domain.

Measuring the success and effectiveness of **managed control plane extensions** involves assessing various aspects related to their functionality, performance, reliability, security, and user satisfaction. Here are several metrics and considerations to measure the effectiveness of these extensions:

- Feature Completeness: Evaluate whether the extensions offer a comprehensive set of functionalities, such as managing DNS addresses, TLS certificates, secrets injection, dynamic provisioning, and service mesh management.
- Self-Service Adoption: Measure the extent to which users or customers utilize self-service capabilities provided by these extensions.
- Provisioning Speed & Efficiency Measure the time to provision new resources or services through these extensions, which in turn measures the success of your Infrastructure as Code strategy. You should also evaluate the level of automation (self-serve capabilities) in the provisioning process and its impact on reducing manual intervention.
- Upgrade Control & service quality: Measure the impact of upgrades or maintenance performed on these extensions to workloads or the platform itself and how that can impact the quality of the platform services.

Many of the overarching themes we discussed earlier from the measurement perspective around security, reliability, and adoption also apply to this domain.

Measures of success for the **platform product services** are usually around functionality, usability, efficiency, and adoption, all in the context of the value realization we discussed in Chapter One. Specifically, you will need to consider the following:

- How complete are the functional capabilities provided concerning what you can do? There are constantly diminishing returns beyond a certain point.
- The capabilities' usability can also be measured quantitatively and qualitatively. Attempt both approaches to obtain a fair view of the outcomes.
- Suppose you have starter kits within the product services leading to a paved path. In that case, you have to measure the efficiency - how quickly new services can be deployed to production using a starter kit. - or just the speed; how much time does it take for a new developer to start doing some deployments to lower environments?
- This domain is where we would measure the value metrics we discussed earlier. Primarily, the measurements around time savings and cost optimization are both key to ensuring the success of your platform product services.

At this point, you are starting to see a pattern emerging that is not surprising. Yes, for you to understand and make the experience of building a platform more successful and seamless, there are several things you have to measure and keep track of. You can only do that with a product mindset and end-to-end observability, self-serve, and self-healing built in across each domain.

## 3.2 path-to-production and Platform Value Metrics

This section delves into the importance of thoroughly analyzing your current path-to-production, from initial backlog placement to code deployment, to optimize efficiency and maximize value. Steps in the path-to-production primarily include the workload created by the developers to build the product. Understanding this process is critical for developing a successful platform strategy that minimizes wasted time and resources while enhancing overall productivity and business outcomes.

As you develop an overall platform strategy, we strongly recommend that you carefully analyze your current path-to-production, from the point where work is first placed in a backlog until the code is being used by the customer in production. Identify every step, the reason for the step, how effectively the activities or ordering of the step address the underlying reason, and the time required to perform. The *path* provided by your platform must optimize these steps, eliminate wasted time, and provide greater value than the cost required to provide the platform.

### 3.2.1 Why should you invest in Platform Engineering?

Platform engineering is more than just technical skills like automating infrastructure, setting up CI/CD, or developing tools. For it to be valuable, the organization needs to invest in and adopt these capabilities. If there's not enough investment, or if development teams don't know about the capabilities, how to use them, or understand their benefits, the platform won't succeed.

Platform teams need to link their work to business value. Typically, product leadership sets the roadmap and priorities for product engineering teams. Platform product managers must clearly communicate how platform capabilities can lead to real business benefits.

To ensure organizational alignment and adoption, we need to address these key questions for both individual capabilities and the platform as a whole:

1. **Outcome Delivered:** What does the capability provide? What technical or governance needs does it meet? How will it impact the team or the business when adopted?
2. **Adoption Summary:** What is the adoption plan? What documentation, examples, or support will be available, and how will adoption be measured?
3. **Value Summary:** What is the expected value for the team or organization from adoption, and how will this value be measured?

To gain the necessary insights, you can engage in specific activities such as:

- Mapping the value stream from development to production.
- Modeling and measuring platform value and metrics.

These activities help connect platform capabilities to business outcomes, ensuring effective communication and organizational alignment.

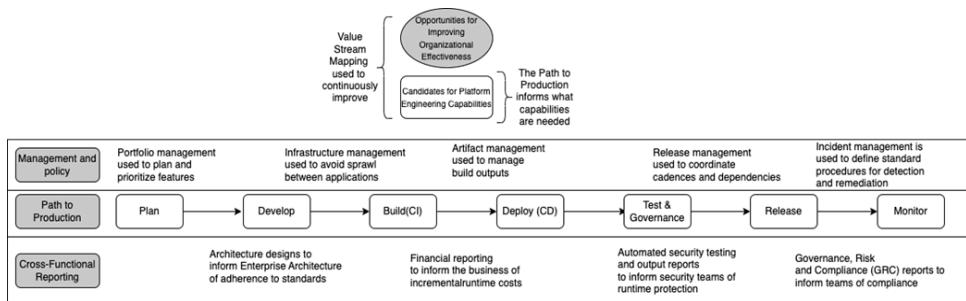
### 3.2.2 How do you identify the scope of your engineering platform?

*Value stream mapping* (VSM) has been a popular process mapping technique that originated in manufacturing to understand the current state and design a future state that eliminates the inefficiencies identified. The fundamental idea behind VSM in software originated with lean software principles as the complexity of the software development process increased, and the need to improve efficiency became paramount. Typically, within software development, VSM is used in the scope of the overall Software Development Life Cycle (SDLC) as the DevOps practices, techniques, and tools made it easier to improve efficiency through automation.

*Path-to-production analysis* is similar but more straightforward to the VSM techniques. It focuses mainly on the SDLC steps without considering many of the business processes, value propositions, and their impacts on agile software delivery.

In the context of platform engineering, VSM for the complete path-to-production provides deeper insights into the requirements for building repeatable platform capabilities, which can help address all the inefficient aspects of the SDLC.

In the diagram below, we show the steps of a simple path-to-production flow along with associated areas of a VSM.



**Figure 3.3 shows An Example of Value Stream Mapping (VSM) and path-to-production (P2P). This is used to identify areas that can be improved by optimizing or eliminating the chain of dependent steps in creating and deploying new features.**

Steps in the production process where you notice wasted time are perfect for platform engineering improvements. However, during a typical value stream mapping, activities like policy adherence and cross-functional reporting often contribute more to inefficiencies but are frequently overlooked. Including these in your platform strategy can lead to better solutions.

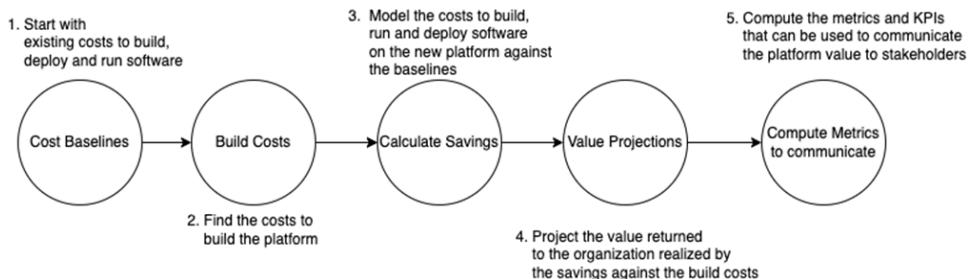
### 3.2.3 Platform Value Modeling and Metrics

How can we measure the value of investing in platform engineering and providing developers with an engineering platform? We need an objective and quantitative approach to show the return on investment (ROI). First, we simulate potential returns, then compare them against actual results. Projected ROI helps secure investment, and assessing actual value helps decide where to continue or stop investing.

This section introduces the Platform Value Model and Metrics (PVM). Building engineering platforms or internal developer portals (IDPs) comes with the challenge of value-based prioritization. PVM helps compute the projected value against the costs of building the platforms, ensuring that platform strategies are value-driven from the start. Without understanding the value generated, platform engineering efforts often fail to reduce developer friction and improve organizational effectiveness.

Defining Platform Value Metrics helps decision-makers prioritize investments and determine when to build new capabilities. The workflow below outlines the five-step process for platform value modeling and metrics computation:

1. Identify inefficiencies and potential improvements.
2. Simulate potential returns on investment.
3. Secure investment based on projected ROI.
4. Compare actual results against projections.
5. Adjust investments based on actual value delivered.



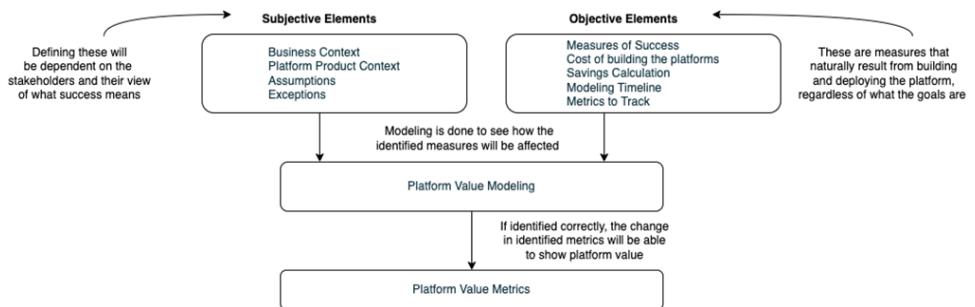
**Figure 3.4 shows the value modeling process. This can estimate the value the organization will see from building and using an engineering platform. Actual metrics and KPIs should result from this, which can be communicated to business stakeholders.**

First, find the cost baselines. Then, calculate the cost of building and supporting your feature. Next, model the savings. Finally, define the measure or metric that can track the value impact.

Technical product owners (TPO) should work with the platform developers to categorize data into subjective and objective pieces.

Remember that most of the value created through platform engineering comes from the quality and acceleration in value delivered to external customers. For example, if development teams save 20% of their time by adopting the platform, that should have the velocity impact of hiring 20% more developers. But more than that, it means software expected to help the business obtain more customers, sell more goods or services, or more effectively retain existing customers is getting in front of customers 20% more quickly.

The following illustration shows the critical elements of building a platform value model.



**Figure 3.5 Components of Value Modeling Process. These are both subjective and objective measurements that a TPO should identify by working with stakeholders and engineers that will be used to show platform value.**

**Business Context:** Working with stakeholders to understand the business context is critical to determining the overall platform strategy. Writing it down explicitly describes your business goals and the problem you are trying to solve with a platform-centric approach. It will be the first step in modeling the value that the process can generate. Writing down the business context will cover the vision and strategy described in section 2.1.

**Platform Product Context:** This is where you will discuss the platform product capability you want to build and the expected value generated from this effort. You need to have a clear vision where you can clearly articulate the specific outcomes and the adoption of the capability built by multiple development teams within your organization. For example, an organization might be saying that they want to develop standardized container orchestration platform capabilities that make the adoption of Kubernetes easier for at least 90% of the development teams where there is no more than a 10% increase in cognitive load, measured through feature development velocity.

**Assumptions and Exceptions:** Assumptions are critical to make sure that you provide an accurate picture of your business context. The assumptions will be like using an architectural decision tree to let the development teams determine if they should use a containerized solution instead of a serverless approach. The number of assumptions can be all-important applicable ones. They should consider the total cost of ownership, which in turn includes the reduction in the cognitive load of the developers.

In certain situations, consider the exceptions applicable in your context. An example of an exception could be that your organization has an existing training program with time allocated for developers to go through the enablement process, making it easier to ignore those costs upfront to avoid double counting.

**Measures of success:** This is the most crucial part of the value modeling process, where you identify how to measure the product's success. An example success measure might be a 30% reduction in the average time it takes for new code to go from the first environment all the way to Production.

**Cost of building platforms:** To understand the value of a particular platform capability, you need to know how much the cost was to build it in the first place. Determining the cost is done by considering the total cost of ownership, which can be a combination of the engineering costs to build it, cloud infrastructure costs, and related licensing costs.

**Savings calculation:** The savings calculation can cover many different aspects of improvement. Reducing the time it takes to perform a task, becoming compliant with legal or contractual requirements, or reducing the number or duration of outages are all examples where value can be estimated.

**Modeling timeline:** You need to identify and write down the timeline that works for your organization. Most digital native organizations are moving towards an agile budgeting model with increased flexibility by constantly forecasting their investments. You need to know the specific model that will work for your context. An example of this will be that a particular capability providing a net positive value for your investment in three years may not work for your investment models within the organization for a startup. However, this might be an excellent timeline for an established organization to build a capability to scale your product line.

**Metrics to track:** Which metrics to track at the investment level will depend on how your organization makes investment decisions. Most companies use some form of return on investment (ROI) calculation, in which the value returned over a period of time is compared to the cost of delivery.

### 3.2.4 Exercise 3.1: Develop a Platform Value Model

Develop a platform value model for PETech that can help communicate the value of your intended platform capability to decision-makers. There are no right or wrong answers for this exercise, as the outcome you seek is a value-driven response to whether or not you should invest in building a particular platform capability.

To complete this exercise, you should answer the following questions.

1. Describe your business context and why this platform strategy will be critical
2. Identify the specific platform product capability you want to build
3. List out the assumptions and exceptions
4. Identify the measures of success
5. Identify all the costs of building the platform
6. Obtain data from the path-to-production analysis (conduct one if you have not previously done a path-to-production analysis as described in section 2.5.2) to determine the savings goals
7. Identify a clear timeline
8. Identify the metrics you want to track against and clarify the formulas you want to use to compute these metrics
9. Compute the savings for these capabilities you are building based on the savings goals from step 6. Also, compute the costs of building these capabilities from step 5
10. Report the target metrics (from step 8) for the period you have identified (from step 7)

### 3.3 Cognitive load of developers and platform engineers

This section will teach you about cognitive load in software development, why it's important to minimize it, and techniques to achieve this. Understanding and managing cognitive load is crucial for improving developer productivity, reducing errors, and enhancing overall platform efficiency.

Cognitive load represents the working memory resources allocated during a given task in cognitive psychology. Even from the earliest days of computing and software development, researchers started thinking about the psychological aspects of development. Arguably, one of the earliest works in this space came from Gerald M. Weinberg in his landmark book, *The Psychology of Computer Programming* published in 1971.

Over the years, more and more attempts were made to improve the developers' user experience. Much of the work in Human-Computer Interaction (HCI) and related areas eventually culminated in the release of the Agile Manifesto in 2001, capturing the 12 fundamental principles that should guide software development. That effort has to take a lot of credit in how we look at software development as a whole, specifically platform engineering. To clarify this further, while HCI primarily focuses on improving user experience through better design and usability, Agile emerged as a response to the need for more adaptive, collaborative, and iterative software development processes. However, there is an indirect relationship in that both fields emphasize user-centered approaches. HCI's focus on creating better user experiences influenced the mindset of putting the 'customer first'—a key principle reflected in Agile.

#### 3.3.1 What is cognitive load and how to measure?

Cognitive load in software development refers to the effort developers put into learning and performing complex tasks. This complexity varies because teams typically consist of developers with different experience and skill levels. The main goal of discussing cognitive load is to reduce it as much as possible so developers can focus on solving new, high-value problems.

Organizations can measure cognitive load in two ways:

1. **Subjectively:** Developers self-report their mental state.
2. **Quantitatively:** Structured measurements assess the impact of cognitive load.

Cognitive load comes in two forms:

- **Intrinsic Load:** Related to the specific task a developer is performing. Platform engineering techniques can often reduce this type of load quickly.
- **Extraneous Load:** More complex, involving the strategic direction of the company, product understanding, and overall culture. Even in organizations with a good culture, misunderstandings about product requirements can create suboptimal situations. Building platform capabilities doesn't directly address extraneous cognitive load, but it's equally important.

For platform engineers, extraneous load is similar, but their intrinsic load is different. They face task complexity and the added pressure of working closely with their "customers" (other developers) and constantly needing to prove their value.

In Section 3.5, we'll discuss cultural measurements and approaches in detail. We'll also refer to the work by Matthew Skelton and Manuel Pais on Team Topologies, which provides a framework for these concepts, focusing on enabling teams.

### 3.3.2 Why reduce cognitive load?

As organizations begin to measure cognitive load and its impacts, it's important to clearly understand what aspects of cognitive load can be measured. Recent research in the developer experience (DX) field has identified two additional key areas to focus on: feedback loops and flow state.

**Feedback loops** in software development are essential for quick and high-quality responses to actions taken. They help developers get input, evaluate it, and adjust their work accordingly. These loops include not just functional testing of code, but also code reviews, performance feedback, stakeholder input, and retrospectives at the end of each sprint. Efficient development depends on fast feedback loops, which allow tasks to be completed smoothly and quickly. Slow feedback loops can disrupt the development cycle, leading to frustration and delays. To improve efficiency, organizations should shorten feedback loops by optimizing development tools and processes, such as build and test procedures or the development environment.

**Flow state** refers to a developer's complete absorption and enthusiasm in their work, resulting in intense focus and enjoyment. Experiencing flow regularly enhances productivity, innovation, and personal growth. Studies show that happy developers often produce higher-quality outcomes. Therefore, fostering conditions that promote flow is crucial for improving employee well-being and performance.

The concept of flow state can be compatible with pair programming and mob programming, though achieving flow in a collaborative setting might look different than in solo work. In pair or mob programming, flow can emerge when the team achieves a shared rhythm, clear communication, and mutual understanding of the task. When done effectively, these collaborative methods can still foster deep focus and engagement as long as the team members are synchronized and distractions are minimized.

While flow in solo work is often about individual immersion, in pair or mob programming, it becomes about group cohesion and collective problem-solving, which can also enhance productivity, innovation, and personal growth. Creating an environment where team members can support one another and stay focused on their shared goals allows for the benefits of flow in a team context.

Reducing cognitive load is critical for enhancing developer productivity. However, not all cognitive load should be eliminated. Loads that contribute to meaningful learning and comprehension are beneficial, as it helps developers improve their problem-solving skills and contributes to the platform's evolution.

Developers are the primary users of engineering platforms and often know best what they need to work on. While this can be debated, platform engineers should not dictate how developers use platform capabilities. Instead, enabling developers and changing the way they work should be a collaborative effort, facilitated through effective technical product management.

The goal is to allow developers to focus on efficiently implementing functions without worrying about the underlying technologies. This can be achieved by abstracting specific tools to be composable and replaceable within the platform. Ultimately, the aim is to improve overall efficiency and effectiveness, boosting both the bottom line and top line of the business.

### **3.3.3 Techniques for reducing cognitive load**

Organizations use several techniques to address the challenge of cognitive load. Traditionally, automation of repetitive tasks was the cornerstone of every strategy to solve this problem. But with an increased focus on improving the friction that causes suboptimal developer experience, there has been a more intentional approach around consistency, training, and democratization of the required data. These techniques can summarized as follows:

1. **Simplification of problem domain:** Suppose you are trying to deploy a kubernetes based application. Since the deployments would use predefined templates, your engineering platform could offer the capability to build a library of the Kubernetes manifests that can encapsulate the most common configurations and best practices.
2. **Consistency:** Using developer portals such as Backstage from Spotify can significantly reduce the number of differing paths by having a paved path to carry out the same task. Suppose your team is using GitHub Actions. A plugin provided in the developer portal can help every developer manage the workflows in the same way, providing seamless collaboration and consistent vocabulary.
3. **Data Democratization:** One cannot overemphasize the importance of having access to the same data across your whole software development lifecycle. Observability is the first approach that guides actionable insights that can fix these problems before they reach the end-user. Suppose you are building a set of containerized microservices orchestrated in your Kubernetes cluster. By using a platform capability that can incorporate a distributed tracing tool into your service mesh, you can now see how the requests are flowing between services and any programmatically identified bottlenecks and address them.

### **3.3.4 Exercise 3.2: Identify and setup measurement techniques for reducing cognitive load**

For this exercise, we want you to create a backlog of ten potential platform capabilities that can help reduce the cognitive load.

We would like you to create a table in the following format.

**Table 3.3 Sample table to be used for tracking the cognitive load issues and the platform capabilities that can fix the issue**

Cognitive load issue	Platform Capability that can fix the issue	How will it solve the issue?	Effort in person weeks
.....	.....	.....	.....
.....	.....	.....	.....

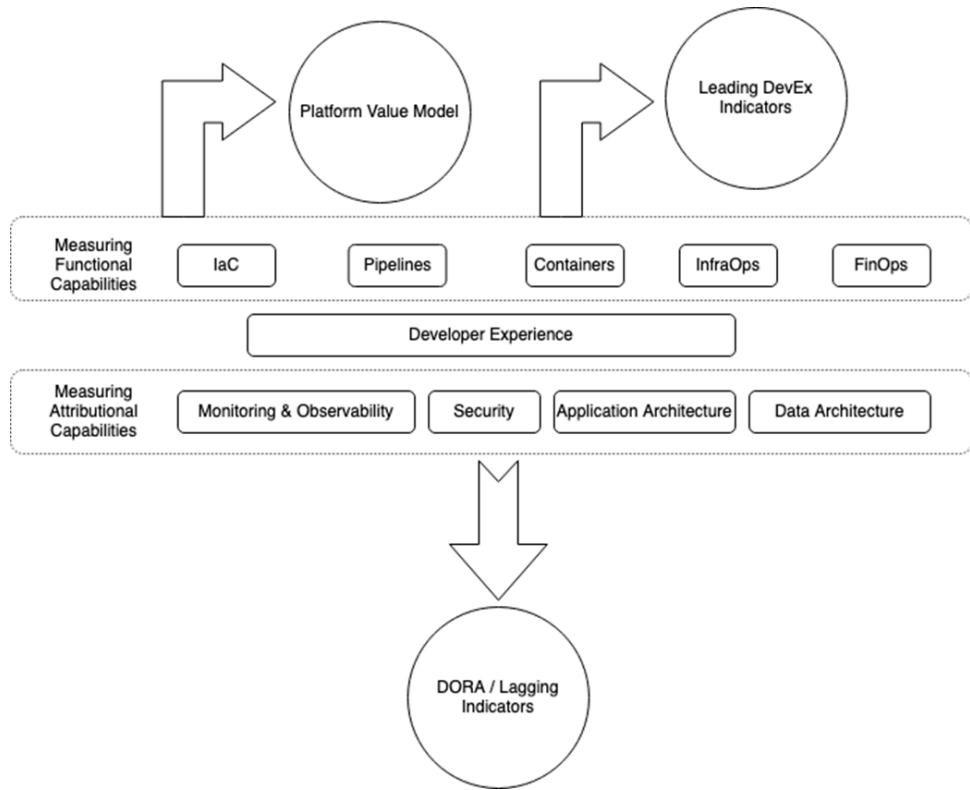
## 3.4 Democratization of capability development

In this section, you will learn about how to streamline the measurement process to reduce cognitive load for developers by using a capability model for platform engineering. Understanding this model is essential for building, managing, and evolving a successful platform within your organization, ensuring continuous improvement and alignment with business goals.

Now that you have learned about and tried out a few hands-on examples of identifying the measurement criteria, followed by the purpose - reducing the cognitive load of the developers, it's time for us to see how to make the measurement process seamless and less arduous. The authors have long been practicing the concept of capability models. A capability model for platform engineering outlines the critical areas of expertise, skills, and functions necessary to build, manage, and evolve a successful platform within an organization. It typically includes various dimensions that cover technical, operational, leadership, and strategic aspects of platform engineering with a quantitative backbone that will help you improve from where you are today.

### 3.4.1 A platform engineering capability model

Shown below is a platform engineering capability model. The model will have five distinct parts. At the core are the actual capabilities, divided into three parts.



**Figure 3.6 Platform engineering capability model.** This capability model has 3 parts that talks about functional capabilities, attributional capabilities and developer experience between those two. Developer Experience components are described below

Part one is the functional capabilities, where you will measure the adequacy of the operational capabilities around Infrastructure-as-code, Pipelines, containers, infrastructure operations, and financial operations. These are the capabilities you build as part of your platform.

Part two is the attributional capabilities, where you will do some building as part of the platform but driven by the key attributes that drive the functional capabilities of the platform.

Part three, as a stand-alone entity sandwiched between the first two parts, is the developer experience. The decisions and standards around the attributes and functionality of the capabilities both drive the developer experience. The developer experience indicators in this space include aspects like ease of *onboarding new developers*, *documentation quality through any channels available*, *overall developer productivity measured both quantitatively and qualitatively*, *ease of maintenance of the code*, *third-party community support enablement and accessibility*, *platform stability and reliability*, *extensibility* and finally the framework for responding in an efficient manner to the available feedback.

While individually measuring these was explained in detail in section 3.1, these layers provide the data to better establish your platform value metrics, as generated from the platform value model and your leading developer experience indicators. The final part of the model is the lagging indicators that tell you how exactly the decisions you made in your SDLC and the platform journey contribute to the business outcomes you seek.

### 3.4.2 Models for development of PE capabilities

Now that we have looked at the overall capability model, we should go down each of the parts described above to understand more. Suppose PETech is figuring out its capability maturity level and wants to increase it; the place it would start could be by mapping to the SDLC. Their example questions were along the following lines.

In Figure 3.7 below we break down the top layer of figure 3.6, which talks about the specific capability considerations in each of the areas of focus.

Infra as Code	Pipelines	Containers	InfraOps	FinOps
IaC Strategy	Functional CI/CD	Container Strategy	Operations team model	FinOps team?
Tool Choice	Continuous Deployment	Orchestration Strategy	Network topology	Cloud spending models?
Config Automation	Continuous Delivery	%age Containerization	Bespoke Servers?	Off-peak hours?
%age Infra Automation	Deploy & Test	Use of Service Mesh	Bursty workloads?	Unit cost economics
Ownership	Workload requirements	Image Tagging	Predictions & forecasts?	Resource tagging?

**Figure 3.7 Functional capability considerations for each of the five areas of focus we described in the top layer of figure 3.6.**

As we consider the attributional capabilities to measure, we need to point out that the best way to measure them is to build an easy-to-use questionnaire with meaningful and trackable responses. Keeping the responses as consistent as possible will also make it easier for you to receive more responses and use them in an actionable manner.

Similarly in figure 3.8, we will break down each of the attributional capability considerations we introduced in figure 3.6. For break this down by specific axes.

Monitoring & Observability	Security	Application Architecture	Data Architecture
Purpose of Monitoring?	Solving the identity problem?	Loosely coupled architecture	Streaming/real-time data?
Purpose of Observability?	Change audit mechanism	Domain Driven Design	Data Security Layer
Application profiling?	Compliance at the point of change?	Enterprise & Solution architects	Data Experiments?
What is the platform of choice?	Code scanning platform?	Architectural Metrics	Data Mesh
Developer access to telemetry?	Use SSO?	Architecture impact on build/test/deploy	Tracking data growth

**Figure 3.8 Attributional capability considerations for each of the four areas of focus we described in the bottom layer of figure 3.6**

Having covered the first two parts, we can now look at the indicators for the developer experience measurement in Figure 3.9. Developer Experience is an area where we see significant improvements in thinking around tracking and measuring.

# Developer Experience

DevOps Culture

Tests Pass before deploy

Atomic Commits

Branching Strategy

Code Quality

Knowledge Management

**Figure 3.9 Developer experience considerations are shown above that tells you some of the key things you should be thinking about when you are trying to improve developer experience**

As described in section 3.2, **Platform Value Modeling (PVM)** is like drawing a map that helps determine if building a platform is worth it. Doing this right at the beginning of planning a platform is important. Why? Well, if you need to know how much value the platform will create, it's easy for all the hard work of building it not quite to hit the mark. PVM also helps decide what to build and how much fancy stuff is needed to get the value you aim for. It's like having a guide to help make intelligent decisions about what to focus on first. The modeling offers insights into potential scenarios and allows measuring the value of platform product capabilities throughout their lifespan. However, effectively communicating these insights to organizational leadership can take time and effort. To address this challenge, we suggested utilizing specific metrics tailored for this purpose, which we refer to as **Platform Value Metrics**. These metrics are typically proactive indicators, offering a clearer view of your platform's progress right from the start rather than retrospective assessments. An example of five of the metrics are shown in figure 3.10

# PVM

- Value to Cost Ratio
- Developer Toil Ratio
- Innovation Adoption Rate
- Platform Gap Ratio
- Capability Adoption Ratio

**Figure 3.10 Platform Value Metrics (PVM).** An organization can strategically determine several PVM considerations. This example shows five of the key ones we believe every organization should use

Compared to the other directional metrics, for PVM, we are providing some concrete suggestions for building them out.

The **Value to cost ratio** is the proportional value generated for the cost you have incurred in building them.

The **developer toil ratio** measures the number of repeated activities developers were able to avoid using a platform capability provided.

The **innovation adoption rate** emanates from the opportunity cost, which talks about whether the developers were able to avoid the toil and were able to use that to increase the new capability development during that time.

**Platform gap ratio** is the concept of the organizations identifying the number of capabilities actually built for the potential number of capabilities you end up making.

The **capability adoption ratio** is similar to the Platform gap ratio, but in this case, we track the adoption of these capabilities instead of simply building them.

### 3.5 Organizational aspects of PE success

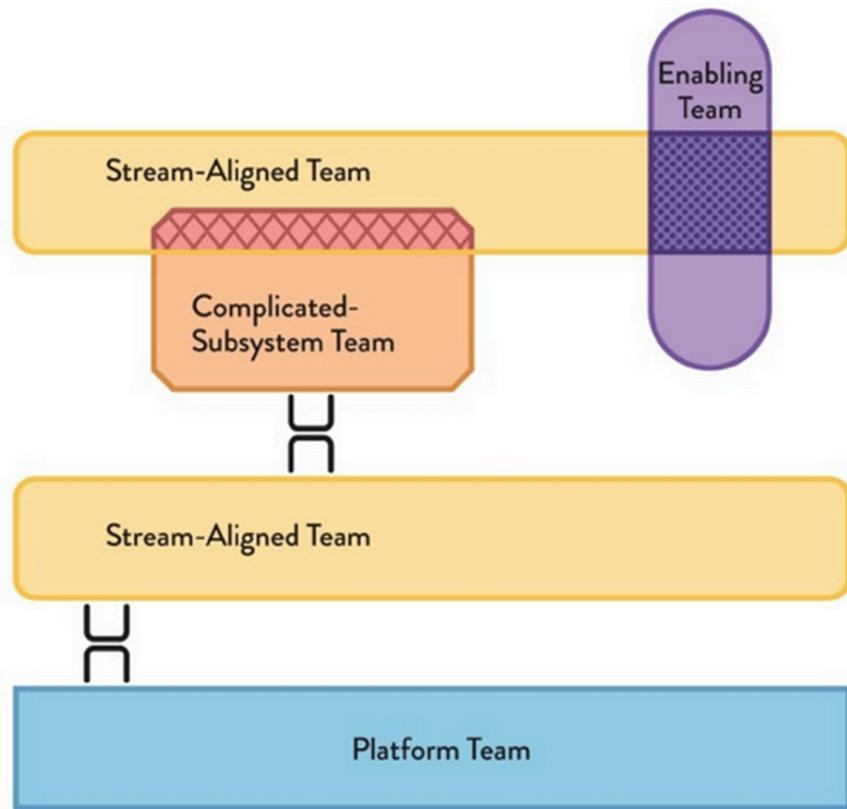
This section will explore how organizational culture influences the success of engineering platform initiatives and why fostering the right culture is critical. Understanding these cultural aspects is essential for creating an environment that supports effective platform implementation and long-term success.

One of the most repeated quotes attributed to Peter Drucker is, "Culture eats strategy for breakfast." Having discussed strategy a lot earlier in this chapter and Chapter One, it would be imperative for us to discuss the value of the organizational culture as something that is more important for the overall success of your engineering organization to support the successful implementation of the strategy.

We will now talk about how culture impacts the overall engineering platform lifecycle in the next cycle.

#### 3.5.1 Changes needed for an organization to prepare for platform engineering

There is a step-by-step process an organization can follow as they embark on their engineering platform journey. Creating a culture that enables the teams to work together with mutual trust and the ability to share information will create more innovation and accountability. This focus on the culture goes to the critical message of *Team Topologies* (Skelton & Pais, 2019), which refers to creating four distinct types of team topologies, as shown below. Their landmark work discusses the critical requirements of creating these four types of teams as shown in figure 3.11.



**Figure 3.11 Primary Interaction Modes for the Four Fundamental Team Topologies.(Image taken from the book *Team Topologies* by Matthew Skelton and Manuel Pais, 2019. Used with permission. )**

In the context of our discussion, it is essential to note that the platform team serves the stream-aligned teams through the idea of the platform product we explained in Chapter Two.

While we have described the concept of a platform team so far, it is also essential to understand what the original authors meant by the following terms.

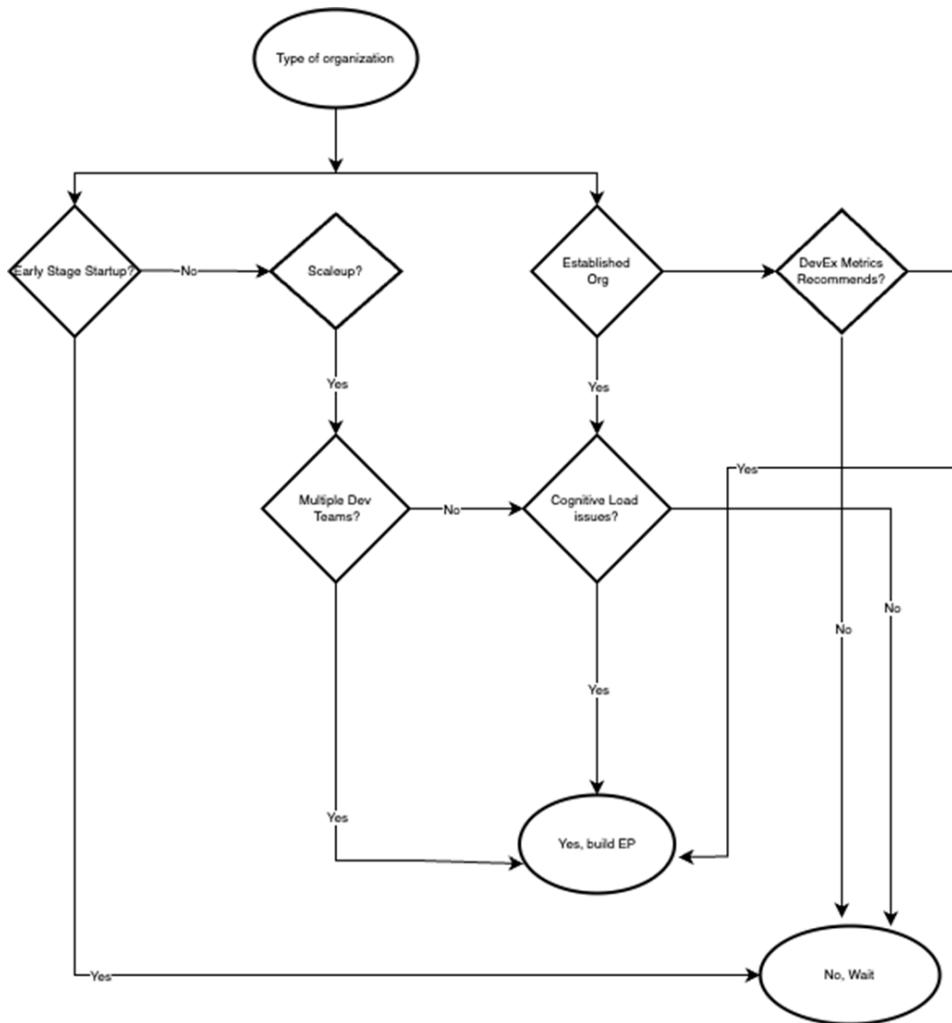
- **Stream-Aligned Teams:** These teams are organized around a continuous flow of work aligned with a specific business or customer value stream. They have end-to-end responsibility for delivering and maintaining their product or service.
- **Enabling Teams:** These teams help others by providing expertise and guidance in specialized areas like automated testing, security, or architecture. They work temporarily with stream-aligned teams to remove obstacles and build capability.
- **Complicated Subsystem Teams:** These teams focus on highly specialized or technically complex areas of the system, requiring deep expertise that is difficult to distribute among other teams. They handle parts of the system that need particular attention due to their intricacy.

While a complicated subsystem may or may not be relevant in your scenario, understanding that there could be a situation where some stream-aligned team could use the support of such a team is essential. The same thing goes for the enabling team concept. They guide the stream-aligned team by enabling them on new practices and technologies created by platform engineering and anything relevant to product development and delivery.

### 3.5.2 Prerequisites for the change

Not all organizations need to have a platform engineering team or be building engineering platforms. We use the following decision tree to determine whether you should be building engineering platforms. Most vendors of platform engineering solutions recommend ways to decide whether or not you are an ideal candidate to start building your engineering platform. We recommend using a simple decision tree, as shown below.

The diagram shown below in figure 3.12 represents a decision tree guiding organizations on whether they should build an engineering platform (EP). It starts by identifying the type of organization, such as an early-stage startup, a scaling company, or an established organization. If the organization is scaling or established, has multiple development teams, or faces cognitive load challenges (e.g., overwhelmed teams), we recommend that an engineering platform be built. Additionally, it considers if Developer Experience (DevEx) metrics suggest building a platform; if not, it suggests waiting. The flow aims to help organizations evaluate whether investing in an engineering platform is appropriate based on their growth stage, team structure, and developer experience needs.



**Figure 3.12 A Decision Tree for Engineering Platform Buildout.** This is a question that is often asked whether an organization is ready for an engineering platform adoption. This simple to use decision tree gives you clarity in making that decision.

Our recommendation is simple. If you are a startup trying to get your first products out to create a viable business model, you should not worry about building an engineering platform. As most startups start scaling up, it is the first time they will start thinking about improving the economies of scale and abstracting out the common capabilities, showing the first signs of the need for an engineering platform.

On the contrary, the simple rule of thumb for the established organization is the concept of cognitive load, which we discussed in section 3.3. Cognitive load can manifest easily as reduced productivity exhibited using lower sprint velocity or other measures such as increased employee attrition. However, we recommend that larger organizations establish their developer experience measurement techniques before such signals. Measuring the specific indicators we described in section 3.4.2 is something every organization should start as soon as they can and by the time they have identified their business strategy and product operating model. These robust metrics will tell you whether you need to start looking into the engineering platforms.

### **3.5.3 Implementing organizational changes**

Implementing the requisite organizational changes is another cultural aspect that has to start with awareness and openness with the support of interdepartmental influences.

DevOps was a software development-driven reaction to the systemic collapse of IT operations' inability to respond due to increased delivery frequency needs from a rapidly maturing software engineering practice.

The original ask from software development teams has evolved over the last 25 years;

For example, "I need to frequently and rapidly push incremental functional and architectural changes to production."

Or, more fundamentally, "I need the rest of the software delivery supply chain (IT Operations, Security, Compliance, Business Operations, et al.) to become as mature in engineering practice as software development."

As a whole, the IT-Ops industry is very resistant to evolving past the models and ways of thinking born out of the '60s and '70s: high capital investment, maximized utilization, maximized lifespan (minimize change), organizational separation by technology function, budgetary separation, cost-center valuation. This "mental model" is among the top contributors to creating the opportunity for the broad market disruption that occurred/continues to occur.

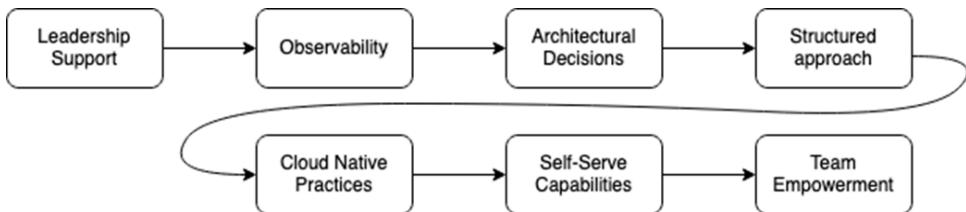
Without a profound and structural shift in the engineering culture and organizational structure of Enterprise IT operations, realized value from the attempt to apply DevOps practices proves to be relatively low and transitory. Platform engineering practice is similarly a software-development-driven reaction to enterprise IT's apparent inability to mature. The EP experience simply results from adopting and applying mature software development practices throughout the software delivery supply chain.

The acceptance and success of platform engineering changes emerge not merely from technical prowess, but from the nurturing of human-centric activities, where champions are identified across every organizational facet, and measures of success are calibrated through the lens of developer feedback. Only then do the wheels of organizational evolution turn with graceful fluidity, encountering minimal resistance along their transformative journey and reducing the cognitive load of the developers.

### 3.5.4 Scaling platforms in organizations

As discussed earlier in this section, **leadership support and alignment** are vital to scaling the platform engineering practice. Nothing succeeds as success itself, and as we detailed in the decision process, not all organizations are ready for platform engineering or need an engineering platform.

Let us look at the key considerations for scaling shown below in figure 3.13



**Figure 3.13 Key prerequisites for scaling platforms. Each of the seven prerequisites indicated here are all equally important in deciding to scale the platforms.**

You can only improve something appropriately with data. **Collection and dissemination** of this data can be quite cumbersome with the volume of data and the complicated ecosystem of tools your organization might use. Have a solid observability strategy spanning multiple axes; we will discuss this in depth in Chapter Seven. The actionable insights directly from the observability will be essential for you to start scaling what you do.

Making your engineering platforms or any platforms **architecturally sound** is the most important thing for your consideration. Just like in any software design, the benefits of loosely coupled architecture give you the necessary flexibility to implement a lot more basic tenets such as composability and replaceability, which will become increasingly important as you move along in your EP journey.

While it is tempting and mostly recommended to focus on the most pressing problems that are visible and trackable, think about a **structured approach to building automation and tooling**. There is no better way to do it than to look at your path-to-production (Section 3.2 provides an approach to looking at this with a value backdrop).

Following the technology evolution is inherent to any of the platform engineering efforts. Embracing **cloud-native practices** and leveraging managed services as part of your platform capability implementation will ensure that the end-users see the most value in the platforms. These practices should inherently include aspects related to cost optimization, modern resiliency, reliability, governance, and security without the need for explicit mention.

**Self-service capability** is yet another of the basic tenets of platform thinking. Any expectation of scaling has to be preceded by a thoughtful approach to having developer self-sufficiency. Making developers self-sufficient is challenging and requires empathy interviews and understanding the user base with a clear technical product management function. It also has the inherent assumption of certain levels of skill sets for the stream-aligned teams. Having the right kind of contextual documentation, the use of chatbots, and an enabling team are all beneficial to achieving this goal. Over time, the lack of a self-service mindset in building capabilities will render all support activities ineffective for scaling.

In platform engineering, as you might already know, **self-serve** allows developers to access autonomously and provision infrastructure, tools, and services without relying on centralized teams. This increases productivity and removes bottlenecks. Examples include developer portals for creating environments, Infrastructure-as-Code (IaC) tools like Terraform, and platforms like Kubernetes that allow teams to manage their deployments.

However, self-serve can present challenges, particularly when developers provision infrastructure and expect others to handle compliance, security, and governance. If not properly managed, this can lead to gaps in security. To mitigate these risks, organizations can implement guardrails, offer pre-configured secure templates, and conduct regular audits to ensure compliance, balancing developer autonomy with security requirements.

**Team empowerment** is closely related to self-serve and naturally evolving from that tenet. We believe every engineering platform should be open to contributions from the end users. Having the stream-aligned team members create pull requests for the platform engineering team to share their solutions, ideas, and changes will create a culture of collaboration, transparency, and ownership. Developers creating pull requests might mean additional work for the platform engineering team in the form of reviews and integrations, but that extra work is well worth the effort in the long run.

It is now time for us to revisit our favorite organization and look at a case study.

### EXERCISE 3.3: CASE STUDY: SCALING PLATFORM TEAM AT PETECH

At PETech, a captivating journey unfolded when a platform engineering team was born from the vision of abstracting essential capabilities. Their knack for value simulation was spot-on—calculating a promising value-to-cost ratio of 1.5, indicating an expected return of \$1.50 for every dollar invested.

With 23 eager teams awaiting the benefits of these platform capabilities, concerns loomed about scarce capacity. Shifting mindsets among specific teams posed a more significant challenge than anticipated, hindering complete buy-in and adequate funding. The platform engineering team initially drew financial support from Team Amulet, armed with a dedicated team of 14 developers managing three interrelated microservices.

Budget constraints made hiring a technical product manager unfeasible. Instead, they invested in one lead DevOps engineer, Emma, who brought five years of hands-on experience in build and release using Jenkins. Her leadership skills, honed as a team lead managing code compilation, integration, and unit tests, were instrumental. Emma's proficiency in Python and Jenkins secured her spot as the lead for PETech's budding platform engineering team.

Joining Emma were four diverse DevOps engineers, each skilled in PowerShell automation, open shift administration, AWS DevOps suite, and Azure Resource Manager/Bicep templates. This strategic mix covered critical aspects—automation, container orchestration, pipelines, and infrastructure provisioning.

If you advise PETech in its platform journey, what are the acceptable patterns and unacceptable antipatterns in the following areas?

1. Hiring and team setup
2. Planning and execution
3. Team empowerment

What did the PETech leadership do right? What could they have done differently?

What did Emma do well? What else could she have done better to enable scaling the platform capabilities?

## 3.6 Platform engineering KPIs

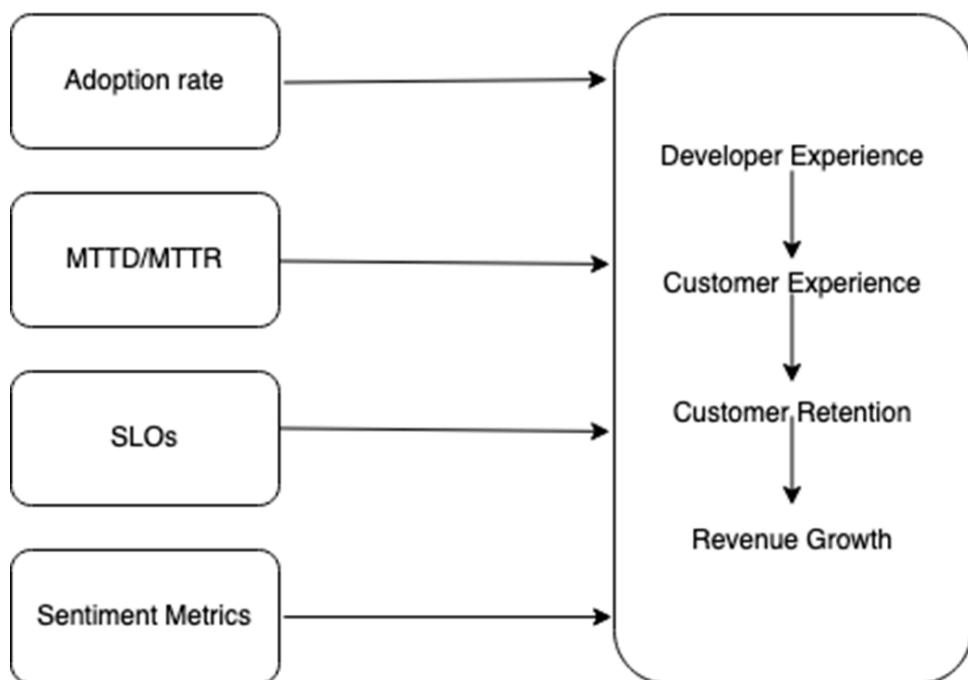
We will now talk about the importance of defining key performance indicators (KPIs) that align with your platform's operating model. Understanding and tracking these KPIs is valuable because they help measure the platform's adoption, efficiency, and impact on business outcomes, ensuring the platform delivers its intended value.

As illustrated below in figure 3.14, the process has to start with defining the relevant KPIs that align with your platform's operating model. There are four fundamental groups of metrics you have to begin with, all of which are related to the value of the platforms

1. Adoption rates that directly impact the developer's productivity
2. Mean time to detect and recover from customer issues that affect the customer experience and retention
3. Service level objectives that can help build trust as the critical element of the measurements
4. End user sentiment metrics, which have a significant amount of subjectivity but are just as necessary as the first three

# Engineering Platform → Outcomes

Groups of relevant metrics that provide early indication of value



**Figure 3.14 A mental model for platform KPIs.** The four KPIs indicated on the left side for the engineering platform drives the business outcomes every business should care about.

In the subsequent subsections, we will look at the KPIs for each of the four fundamental groups of metrics and what to look for from leading indicators for each of those, followed by how to map that into the eventual business outcomes. Now, let's examine how the lagging indicators in the outcome group impact each metric group presented in section 3.6. We are only considering the most vital lagging indicators. As part of the exercise following this section, you will be able to identify more lagging indicators that you can measure.

We've created summary tables for each of the four outcome groups. Here's how to interpret them: Each table outlines how the lagging indicator for each outcome group might affect your engineering platform. We've mapped these lagging indicators to the four standard sets of metrics used for evaluating the engineering platform.

### 3.6.1 Developer Experience KPIs

**Developer Experience:** For Developer experience, we are considering six lagging indicators. We are starting with the standard DORA4 metrics and adding one other critical qualitative metric essential in an engineering platform framework— Developer sentiment for the stream-aligned developers.

To determine if the **deployment frequency** will be positively impacted, assess whether developers are onboarding onto the platform and evaluate the overall effectiveness of the process. You would need lower MTTR for the problems in platforms, which can help the developers increase the frequency of the deployments. As part of providing the platform capabilities, it is essential to establish the service level objectives for the platform capabilities so that the developers can access the capabilities needed for frequent deployments. As the sentiment becomes more positive, you will start seeing this through the continuous delivery process to deploy to lower environments.

The next lagging indicator under developer experience is the **lead time for changes**. This indicator measures the time it takes for a specific change in functionality to move through the flow and get deployed in production. Coverage metrics - the amount of platform capability covered - will help drive the developer's appetite to adopt the capabilities as they try to reduce the lead time. The faster you identify the issues, the quicker the platform engineers can turn around and help this cause. Adhering to the established target objectives stands as the critical factor. The mere availability of automated paths to accelerate the areas of friction in your value stream makes this process seamless.

Regarding **MTTR** as the lagging indicator, platform feature (capability) coverage utilization is a key trigger point. The fundamental question would be to see what features are available to make it easier for the developers to detect and resolve the issues they see in their product. While the domain or functional-specific capabilities are of primary interest to the developers in this context, letting the developers focus on those would require sufficient coverage of the platform capabilities. To achieve the SLOs set for the activity, developers expect an increased level of self-healing of the problems. This way, issues are addressed without explicit intervention. This can lead to an increased MTBF (Mean Time Between Failures), improving the developer sentiment.

Reduction in developer toil is the primary indicator for improving the **change failure rate**. Predictive analytical models that base the decisions on prior activities and their responses will help determine the resolution time. You should be careful not to get sucked into the illusion of arbitrary measures and should be established upfront based on how much the business is ready to invest for that. All this can lead to an increased innovation adoption rate, where the developer's sprint velocity for new features improves naturally.

While most high-performance organizations track the four lagging indicators from DORA metrics, monitoring and addressing the challenges around **developer sentiment** is just as important. There is a direct correlation between improved developer sentiment and the feedback frequency to the platform team. You will now start seeing more pull requests from the developers as they start believing in the platform strategy and want to be part of it. Ultimately, this means far better onboarding experience and frequency.

Table 3.4 below summarizes this discussion for easy reference. As described in the previous section while introducing these tables, the correlation is made as follows.

For example for the second column in the first row, the *onboarding experience and frequency* of the developers help the adoption rate of the platform capabilities. This indicator is the early sign that the eventual lagging indicator of deployment frequency is adequately met. Similarly a mapping is established between every column (leading indicators) for each of the rows.

**Table 3.4 Developer Experience Lagging vs Platform Leading Indicators**

		Leading indicators			
Lagging Indicators		Adoption Rate	MTTR	SLOs	Sentiment
Deployment Frequency		Onboarding experience and frequency	Lower MTTR	Platform capabilities	Continuous Delivery
Lead time for changes		Coverage metrics	Faster identification	Alignment with target SLOs	Automation tools
MTTR		Coverage utilization	NA	Self healing	Increased MTBF
Change Failure Rate		Reduced developer toil	Predictive analytical models	Quantified metrics	Increased innovation adoption rate
Developer Sentiment		Feedback frequency	Pull requests	Pull requests	Onboarding experience & frequency

### 3.6.2 Customer Experience KPIs

Next, let us consider the next step in the outcomes that follow the improved customer experience. Several factors measure customer experience. The important ones among them are the *decision to remain a customer, the errors they see functionally on the product, security incidents of importance, adherence to service level agreements, and eventually, the customer satisfaction score (CSR)*.

These factors directly impact the leading indicators you see in your engineering platform work. The customer's eventual **decision to stay** is typically rooted in the value they derive from using the product, which directly correlates with the product's development quality and ability to address the specific problem they aim to solve. While that is indirectly related to engineering platforms (EP), MTTR and SLOs are directly associated with EP. The supplier's responsiveness to the consumer's issues and the overall offering's reliability are a natural offshoot of how well the EP is conceived and built. This responsiveness leads to better overall engagement. **Customer visible errors** are another lagging indicator most customers look for as they assess the experience widely. As the adoption rate increases, the consensus is that the number of visible customer errors proportionately increases. Still, the accurate measure is whether EP can help decrease or stabilize that number. Similarly, on the MTTR, you expect a reduction in time with the help of engineering platforms, and the objectives would lead to alignment. The measure that you should aim towards is improved customer happiness concerning errors with more users for the product. Sounds strange? No, this is the real benefit of EP you should have as a goal.

**Security incidents** in the products get a lot of press, leading to reputation impact and, more often than not, impact on revenues. Engineering platforms should be your friend as you look for options. Similar to functional errors, or perhaps even more so, we expect EP to help reduce the number or rate of errors with increased adoption, leading to reduced time to resolve issues. Every security incident is one too many for most service-level objectives, but its severity could have a significant impact on reputation. More and more customers are very clear about their **SLA expectations**, especially with many cloud-native services becoming part of any product architecture. Maintaining SLAs is essential, but engineering platforms help you use the built-in telemetry capabilities to stay manageable without overengineering your solution. The expectation is that when downtime happens, the SLAs ask for minimal downtimes and continued adherence to the SLOs while fixing them. The EP directly helps both of these. Improving **customer satisfaction scores** starts with building and increasing the user base. The ability of your product teams to respond to problems quickly and in the established framework builds the confidence and trust the customers have in your product and eventually delights them. These are summarized in Table 3.5.

**Table 3.5 Customer Experience Lagging vs Platform Leading Indicators**

		Leading indicators			
Lagging Indicators		Adoption Rate	MTTR	SLOs	Sentiment
Decision to stay		Value realization	Responsiveness	Reliability	Engagement
Customer visible errors		Decreases	Reduces	Alignment	Improves
Security incidents		Decreases	Reduces	Impact	Reputation Revenue
SLAs		Maintain SLAs	Minimize downtime	Adherence to SLOs	Reputation Revenue
Customer Satisfaction Score		Build User Base	Build Confidence	Build Trust	Build Happiness

### 3.6.3 Customer Retention KPIs

Let us look at how engineering platforms can help you retain your customers. Businesses need to consider the impact of an internal engineering platform as directly impacting customer retention. In most companies, business operations and the leadership team think of EP efforts as something their technology teams should worry about and pay for. When the discussions come up around funding these efforts at planning and budgeting sessions, some business leaders tend to deprioritize such efforts instead of getting more customer features that can lead to top line growth. That is not the right approach. Here's why.

As in the previous section about customer experience, we will first identify five lagging customer retention indicators. There can be many more, but these five will make a compelling case for EPs. They are *customer lifetime value, net promoter score, renewal rates, referral rates, and customer engagement metrics*.

**Customer lifetime value** has to increase with an increased adoption rate of the products. This increase in adoption can happen only with a more streamlined approach to building the capabilities. If the developers need that ability, they need a much more capable engineering platform to support their development activity. Reducing your MTTR increases the value perceived by customers, boosts revenues, and progressively enhances the value that customers bring in. Aligning the objectives set with the SLOs makes it a straightforward decision for the customer to continue retaining your products. The challenges of keeping the **Net Promoter Score** high with increased adoption are immense as you look at promoters, detractors, and passives. This situation is so because the adoption rate for your early adopters will be very different from your laggards. NPS management opportunities are precisely why engineering platform capabilities can help developers address other issues equally well, reducing the time between failures and creating a more reliable product that builds trust.

The continued use of the product naturally leads to improved **renewal rates** where the expectation around minimal disruption and more inherent trust of the SLOs happen. Product usage increase occurs through more confidence. Once the customers are happy with the value they generate, they look like superstars within their organization. At that point, you start getting more **referrals** through word of mouth. When you are in this situation, the quality of problem resolution powered by engineering platforms is an important factor to consider, and it may not be a bad idea to provide incentives to the clients for referrals through the core SLOs you have already established. With the customer starting to get delighted and probably referring you to their friends, it is not the time to relax. You want to be tracking the **engagement metrics** to see that your product is still top-of-the-mind for them in a highly competitive market. That is where you need to have your CRM tools measure the engagement metrics and feed them back to the engineering and other teams. Engineering platforms should use that data to enable the product developers to develop strategies around fewer disruptions and give the developer team the confidence to increase their commitment to the newer features and slowly work towards making your products indispensable for your customers.

Yet again, we summarize the discussion above in an easy-to-refer table below in Table 3.6.

**Table 3.6 Customer Retention Lagging vs Platform Leading Indicators**

	Leading indicators			
Lagging Indicators	Adoption Rate	MTTR	SLOs	Sentiment
Customer Lifetime Value	Increased	Reduces	Aligned	Retention
Net Promoter Score	Higher	High MTBF	Reliability	Trust
Renewal rates	Continued use	Minimize disruption	Trust the SLOs	Confidence
Referrals	Higher value	Quality of resolution	Incentives	Word of mouth
Engagement Metrics	Increased engagement	Fewer disruptions	Increased commitment for new features	More time with the product

### 3.6.4 Revenue Growth KPIs

So far, you have improved the developer experience, tracked and helped with customer experience, and made a case for customer retention by using engineering platforms. Great, but how are you, as the engineering platform leader, assisting the business with its most important metric? - Continued revenue growth keeps you in the business - During the macroeconomic slowdown in 2020, many intelligent, forward-looking business leaders doubled down on strategic investments in EP because they understood its strategic importance. However, some other businesses deprioritized these efforts because they needed more awareness regarding the benefits of revenue growth.

Our five lagging indicators are *total revenue, profit margins, revenue from new vs existing customers, revenue predictability, and market share*.

As discussed earlier, assuming your revenue realization models are set correctly, higher adoption rates drive higher revenue. Such an experience requires the lowest possible time to resolve the issue, adhere to the SLAs, and build brand loyalty. All of these can only happen with a solid engineering platform that lets the developers and the product managers reduce their cognitive load to focus on the most critical aspects. As the adoption increases, you are looking for more streamlined workflows to remove potential inefficiencies. A prime example of such inefficiency is the manual intervention required for customer authentication, which your engineering platform can promptly address by offering a standard set of capabilities consistently used across all product developers. You do have the expectation now that the problem resolution is fast and the mean time between failures is very high. As consistency comes to the fore with your objectives, customers are ready to include your product as they build their plans.

Business leaders struggle with building more predictability even during the best times, so no wonder this keeps them up at night during difficult times. Providing increased collaboration on real-time adoption needs through an extensible engineering platform will be highly valuable. You would now have to use the EP to demonstrate the trends of lower MTTR quarter over quarter and a positive trend around SLOs. You can plug historical sentiment data into your observability platform to correlate it with the rest of the engineering metrics. Improving the market share requires highly efficient adoption, lower, minimal, and predictable MTTR, and your SLOs to be better than your competitors. You can achieve all these by using the engineering platforms to make your engineering processes more consistent, reliable, scalable, and efficient.

The final summary table, Table 3.7, below captures the impact of engineering platforms on the lagging indicators that impact revenue growth.

**Table 3.7 Revenue Growth Lagging vs Platform Leading Indicators**

		Leading indicators			
Lagging Indicators	Adoption Rate	MTTR	SLOs	Sentiment	
Total Revenue	Higher	Low	Adhere	Loyalty	
Profit Margins	Streamlined workflows	Fast resolution and improved MTBF	Consistency	Building their future	
Revenue from New vs Existing customers	Innovation	More self-serve/self-heal solutions	Published track record	Thought leadership	
Predictability	Collaborate through EP automation	Trends of lowering MTTR	Positive trending	Historical data	
Market Share	Higher efficient adoption	Lower, minimal, predictable MTTR	Better than competitors	Betting their future	

### 3.6.5 Exercise 3.4: Establish a set of KPIs and measure for success at PETech

In this exercise, you will create a simple mapping of your platform capabilities to the business outcomes as explained in Section 3.5 of this chapter.

You will need to do the following

1. Identify a set of technical outcomes you are interested in achieving. An example of a technical outcome could be *faster application startup*.
2. For each of the technical outcomes you have identified, identify one or more platform capabilities that can either *enable* or *contribute* to the intended outcome. An example of a platform capability that can help achieve the *faster application startup* outcome could be building a *scalable CI/CD system*.
3. In the next step, map your technical outcomes to the business drivers. An example of a business driver could be *improved developer experience if the application starts up faster*.
4. In the last step of the exercise identify the business outcome. Your business outcome could be any of the *lagging indicators* we introduced in section 3.5 or could be something that is more contextually relevant to your business.

### 3.7 Summary

- We should look beyond just revenue as a success metric and also consider employee morale and other factors in platform engineering.
- To determine if our initiatives are effective, we need both objective and subjective assessments, as proving results shows the value of our investments.
- We need to set measurement criteria at both the domain level and throughout the SDLC process to get a detailed view of platform engineering metrics.
- By tying measurements to core platform principles, we can assess success across different areas, promoting a product mindset and using tools like end-to-end observability and self-healing mechanisms.
- Simple models and value stream mapping help us quantify returns and justify investments in platform engineering.
- Reducing cognitive load allows developers to focus on high-value tasks, using strategies like simplifying problem domains and promoting data accessibility through observability.
- A capability model for platform engineering streamlines measurement and improves the developer experience.
- Corporate culture has a big impact on platform engineering, so managing cultural change is essential for successful adoption.
- For effective platform engineering, data-driven improvements and scalable practices like structured automation and cloud-native technologies are crucial.
- We need to define Key Performance Indicators (KPIs) that align with the platform's operating model to measure the platform's value effectively.

# ***4 Governance, Compliance and Trust***

## **This chapter covers**

- The layers of Governance, Compliance, and Trust
- Compliance at the point of change
- Policy as Code
- The Software Supply Chain
- Identities in the Engineering Platform

One of the mistakes we'll often see made is the assumption that engineering platforms come with developer freedom automatically. That by simply building or buying a platform, the platform's users are immediately unblocked and free to deploy new software into production.

The reality is quite different. Each area of delivering software still requires strict adherence to the domains of compliance, audit, security, and data. Teams like Infosec, Security, Infra, Audit, and Governance are not going to disappear.

Each of them has different requirements. Consider for example the following:

- Our security team requires that there are no exploitable vulnerabilities rated "CRITICAL" deployed to production.
- The audit team wants evidence of every deployment logged into an immutable datastore to meet SOX compliance.
- Infosec has stated they need to enforce the use of Authz policies on sensitive dataset apis, to ensure only authorized users have access.

These requirements are similar to ones you would find in any environment. But if we hand the keys to each of these teams for enforcement, we will return to creating roadblocks and friction for our developers, going against the goals of our engineering platform.

From just these 3 examples, we can see that the complexity of building an engineering platform that truly frees our developers up is much higher than originally anticipated. We'll need to apply some new patterns and ways of thinking to overcome them and provide a positive experience for our development teams.

## 4.1 Autonomy and Policy as Code

The focus on autonomy may be self-evident, but there's also some science behind why it's such an important aspect of our platform.

In 1979, two brothers named Stuart and Hubert Dreyfus conducted a study of mastery on airline pilots. They had a set of expert and instructor pilots make a checklist for novice pilots to follow in an emergency simulation. When the novices used the checklist, their performance was markedly improved. When the experts used their own checklist, their performance suffered when compared to running the simulation with no limits and full autonomy.

TABLE 1

Skill Level Mental Function	NOVICE	COMPETENT	PROFICIENT	EXPERT	MASTER
Recollection	Non-situational	Situational	Situational	Situational	Situational
Recognition	Decomposed	Decomposed	Holistic	Holistic	Holistic
Decision	Analytical	Analytical	Analytical	Intuitive	Intuitive
Awareness	Monitoring	Monitoring	Monitoring	Monitoring	Absorbed

**Figure 4.1 The original table from Dreyfus and Dreyfus paper (Dreyfus, Stuart E.; Dreyfus, Hubert L., A Five-Stage Model of the Mental Activities Involved in Directed Skill Acquisition, p19)**

The brothers concluded that the experts had reached a level of mastery commonly referred to as intuition. Using their findings, they defined the "Dreyfus Skill Model" as shown in Figure 4.1. You'll notice that at the levels of *Expert* and *Master* the Decision Making function has moved to *Intuitive*. This is similar to how a master chef can make a dish from scratch with no direction. And when asked to explain it step by step they can't, they made it on intuition alone. Or when a Doctor walks into the room and accurately predicts the diagnosis within 5 minutes of seeing the patient.

As humans, we develop intuition in directed skills over time. We pick up on very subtle cues (many of which we aren't directly aware of, it's subconscious thinking) that tip us off to the task at hand.

As another example, a master hockey player skating down the ice sees very tiny movements and minuscule shifts in weight from the defenders in front of him, and he instinctively knows which movement to make to counter them. A novice or intermediate player barely has time to look up, let alone analyze the movements of the players in front of him. But when given a set play or pattern, novices are often taught to “skate to the outside and then cut inward”, they have increased success. The best players are often defined by their “Hockey IQ.” This comes down to their ability to make incredibly fast decisions in fractions of a second, being in the right place at the right time or placing the puck in the perfect spot. Novice players are taught a simple checklist: Scan, Ask, Act. They “scan” the ice, they “ask” themselves what is about to happen and what-if they went somewhere, and they “act” on their questions. Over years and years of training, this list becomes instinctual. Master players don’t use the checklist in their head, because they are doing it constantly in their subconscious, every moment is analyzed and actioned on. If placed on the same sheet of ice, the master intuitively knows where to be at all times, while the novice is still on step one. If we forced them to follow the list audibly (or mentally), it would slow them down tremendously.

The purpose of outlining this study and examples, is to show the reader that forcing experts and masters to follow a script will lead to reduced performance (or worse, they’ll just find a new place to work). We have to provide a platform that helps the novices get off the ground (for example, with starter kits) and allows the experts to quickly make design decisions without being slowed down by the requirements of policies across the org.

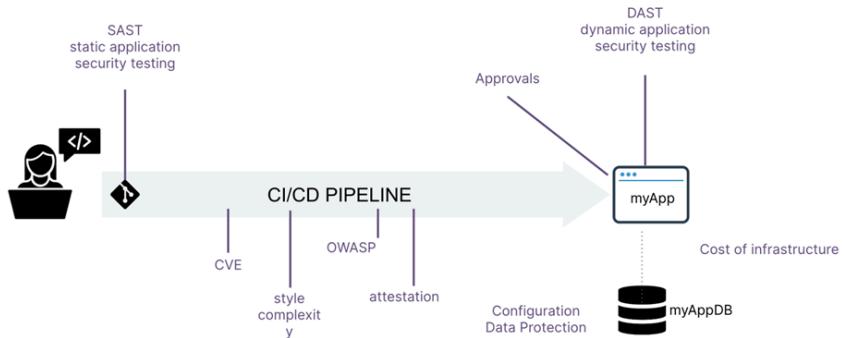
Our experts intuitively know what changes and design decisions to make in our software codebases. And often our senior-most experts move around from project to project, helping teams improve their codebase (i.e. SREs). If they are given the freedom to make those changes without hundreds of hurdles from policy-focused teams, the impact is huge on the org.

#### **4.1.1 What does it mean to make a development team autonomous?**

In the legacy environment of PETech, teams that needed simple things like an S3 bucket, database access, and even deploying their application to production faced a mountain of tickets, manual requests, phone calls, and maybe a smoke signal. So we’ve decided to take this journey to remove much of that friction for our development teams. To do this, we must analyze what areas affect our teams the most in their day-to-day engineering processes.

The most utilized point between engineering and the platform is the development pipeline. As we pointed out in the previous section, taking control of the pipeline away from the development teams is a poor idea. We want to give teams complete control over their own pipeline. Typically, a pipeline looks like figure 4.2:

## The *Everyday Context*

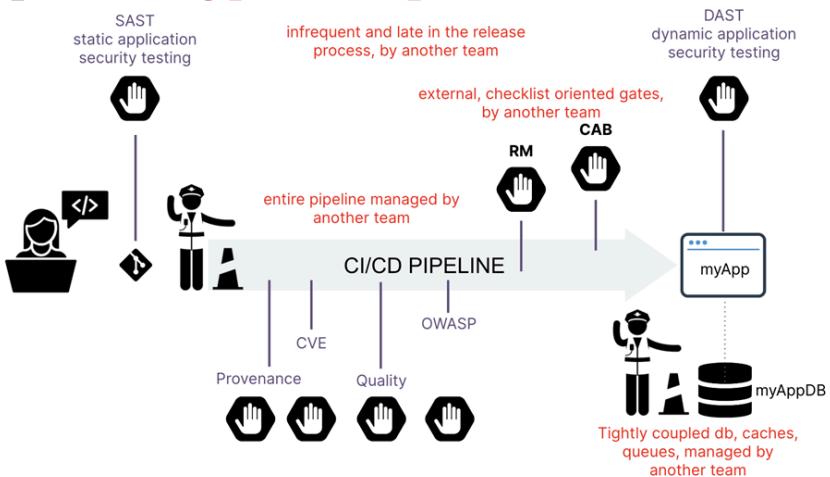


**Figure 4.2 The Everyday Pipeline**

And this is a fairly normal pipeline. We are scanning for CVEs and OWASP vulnerabilities, we are running static analysis and provenance on our code, and maybe we're also doing code quality analysis.

In the legacy world of PETech this same pipeline looks more like figure 4.3 in reality:

## Compliance: Typical Response



**Figure 4.3 The reality of the everyday pipeline**

Each step of the pipeline is often controlled by a different team. This leads to bottlenecks and friction between our development and security, audit, and compliance teams.

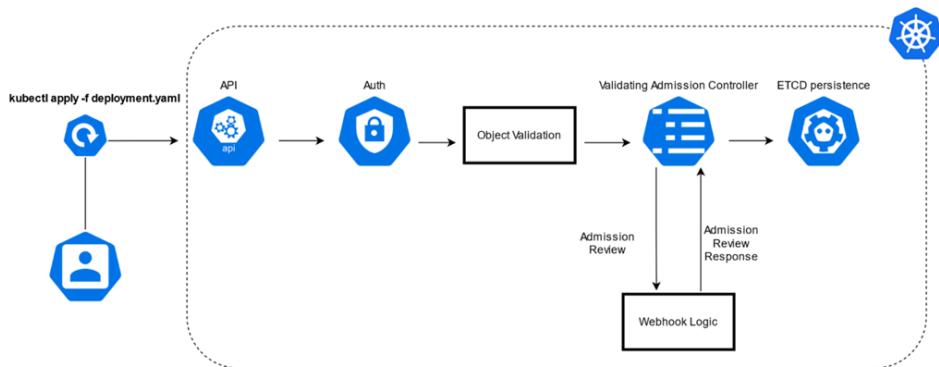
To remove this friction, we need to split off part of the work of pipelines, specifically #2:

1. doing the work of compliance
2. verifying that the work of compliance was done

To accomplish this, we move #2 to the end, at the point of deployment. At Thoughtworks, we often call this the “Point of Change.” We call it this because we are changing the deployment in our target environment. So the point of change is the boundary between the deployment pipeline and the target environment.

To move verification to the point of change, we’ll need to introduce some new concepts. Kubernetes has a very mature way of handling this, currently referred to as an Admission Controller.

## Admission Controller



**Figure 4.4 Kubernetes Admission Controller**

The admission controller API allows us to create validating webhooks at the boundary of the environment. This successfully meets our criteria of decoupling the work of compliance from the verification of compliance. This is because our security team can now own the policies deployed to the admission controller, instead of in the development pipeline.

For example, let's consider code provenance. Our development team is now responsible for signing their code. The security team has declared it must be signed with a specific key and published to at least one of several SBOM format options. The development team chooses to publish their signing with Cosign and SBOM in the SPDX format using CycloneDX. They may only choose to do this step at the very end of their pipeline process, so if it fails, they aren't prevented from iterating on a feature and working on getting things like tests passed. Once ready, they enable the feature to deploy and send it to the Kubernetes environment. Our Admission controller will fetch the signing key and SBOM artifacts from the development team's specified locations (such as an image registry) and verify or reject the application.

Consider figure 4.5 as a very simple example, where we are enforcing the presence of certain labels on all deployments. While that sounds very trivial, this is actually an important gate. Most cost-optimization and tracking softwares depend on specific tags in order to do their work.

```

apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8srequiredlabels
spec:
  crd:
    spec:
      names:
        kind: K8sRequiredLabels
        listKind: K8sRequiredLabelsList
        plural: k8srequiredlabels
        singular: k8srequiredlabels
      validation:
        # Schema for the `parameters` field
        openAPIV3Schema:
          properties:
            labels:
              type: array
              items: string
      targets:
        - target: admission.k8s.gatekeeper.sh
          rego: |
            package k8srequiredlabels

            deny[{"msg": msg, "details": {"missing_labels": missing}}] {
              provided := {label | input.review.object.metadata.labels[label]}
              required := {label | label := input.parameters.labels[_]}
              missing := required - provided
              count(missing) > 0
              msg := sprintf("you must provide labels: %v", [missing])
            }

```

**Figure 4.5 Admission controller that checks for labels on the deployment**

Here we are going to ACCEPT or DENY the deployment based on the presence of the labels object in a Kubernetes deployment. We might further enhance this policy by specifying which labels need to be present in the labels object.

### 4.1.2 Policy-as-code

Let's say that the CTO has declared all deployed code must now have 90% or higher test coverage. In the old way of doing things, the DevOps team would code up a script or pipeline orb and add it to the standard development pipeline. But no DevOps team owns the pipelines in our new world of Platform Engineering. Instead, every development team owns their pipeline. And while our CTO trusts his development org, he wants to keep everyone honest without taking control of their pipelines. We do this with our Compliance at the Point of Change Admission Controllers we talked about before. But to implement our admission controllers, we'll need a policy engine to do it.

You may have noticed in figure 4.5, a funny looking block of code under the `rego` key. This code is written in a language called `rego`. It is the language used to interact with Open Policy Agent, a graduated CNCF project that provides a declarative policy engine. There are other options in this field, such as Kyverno (<https://kyverno.io/>), however we are going to focus on Rego and OPA, as these solutions are the most flexible and fit into ecosystems outside of Kubernetes.

### OPEN POLICY AGENT

OPA, or Open Policy Agent, allows us to write admission controllers as we described in 4.1.2. This works by handing our admission decisions to the OPA agent, where specified policies are given the event data, to which they will output a true for admitted or approved or a false for rejected or not approved to deploy.

When working in `rego`, it's easy to misstep and create additional friction if you're not careful because Rego is a declarative language. We have seen many clients and engineers trip over Rego and make early mistakes, mainly due to a gap in understanding.

Rego is a declarative language. Similar to Terraform, this means that the language does not provide a series of steps that are to be performed. For example, in Java, you might tell the computer to print with `System.out.println()`, where the subject is explicitly called as `System`. In `rego`, the desired results are described within the boundaries of a specific domain. Such as in Terraform, the domain is the creation of infrastructure. So, a declarative terraform statement might describe the aspects of a (desired) server: 2 CPUs, 4 GB memory, and so on.

In Rego and OPA, our domain (also known as the *input*) is the assertion of policy. We assert policies by writing predicates expressed over data.

*Bryan is writing a book*

Remember that predicates are properties, or descriptions, of their subjects in grammar. Now, let's transition this understanding into simple mathematics. If we replace "Bryan" with X, we have:

*X is writing a book*

What is X? It's a predicate variable. Using our new semi-equation, we can define a *domain*. Our domain is Nic, Ajay, Bryan, Sean, and Brandon. Then we might say:

*For all values X where X is one of [Nic, Ajay, Bryan, Sean, Brandon]*

*X is writing a book*

The output of this assertion...is False! Nic, Ajay, Bryan, and Sean are writing a book. Brandon is not. So, given the *domain* of X is equal to those five values, our predicate is false. To be clear, Brandon **is** part of our domain. And when Brandon is in the domain, our predicate is false.

This helps us arrive at a helpful definition for predicates - it is a statement that contains variables, and it may be true or false depending on the values (also known as the *domain*) of these variables. Consider an equation like:

$$F(y) = 2y \text{ is greater than } y$$

In reading this predicate, you may notice that this statement's potential "output" or value is quite binary, either true or false! If we set y to -1, our predicate would be false (because -2 is not greater than -1). If we set y to 1, our predicate would be true (2 is greater than 1).

Let's look at another example, but now use Rego. First, let's describe our predicate in English.

*There exists an x equal to 12*

Standalone, this expression means absolutely nothing. But let's give it some context. In OPA and Rego, inputs (*i.e.*, *domain*) are provided in JSON. So, we'll pass a straightforward input:

```
{
  "Values": [1, 2, 12]
}
```

Then, let's convert our predicate statement into valid Rego:

```
our_policy {
  some x
  input.values[x] == 12
}
```

If we evaluate this expression with our input using Open Policy Agent, we'll get the following response from our OPA evaluation:

```
{
  "our_policy": true,
}
```

It's essential to understand and remember that OPA will not return anything for a false predicate unless it is first declared with a default. This will come in handy when you do the upcoming exercise.

Remember our discussion earlier where we looked at predicates in English? Well, this policy says that for some Value of X, there exists a 12. Given the domain of [1, 2, 12], the predicate is factual! This is exactly like our example where our Predicate was false when the domain had Brandon, but we said for all domain values in that example.

Now, if we added the following line to our policy

```
package policy

import future.keywords.if
import future.keywords.in

our_policy if {
    some x in input.values
    x == 12
}

our_negated_policy if {
    not our_policy
}
```

Our response would be:

```
{
    "our_policy": true,
}
```

What happened to *our\_negated\_policy*?

We need to create a default expression for it to get it back:

```
package policy
default our_negated_policy := false
import future.keywords.if
import future.keywords.in

our_policy if {
    some x in input.values
    x == 12
}

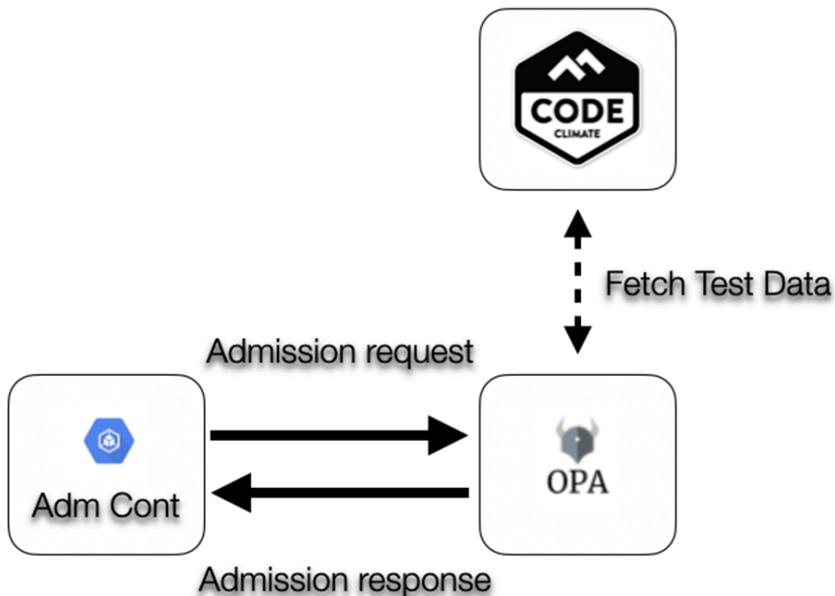
our_negated_policy if {
    not our_policy
}
```

Now, OPA will evaluated to:

```
{
  "our_negated_policy": false,
  "our_policy": true
}
```

Now that we've better understood Rego, let's return to using it with an admission controller, as discussed in the previous section.

To understand how this works, we start with a deployment request that gets sent to our Admission Controller. Our Admission Controller then makes an admission request to our Open Policy Agent. Our Open Policy Agent fetches the applications unit test data from our testing tool, parses the information, determines if this deployment meets our unit test standards, and returns an allow or deny to the admission controller, as we see in figure 4.6.



**Figure 4.6 Example of a Kubernetes Admission Controller**

To make things simpler, we'll use the Rego Playground; this will help us “mock” the policy engine agent, as well as the CircleCI response data. In production, you'll have OPA running in your environment, calling out to CircleCI's API.

**Listing 4.1**

```
{  
    "kind": "AdmissionReview",  
    "request": {  
        "kind": {  
            "kind": "Pod",  
            "version": "v1"  
        },  
        "object": {  
            "metadata": {  
                "name": "myapp"  
            },  
            "spec": {  
                "containers": [  
                    {  
                        "image": "hooli.com/nginx:3d9ae9e",  
                        "name": "nginx-frontend"  
                    },  
                    {  
                        "image": "hooli.com/mysql:4e8tE32",  
                        "name": "mysql-backend"  
                    }  
                ]  
            }  
        }  
    }  
}
```

This is our Input. AdmissionReview inputs are the inputs generated when you use OPA as an admission controller.

**Listing 4.2**

```
{
    "coverage_required": 80,
    "scans_api": {
        "hooli.com/nginx:3d9ae9e": {
            "unit_test_coverage": 91,
            "tests_failing": 0
        },
        "hooli.com/mysql:4e8tE32": {
            "unit_test_coverage": 80,
            "tests_failing": 0
        }
    }
}
```

This is our “api server” that we are mocking with the OPA playground data section. Now, our policy:

**Listing 4.3**

```
package kubernetes.validating.images

import future.keywords.contains
import future.keywords.if
import future.keywords.in

default coverage_above_required := false

deny contains msg if {
    not coverage_above_required
    some container in input.request.object.spec.containers
    not data.scans_api[container.image].unit_test_coverage > data.coverage_required
    msg := sprintf("Image '%v' does not meet the unit test coverage requirements",
    [container.image])
}
```

Let’s go through this step by step. We are using a new keyword here, `contains`. This boolean operator will append the message at the end of the policy to the output policy (in our case, named “`deny`”). So if the policy evaluates to true, we will see the string at the end of “`deny`” returned in an array-like object called `deny`. Reading the policy in plain English, you can read it as: Deny will contain the message “image X does not meet the unit test coverage requirements”, where x is the image being evaluated.

This policy iterates through each container in our deployment input and then asks our “API server” for the required coverage percentage and this image’s coverage percentage. If the image’s coverage percentage exceeds the required percentage, this policy will return true.

Note that this policy is not implicitly an AND operator. What we mean by that, is you could have three images in the input, and 2 of them might meet the coverage requirement but the 3rd does not. We simply construct a deny object where the message is appended for ALL resulting failures. This means that if two images fail the coverage requirement, there will be two messages in the denied output.

Finally, evaluating our policy in the sandbox we get:

```
{
  "coverage_above_required": false,
  "deny": [
    "Image 'hooli.com/mysql:4e8tE32' does not meet the unit test coverage
requirements"
  ]
}
```

Where our first image in the input, nginx, met the coverage requirements. But our second image, mysql, did not.

As another example, what if we just rolled out an istio service mesh and want to verify that all pod deployments to our environment are also creating a sidecar so they can adequately participate in the mesh?

#### **Listing 4.4**

```
package kubernetes.validating.istio

import future.keywords.contains
import future.keywords.if
import future.keywords.in

deny contains msg if {
  input.request.kind.kind == "Pod"
  not has_istio_sidecar
  msg := "does not have istio sidecar"
}

has_istio_sidecar if {
  some container in input.request.object.spec.containers
  startswith(container.image, "hooli.com/istio")
}
```

You can play more with predicates and Rego using the OPA playground here: <https://play.openpolicyagent.org/>

## **EXERCISE 4.2: WRITING AN ADMISSION POLICY TO VALIDATE THAT SSH PORTS ARE NOT EXPOSED**

Taking what you have learned so far, write a Kubernetes Admission policy that does something very practical: write an admission policy that looks at all of the exposed ports for deployment and makes sure that deployment isn't trying to expose an SSH port. When you write this policy(s), consider what Kubernetes objects you need to verify in your admission controllers. Hint: it's not just the Pod you need to work about!

### **4.1.3 Software Supply Chain Security**

As Platform Engineers, we must find ways to enable our development teams to adhere to the strictest and best practices of Software Supply Chain Security while still being able to do their work and release new features.

Around the end of 2020, you may recall a little attack aptly named "Solarwinds." If you aren't familiar with it, this was one of history's largest cybersecurity breaches. Thousands of organizations were affected, and the breach ran undetected for over a year. While the attackers ran all sorts of attacks and used many methods after a breach, the initial way that they got in was by finding a way to make changes to SolarWinds source code and appearing as if these were changes and commits that Solarwinds itself had made. Every Solarwinds customer then installed the malicious code during regular updates and patching, and the changes were trusted because the Solarwinds CI/CD process had signed them.

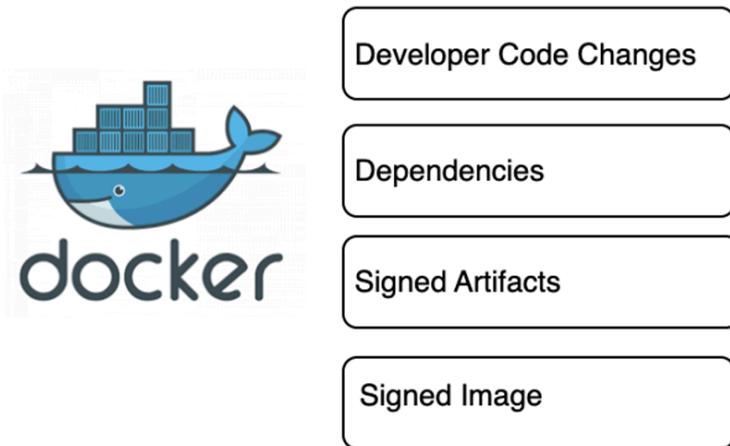
This breach sparked a massive industry-focused effort in Software Supply Chain Security. It also prompted the involvement of high-level government officials, including the President, who released Executive orders and funded over a billion dollars in State and Local cybersecurity programs.

Software Supply Chain has always been an important topic, but after this attack, the reality of its importance struck our entire industry deeply. It's important to understand how much easier it is to protect yourself from an attack than to root out an attack that has already occurred. So,

One of the most effective methods to provide Software Supply Chain Security is to apply some of the principles we have already learned about Policy as Code and Compliance at the Point of Change. Combining these principles with a zero-trust stance gives us a powerful and effective means to deliver Supply Chain Security to our development teams without producing unneeded friction.

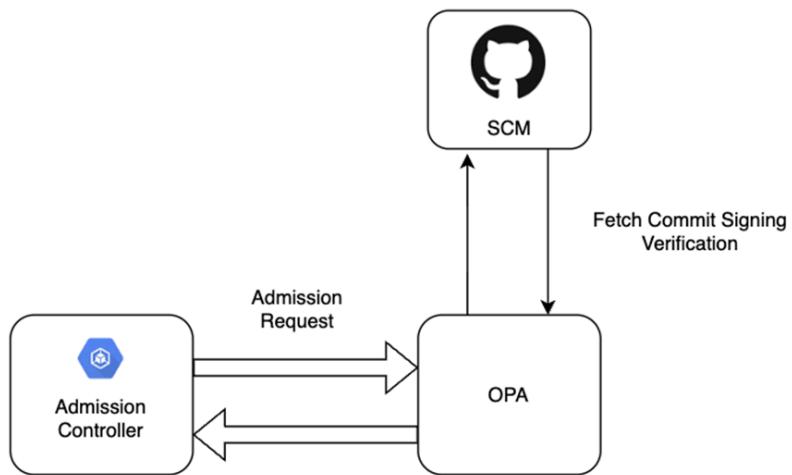
Let's take, for example, the provenance of a docker image release.

In figure 4.7, various layers contribute to the overall signed artifact. Firstly, the Developer Code changes. How do we ensure the integrity of those changes? We should require all our development teams to sign their commits automatically with a valid GPG key pair to do this. Our source code system that manages our Platform Code will enforce and verify that a valid developer key signs all code committed. Then, our dependencies and sign artifacts are pulled from a trusted registry that has already verified the integrity of the artifacts and dependencies. Lastly, at the end of our build process, the app team at PETech will sign the image they are releasing with their key (probably using a tool like Cosign or similar).



**Figure 4.7 Docker Images are made of many layers.**

In the legacy platform, all of these steps would have been owned and managed by the DevOps or Security team. But in our Engineering Platform, we have too many teams to have a central body controlling every one of these steps. Instead, we apply Compliance at the Point of Change to meet each requirement, by writing a policy that enforces our image policies, like we see in figure 4.8.



**Figure 4.8 Commit Signing is pulled from the SCM**

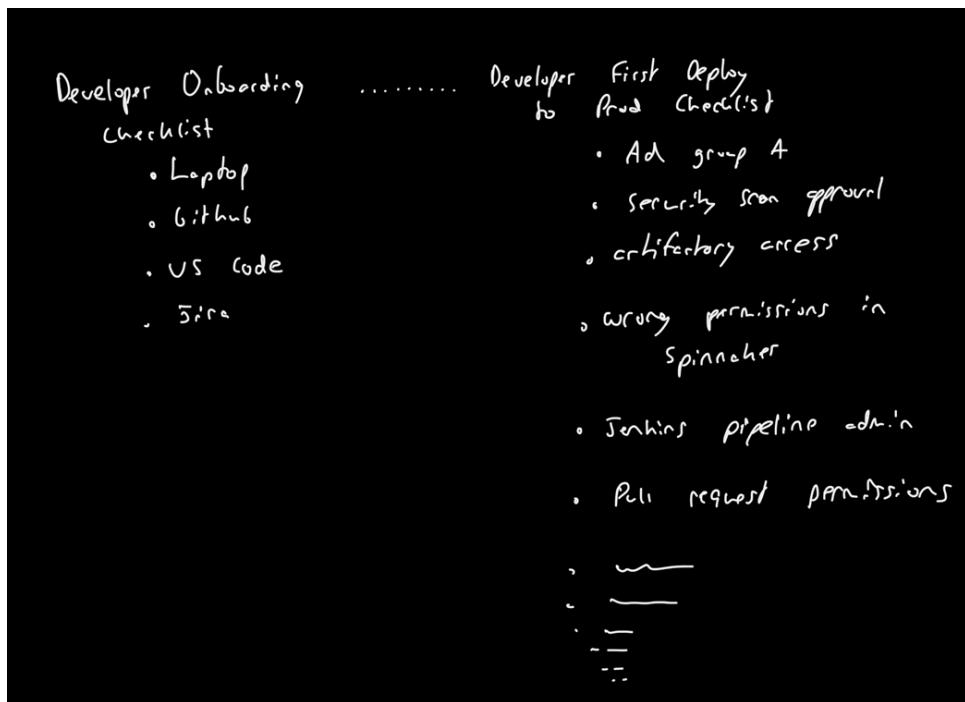
Taking commit signing as an example, this policy is exactly like our example in 3.3.3 where we had a policy verifying that our code coverage met a certain threshold. Applying this type of policy to verify each layer of our images gets us closer and closer to a zero-trust approach to supply chain security.

#### 4.1.4 Exercise 4.2: Create a Co-sign Policy to allow Signed Dockerhub Images

Now that we've talked about how to write policies, use them with compliance at the point of change, and the importance of supply chain security, apply what you've learned and write a zero trust policy that enforces our requirements and can be reused as a Compliance controller. To do this, you'll need to use a tool called Cosign. Cosign is a project managed by Sigstore, a governing body within the OpenSSF (Open Source Security Foundation).

Also, consider how you might write a policy that enforces our image security concerns, from code signing to image signing.

Note: You can learn about Software Supply Chain security in great detail by learning more about the projects within Sigstore and the OpenSSF. The first area we need to consider is the most obvious yet difficult to get right: team onboarding. The reason for this is simple: when we expand our definition of “onboarded” to include the ability to deploy to production, our metric for success in most organizations drops off a cliff. In fact, in analyzing the platform mechanisms of many companies, we've found that “team initialization” and “team deployed to prod” are two of the worst-performing metrics for most organizations regarding cloud and cloud-native engineering disciplines. So, what do we mean by team initialization?



**Figure 4.9 “Onboarding” vs Time to First Deployment Reality. The number of steps involved causes this time to be long when manual steps and cross-functional check-ins are required.**

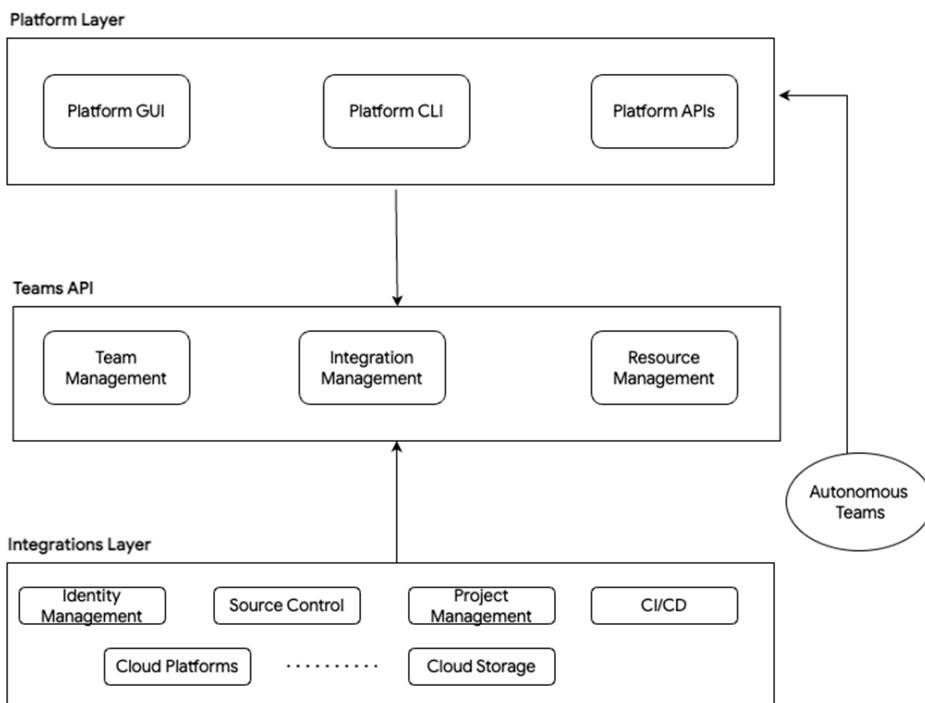
Consider that in an Engineering Platform, we plan to enable our teams to be autonomous. This goal is true of many organizations that have adopted cloud or platform thinking. But to achieve “autonomy,” we must provide teams with a means to draw a circle around themselves and the objects within their realm.

They own their repositories, applications, files, drives, scrum boards, etc. The point is that every team must initialize by drawing a circle around these artifacts and gaining access to them in some fashion. And at many organizations, this process is excruciating. To define our team, we may need some form of identity group in the company’s user account system, say Active Directory or something similar. This will probably require a ticket. Then, once we have our identity, we need to get our code repositories, Jira board, cloud file share, namespaces in the cloud platform that we can deploy to, pipelines...etc. A different team controls each of these artifacts. The IT Ops team maintains the source control system, the Jira projects by the PMO, the Security team owns the pipelines, our storage team shares the files...and on and on we go.

What if each system was connected to our platform and given to our new teams at creation? And we aren’t just talking about the usual response to friction here, i.e., the typical response to “things are slow! Automate them!” We mean creating meaningful connections between the platform and these connected systems via APIs, CLIs, and Platform GUIs. We want our development teams to interact *with the platform* to create them. This consolidates the experience of our Platform interfaces and creates a standard pattern for enablement that teams come to expect, making onboarding to new features of the platform just as efficient.

We can achieve this without removing control of these systems from the teams we mentioned earlier. The security team can still have administrative rights to CI systems, the PMO can maintain Jira, and the IT Ops team can still manage Github. But we now want to have each of these teams connect their tools to the platform via integration, and our platform will abstract those integrations via meaningful platform APIs.

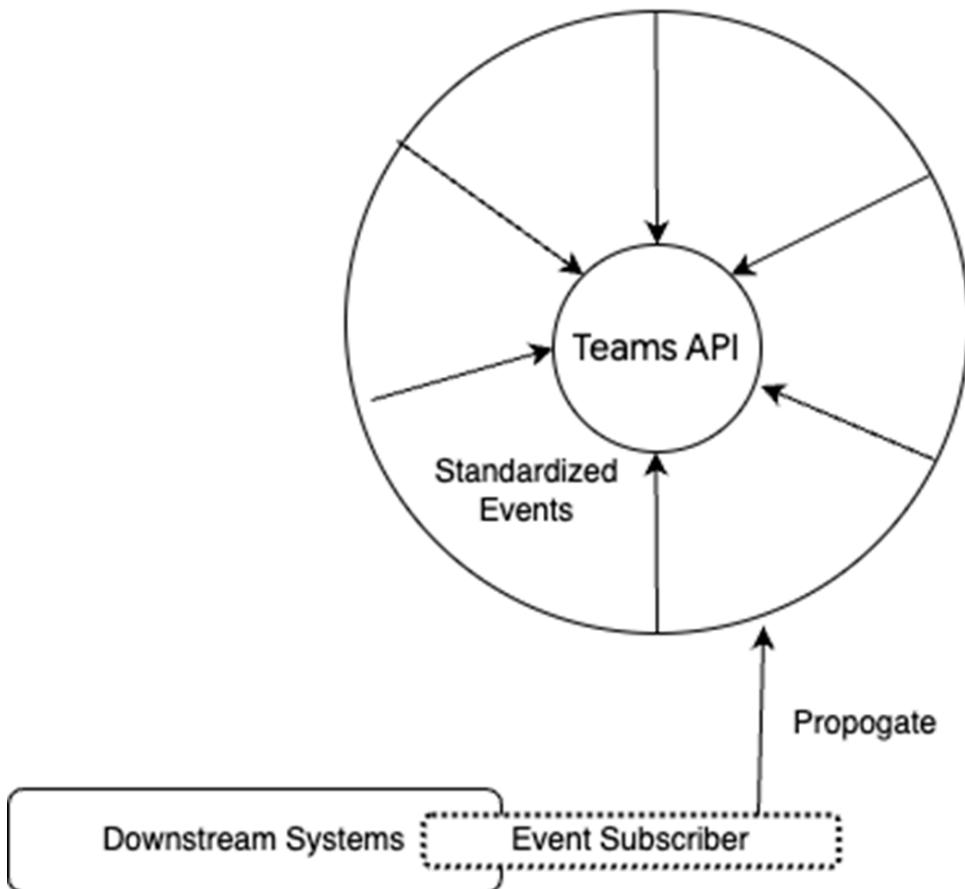
So, how do we provide these connections in a repeatable and well-architected manner? To do this, we propose the concept of a Teams API. The function of this API is simple: teams, their members, and their integrations are defined and interacted with via the Teams API as shown in figure 4.10.



**Figure 4.10 Shows the high level concept of Teams API with the platform layer and integrations layer, and the separation of concerns from cloud vendor details.**

From this, we realize that our API should drive forward many systems we intend to interact with as a team. But instead of talking to them directly, we enable each downstream system to integrate with the team. The reason for this is simple: they *are* integrations with teams. So why don't we treat them as such?

Since each of these downstream systems and integrations has its API and state, we interact with them in an event-driven way. When our team API requests a new team integration, we propagate this change to the subscribed systems via a standardized event. This allows us to write a well-specified subscriber and makes adding new integrations and downstream systems a routine and simplified pattern.



**Figure 4.11 shows how the downstream systems can propagate the events through a subscriber mechanism**

#### 4.1.5 Separating compliance work from verification

Now that we've established our desired state and would like to provide a consistent team experience with self-service for our platform's downstream functions and features, another challenge arises! Our security team is worried that they no longer have control or visibility over what is getting into production. Your security team lead wants to know how to stop vulnerable code from getting to production, especially since the security team is no longer in the direct path regarding secure coding practices!

We must discuss a new engineering pattern we developed at Thoughtworks to fit this requirement. This is known as **Compliance at the Point of Change**. Let's consider a traditional pipeline:

## The *Everyday Context*

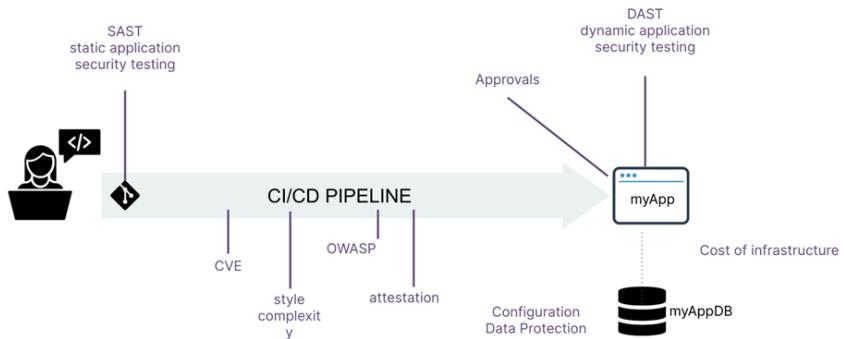


Figure 4.12 The Everyday Pipeline

This is all well and good. Looks nice, right? What happens when we apply all of the usual guardrails that an enterprise requires to function securely?

## Compliance: Typical Response

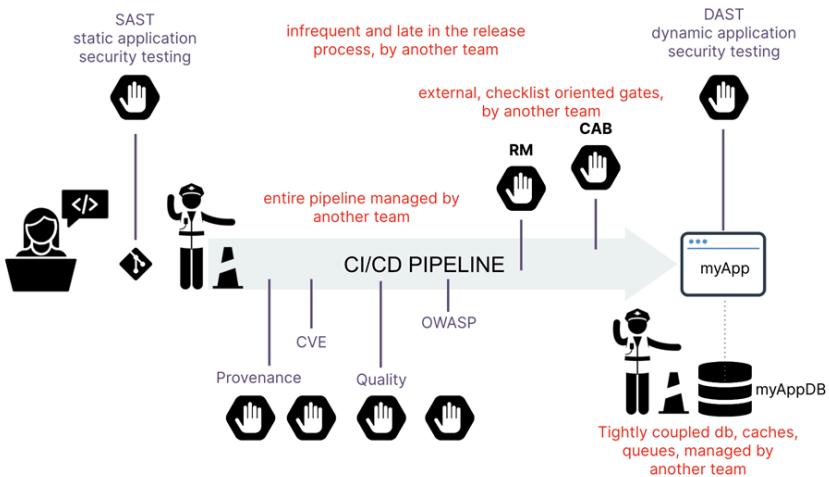


Figure 4.13 The reality of the everyday pipeline

Suddenly, you feel a little sick, right? This is representative of the typical pipeline process we see at large organizations. Security and compliance teams owning most of these functions leads to bottlenecks and tickets across the board. Let's consider code provenance, for example.

At PETech, we were told by the CISO that Solarwinds is something we must all learn from, and the security team will now be adding code provenance into every step and package of our development process. Now, you as a new Platform Engineer, wonder if this is the type of function that could be enabled via self-service, but the security team isn't convinced. How do we ensure all development teams comply if we don't own this step in the pipelines, says the security team.

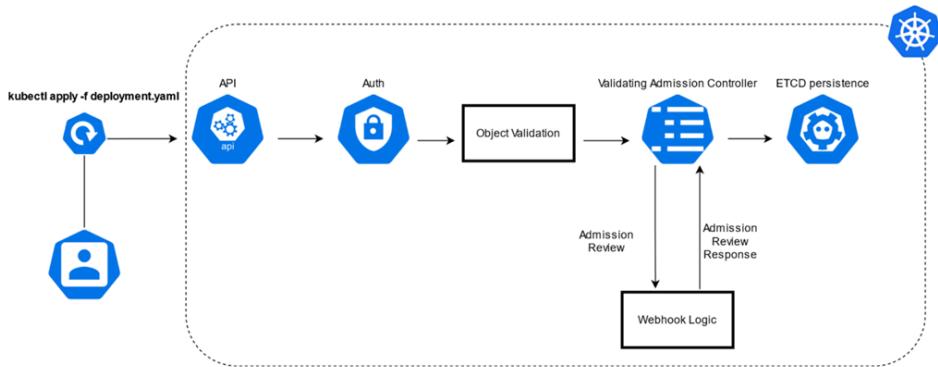
To accomplish this, we have to decouple the two actions required:

1. doing the work of compliance
2. verifying that the work of compliance was done (correctly)

What is significant about decoupling these two actions is it allows our security team to own step #2 and our developers to own #1! If our development teams are responsible for their pipelines, they are responsible for doing the compliance work at some stage in their development process. And if our security team is only responsible for #2, they only need to define the policies development teams must pass to deploy their software.

In a Kubernetes-enabled environment, we would do this via what is called an Admission Controller.

## Admission Controller



**Figure 4.14 Kubernetes Admission Controller**

The admission controller API allows us to create validating hooks at the boundary of the environment. This successfully meets our criteria of decoupling the work of compliance from the verification of compliance. Our security team can own the policies deployed to our admission controllers in Kubernetes, and our developers can work on compliance in their pipelines, knowing it will be verified when they try to deploy and pass the admission controller gate.

Let's go back to our example of code provenance. Our development team is now responsible for signing their code. The security team has declared it must be signed with a specific key and publish at least one of several SBOM format options. The development team chooses to publish their signing with Cosign and SBOM in the SPDX format using CycloneDX. They may only choose to do this step at the very end of their pipeline process, so if it fails, they aren't prevented from iterating on a feature and working on getting things like tests passed. Once ready, they enable the feature to deploy and send it to the Kubernetes environment. Our Admission controller will fetch the signing key and SBOM artifacts from the development team's specified locations in the standardized deployment object and verify or reject the application.

To implement this in a non-kubernetes environment, one needs only to implement a change api in front of the target deployment system. Effectively, develop an API proxy in front of the deployments API of your target environment. Then, write the logic necessary to evaluate that deployment, prior to passing it along to the downstream deployment target.

In summary, to enable self-service deployments, separate compliance tasks from their verification. Shift compliance checks to the point of change, moving verification to your environment boundaries. If you're using Kubernetes, admission controllers are key; for other platforms, implementing an API-driven approach can achieve similar results.

## 4.2 Point of Change Security

In the context of a particular domain and communication layer, **0-trust** means applying a default DENY policy between all systems and components that fall within that domain+layer, and applying explicit one-to-one ALLOW policies to establish trust between those components on a component-to-component basis.

In applying 0-trust principles to our platform, specifically to the target environments, we are further enhancing our point of change security. There are multiple ways to adopt zero trust security:

- Networking
  - OSI Model Layer 3 & 4
  - OSI Model Layer 7
- Authentication & Authorization
  - At the boundary
  - On every request

### 4.2.1 ZTA, Networking

At PETech, our CISO has mandated that only authorized services can talk to our critical credit card and account processing APIs. Everything else should be blocked or dropped upon request when trying to talk to those apis.

There's a number of ways our Platform team at PETech can handle this.

At Layer 7, we have 2 options for blocking the requests as our CISO has demanded. We can use Authentication and Authorization tactics. For example, in Istio we can define Authentication and Authorization policies, in which we specify exactly which services are authenticated to talk to our apis, and how they are allowed to talk to them.

Hard-coding those policies into istio authn/authz is quite cumbersome, however, and not very scalable. If you recall our sections previous on Open Policy Agent, another handy feature of OPA is the ability to proxy requests for our apis

In the OPA enabled context, our request first gets processed by OPA before it's passed on to our workload container. This is done by installing OPA as a network proxy within the POD itself. It takes over all requests, by modifying the IPTables of the workload, to force all traffic to flow through its container when traffic is sent to our POD.

OPA then evaluates its installed policies for this specific pod configuration. Policies can be stored locally or in a remotely callable datastore.

Using web tokens, our OPA policy will validate the token, decode it, and use the token payload for verifying the request. The attributes of other request (in the token payload) can provide info about the caller, that OPA can use to verify if that application is allowed to make this call. OPA also can see the identity of the calling service, because it's sitting in the network layer of the POD.

What's more interesting, is because we are operating at Layer 7, we can make a better and more flexible architectural decision. Instead of hard-blocking all requests from specific services, we can use the attributes of the request to only block identities that are not allowed to make the request. Meaning, maybe only specific users can call our billing api. We can populate the web token with the identity of the user making the call, and specific attributes of that user. OOPA can use those attributes to make an informed decision about the call, and ALLOW or DENY it.

## DNS BASED ZTA

But let's say our CISO has also mandated that all external requests must be blocked, unless they are approved vendor urls.

We can handle this in a few ways. In our service mesh, we can default block all external requests. This is simple enough

```
meshConfig.outboundTrafficPolicy.mode = REGISTRY_ONLY
```

This setting is a simple default deny, that effectively means, unless a service is registered with our service mesh, we will block it by default.

To then add a service to our mesh, we simply create a ServiceEntry. We can make a simple ServiceEntry for google.com, which adds google.com to our list of services in Istio's service registry. This allows any workload to access google.com as if it were just another service in our service mesh.

There's another way to block external traffic, however. We can adopt L7 DNS aware CNIs. A CNI is a Container Network Interface. There are dozens of CNIs available, Calico, Cilium, Flannel, to name a few. CNIs that enable DNS based policies are typically eBPF based CNIs. These CNIs allow us to write DNS based policies that block all requests to DNS entries, except for explicitly allowed DNS entries.

```

matchLabels:
org: empire
toFQDNs:
- matchName: "api.github.com"

```

Cilium allows us to get a little bit more explicit with our policies. In the Istio case, we were setting policy for the entire cluster. In the Cilium example, we are enforcing the policy for a specific workload.

It might sound cumbersome to make policies for each workload, however this can be mitigated by applying policies to specific labels, instead of specific workloads. In figure 4.14, we are applying the policy to all workloads with the label 'org: empire'. This means that all pods with that label will have the same policy enforced when they try to submit external traffic.

## POLICY AT THE NETWORK LAYER

Let's say our CISO isn't satisfied with our Layer 7 policies. She's concerned that there are ways to get around them. How can we further harden our policies to prevent workarounds from occurring.

At the core network layers, we have Layer 3 and 4. We're going to focus on Layer 4.

At layer 4, we can explicitly modify the IPTables of our workload to ALLOW and DROP packets based on the source and destination of our request.

This approach, while powerful, is quite challenging to manage at scale. Imagine having 100s or thousands of services, and many thousands of network policies to manage as a result. This, as you can imagine, will get very complicated very fast.

However, there's a new api coming to Kubernetes that helps, the AdminNetworkPolicy api. This api allows us to apply more general policy apis on larger groups of services, making it very easy for us to write a policy that only allows explicit traffic to our billing services. We can then leave the rest of our workloads and services alone, we don't have to write policies for every workload in our environment in order to create a zero trust environment for our billing services.

## SUMMARY AND THOUGHTS

Whether you are applying zero trust with layer 7 or layer 4 policies, make sure the tradeoffs are considered before doing so. The overhead of applying these policies will create more work in the future. Any time a new service is added or modified that needs access to our restricted services, we now have to make changes in order to allow those connections.

There are other, highly secure, approaches to platform engineering that don't require the overhead of explicit network rules across our platform.

As we've discussed, things like code provenance, policy as code authz, non root containers, confidential computing, and more - all lend to a highly secure platform, without the burden of explicit network policies.

## A NOTE ON NETWORK PERFORMANCE

If you are familiar with lower level networking concepts, you may be wondering about the performance differences with IPTABLEs, NFTables, and eBPF. And more importantly, how each exists within the cloud native ecosystem.

Generally speaking, most Network interface specifications for cloud native platforms are moving towards NFTables based technologies as the standard. This requires significant work (cncf/k8s link) to move towards, but is making progress.

Notably, however, eBPF based interfaces and proxies are beginning to gain popularity. The advent of the eBPD "data plane" has reduced the number of network protocol steps a request has to take, which increases performance and reduces overall latency. That said, without significant experience in eBPF, we recommend going with the more widely used nftables approach unless a specific need for eBPF arises (such as the aforementioned Layer 7 and dns based policies).

### 4.2.2 Platform Team vs Platform Customer Identity

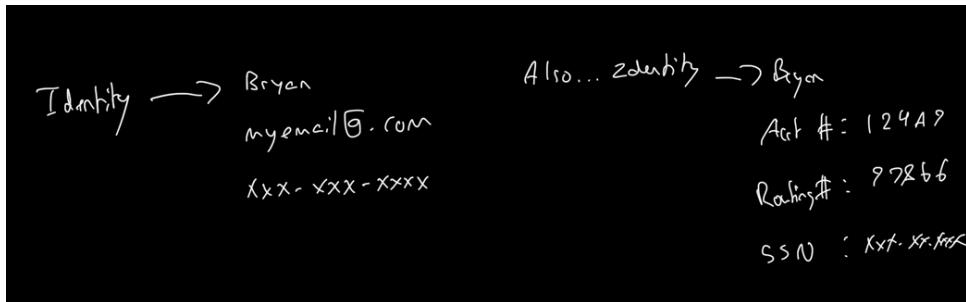
In modern engineering platforms, distinguishing between the Platform Team and the Platform Customer Identity is crucial for understanding the dynamics of platform interactions and integrations. The Platform Team refers to the group of individuals responsible for building, maintaining, and evolving the platform itself. They ensure the platform's reliability, scalability, and extensibility, and they create tools, APIs, and interfaces that other teams can use. On the other hand, the Platform Customer Identity represents the identity and permissions of the teams or individuals who utilize the platform's services and resources. These are the "customers" of the platform, often comprising various engineering teams within an organization that rely on the platform to build, deploy, and manage their applications. While the Platform Team focuses on enabling and empowering these customers by providing robust and seamless tools, the Platform Customer Identity is concerned with how these customers are recognized, authenticated, and authorized to interact with the platform's resources. This distinction underpins the architecture of platform solutions, ensuring that both the providers (Platform Team) and users (Platform Customer Identity) have clearly defined roles and responsibilities, leading to more efficient and secure platform operations.

When building the foundations of your platform at PETech, you may realize there's a fundamental missing identity. You want to make sure as Platform Builders you can do the things you need to do to build the platform, and you want to make sure the customers of your platform (i.e. application team developers) can do what they need to do to deploy to it.

### 4.2.3 Value of decoupling customer identity from infrastructure

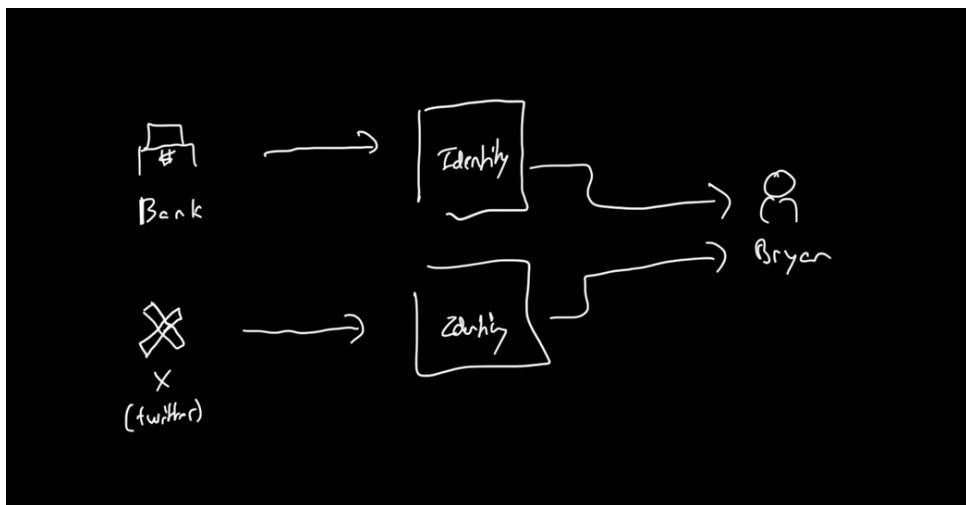
At PETech, you may be used to following a certain pattern for giving access to systems. When you need access to something, you file a ticket request to be added to a role/group, and a DevOps or security team approves it. But in building your engineering platform, you may realize that won't work. As the owners of all the systems and tools that make up an engineering platform, your small team can't possibly keep up with the hundreds or thousands of requests to access every day.

To accomplish this, we must provide a self-service means for teams to form and manage team members while staying compliant with our company's access policies. Thus, we don't give any customers direct access to any of our systems or infrastructure. We already have an identity system in-house, why would we want to replicate customer identities across our systems and infrastructure? Let's instead define a new way to think about identity in terms of *entities* and *identities*. To start, we define an *entity* as a single thing, an object, or a single individual. An example of an entity would be a user or a customer! An identity, then, is a combination of attributes that can be used to distinguish an entity in a specific context.



**Figure 4.15 An example of an identity decoupling**

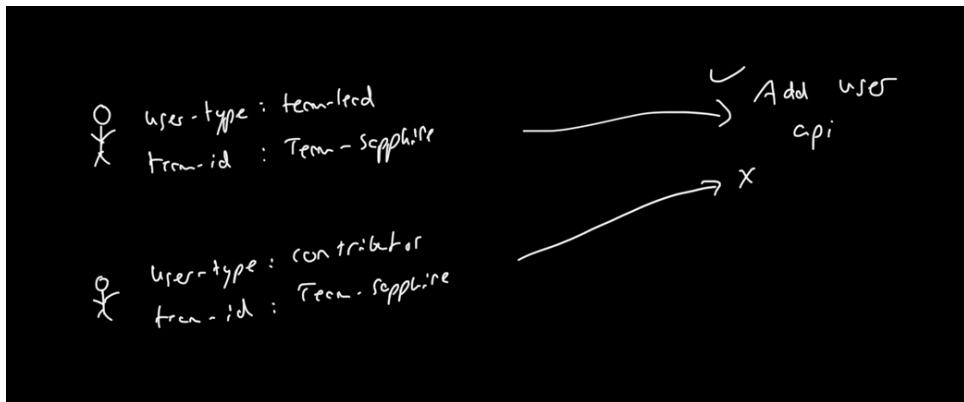
Take, for example, a social network. Bryan's identity as perceived by a social network, is his name, email, and phone number. But then consider his bank. Bryan's identity as perceived by the bank, is completely different, it's a combination of his name, account number, routing number, social, etc.



**Figure 4.16 An example of an identity evolution**

The point is the entity Bryan has many identities from the eyes of different systems. So, we can think of identities as the lens through which an application views an entity.

Much like the bank or the social network, our platform will grant team identities access to systems and infrastructure. And team leaders will be able to manage their teams with self-service APIs, adding and removing members and admins of their team as needed.



**Figure 4.17 User Attribute Policy**

In our case, the identities required are attributes of our customers that are used to make decisions. For example, our team lead may have the attribute *Team-Lead*, and another attribute of *Team-ID: Team-Sapphire*. And one of the developers on her team may only have the attribute of *Team-ID: Team-Sapphire*. Both engineers are members of team-sapphire, and this will allow both of them to interact with all of team-sapphire's tools and infrastructure. However, only the team-lead will be able to add and remove additional team members with the self-service API, because she also has the user type of *Team-Lead*. The self-service API will use this attribute to decide whether the requesting user can add and remove users. However, their Source Code repository may not need to check this attribute, it's only concerned with team membership to allow code changes.

The advantage of this approach is that we have completely removed the need for a central team to manage team requests. The Platform team can define which attributes are required to access specific tools within the ecosystem, and the team leads of customers can decide which team members have those attributes.

Further on, we'll talk more about how we can use these attributes in a structured and predictable way. But first, we need a protocol that allows us to use the attributes.

#### 4.2.4 Enabling Platform Single Sign On with OpenID Connect

Consider the fact that we need to enable platform Single Sign-On (SSO) with OpenID Connect (OIDC). For this to happen, it is essential to understand the components and processes involved in decoupling user identities from the underlying infrastructure while maintaining secure and flexible access control. OIDC, as an extension of OAuth2, provides a robust framework for this by allowing identity and authorization to be handled separately from the core platform services.

At the heart of OIDC is the **Authorization Server**, the intermediary between the platform and the Identity Provider (IdP). The Authorization Server's primary role is to authenticate users and issue tokens that encapsulate the user's identity and authorization information. This server handles the interaction with the Identity Provider, the system responsible for verifying user identities and managing the associated attributes.

Let's now examine the relevant key concepts, such as entity, identity, AuthN, and AuthZ.

**Entity:** An entity is any distinct user or system interacting with the platform. It could be an individual user, an application, or a service.

**Identity:** An identity is a collection of attributes uniquely identifying an entity within a particular context. For instance, a user's identity might include their username, email address, and other attributes specific to the platform.

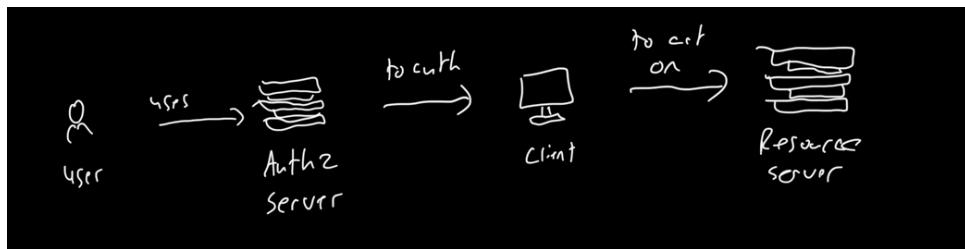
**Authentication (AuthN):** Authentication is the process of verifying an entity's identity. With OIDC, this is achieved by the Authorization Server interacting with the Identity Provider to confirm that the entity is who it claims to be. This step usually involves username/password verification, multi-factor authentication, or biometric checks.

**Authorization (AuthZ):** Authorization determines what an authenticated entity can do within the platform. It involves verifying the entity's permissions based on policies and attributes that are often included in the tokens issued by the Authorization Server.

When a user or an application (the entity) attempts to access the platform, the following process typically occurs:

1. **Authentication Request:** The entity initiates an authentication request, which is redirected to the Authorization Server.
2. **Interaction with the Identity Provider:** The Authorization Server communicates with the Identity Provider to authenticate the entity. If the authentication is successful, the Identity Provider returns an ID token and possibly an access token, which are sent back to the Authorization Server.
3. **Token Issuance:** The Authorization Server issues tokens that contain claims (attributes) about the entity. These claims represent the entity's identity and any additional attributes required for authorization decisions.
4. **Authorization Check:** When the entity tries to access a specific resource within the platform, the platform uses the information in the tokens (e.g., claims) to enforce authorization policies. The platform checks whether the entity has the necessary permissions to perform the requested action.

By leveraging OIDC, the platform can decouple user identities from its infrastructure, allowing for more flexible and scalable access control mechanisms. The attributes within the OIDC tokens can be used as claims, which are crucial to defining fine-grained authorization policies. These claims provide the necessary context for making authorization decisions based on the entity's identity and role within the platform.



**Figure 4.18 A recommended flow of attribute relationships**

#### 4.2.5 Claims-based User Authentication

Now that we've established we want to use OIDC at PETech, how do we authenticate our platform engineers and customer engineers? To do this, we can get into the finer details of how we want to use OIDC. If you recall, we discussed how OIDC provides additional user attributes or claims. In our case, these claims will be part of our JWT token payload. These claims can be anything. The standard claims are documented here: [https://openid.net/specs/openid-connect-core-1\\_0.html#StandardClaims](https://openid.net/specs/openid-connect-core-1_0.html#StandardClaims).

But we aren't just limited to the StandardClaims; you can include custom claims in the JWT Payload or via custom claims with [Aggregated and Distributed Claims](#).

To provide authorization for our platform engineers, we need to authorize them based on their use type, i.e., platform engineer. This claim can be provided to us via our authorization server. In our case, we will use GitHub as both the Identity server and Authorization server. Still, readers should note that in the enterprise context, the identity server would probably be Azure AD or another similar enterprise-wide identity system (one that is not specific to SCM).

When we authenticate our Platform Engineers with their Github Org, we will ask for a set of attributes (or claims), including their team memberships! The way we authorize them to perform administrative actions against the engineering platform is we perform validations against those team memberships, looking for the engineering platform contributor type or however you end up defining this team. Note that attributes may not exist in a central system.

The point is, we will need to Authorize users to perform actions after they have been authenticated as valid users, and then each service within the Platform will need to determine whether a given user is allowed to act. Deployed services will perform *policy-based* authorization based on claims or *attributes* the user presents when an operation is called. Infrastructure and platform services will also (when possible) perform policy-based authorization based on claims.

This is often called attribute-based access control. ABAC moves away from the traditional role-based authorization (does user x have y role) to a more granular and distributed authorization mechanism, one in which each system may have different access and usage policies and maintain them independently of each other.

This type of ABAC differs from the more traditional RBAC (role-based access control) in a number of ways. With traditional RBAC, a central repository will have mappings of users to predefined roles. Authorization decisions are based on an ACL allowing or disallowing certain roles. With ABAC, an access decision is made based on one or more attributes presented by the user along with what access they have to a given resource, and this information may be retrieved from multiple sources. In addition, by using policy-based authorization, each service will determine its own access rules instead of a centralized system that would decide what permissions a user has.

For example, with RBAC, a user with the role “buyer” may be allowed to add items to a cart. With our proposed ABAC solution, a user may be allowed to add an item to a cart if they present the “Buyer” user type claim, they are a member of the org they are trying to access a cart for, they are accessing the service from within the US, and they have been validated to make purchases for the type of equipment specified over a given dollar amount. User types can still function as groupings or “roles” for different users within an org, but they need not be the sole decision point.

Input from the business is needed for the following:

- Deciding what claims are available
- What the initial types are that users can be assigned to
- What Authorization policy should be applied to a given operation based on the claims available

By using ABAC with policy-based authorization:

- The amount of code that needs to be written to account for different use cases is greatly decreased
- Authorization decisions can be made as simple or as granular as desired, with easy extensibility over time
- Authorization can be changed quickly and without interruption. For example, if an auditor can see all cart items and a decision is made that this should not be allowed, we simply remove that claim, and after a policy update on the next access, it will be denied
- Each service can define its own AuthZ policy external to their code, meaning that changes to that policy do not require an application redeployment
- New Authorization policies and claims can be added without affecting code already deployed. If a team wants to take advantage of new attribute claims, they can do so when ready

Imagine an E-commerce system or trading system. A series of user types could be defined that an authorized member of an organization can grant to their users. Examples include

- Owner
- Admin
- Buyer
- etc....

These types would be defined from an end-user point-of-view regarding what they would like members of their organization to be able and unable to do. The definition of this distinction should be made clear to the decision maker when selecting, along with a statement that the user type may not be the only factor in authorizing an action within the system.

**Listing 4.5**

```
# input.json
{
    "method": "POST",
    "path": [
        "orders"
    ],
    "headers": {
        "Authorization": "<token>, use jwt.io to test"
    },
    "body": {
        "some": "order data object/json",
        "org_id": "e3179a3a-3197-43a3-a403-69376559c91a"
    }
}

package httpapi.authz
import future.keywords.in
default allow := false/

# Example is using a secret generated at jwt.io
#   As a developer you will need to properly decode the token using your signer's JWKS
#
# Data plugged into JWT.io:
# {
#   "user_id": "eebac5ae-4bc9-4bee-ab85-c1b73a49cd5a"
# }
#
# secret = "secret"
#
claims := payload {
    [valid, _, payload] := io.jwt.decode_verify(input.headers.Authorization, {
        "secret": "secret",
        "alg": "HS256"
    })
}
```

```

# We assume that the org_id is present in the request body, and we can verify it
# and fetch the user types at the same time.
user_by_org := http.send({
    "method": "get",
    "url": sprintf("<api_url>/account/v1/customers/%v/organizations/%v",
        [claims["user_id"], input.body.org_id])
})

# Parse the user's org object from our response.
# Should be of the form:
# {id, name, type}
allow {
    input.path == ["orders"]
    is_org(org)
    is_buyer(org)
}

allow {
    input.path == ["orders"]
    is_org(org)
}

is_org(org) {
    "id", input.body.org_id in org
}

is_buyer(org) {
    org.type == "BUYER"
}

```

Here, we have a straightforward OPA policy, that checks for 2 things:

1. Is the person requesting the purchase a member of a valid organization that can purchase
2. Does the person requesting the purchase have the “buyer” attribute, allowing them to make purchases on behalf of their organization?

So we can see how in the above example, there is an e-commerce purchasing API that uses attributes of the person requesting the purchase to make a decision. It’s important to understand that it is the API itself that makes the decision, not a centralized body.

So at PETech, each API will have a policy allowing, denying, or filtering access based on certain attributes. User-identifying claims (user type, userID, orgID, etc) should be embedded in and retrieved from a signed login token generated and signed by an AuthN service and presented during policy evaluation. In this way, the claims can be trusted, as opposed to claims included in a header, query-string or post body that can easily be mocked.

When we need to look up additional attributes or claims, it should be to an external system that can be queried using claims retrieved from the original login token.

#### **4.2.6 Exercise 4.3**

1. Write an OPA policy for an API server that allows a Team-Lead READ/WRITE/UPDATE/DELETE access to the /members endpoint and allows a Team-Contributor READ access to the /members endpoint
2. Extra Challenge: Modify the example provided for buyer user types to work for engineering platform user types

To tackle this exercise, you'll need to remember our learnings from 3.3 with OPA and incorporate what we learned about OIDC to write a successful attribute-based policy. Think about how you might apply this policy in the real world; what identity systems might you use to authenticate your users? What authorization servers work with your identity system? For an added challenge, test your policy with real tokens; you can generate them at <https://jwt.io>, which is an open, industry-standard way of generating secure web tokens

### **4.3 Chapter Summary**

- Governance, compliance, security, and audit must be integrated into both the platform and development processes.
- Applying governance policies to a self-service platform requires defining trust relationships and service boundaries.
- Expanding the software-defined platform with new engineering techniques is necessary for secure development and release processes.
- Reducing friction for development teams involves enabling autonomous access to resources and improving team onboarding processes.
- The creation of a Teams API allows for efficient management of team integrations and access to resources without removing control from system owners.
- Compliance at the Point of Change decouples compliance work from its verification, enabling self-service deployments while maintaining security standards.
- Kubernetes Admission Controllers enforce compliance at environment boundaries, allowing security teams to define policies and development teams to handle compliance.
- Open Policy Agent (OPA) and policy-as-code enable the security team to control policies within a verification strategy using the declarative Rego language.
- As platform usage increases, securing both the platform and deployed services becomes a critical challenge.

- Zero Trust Architecture applies default DENY policies with explicit ALLOW policies between systems, balancing security with productivity through self-service.
- Software Supply Chain Security is crucial to protecting against risks, with an emphasis on compliance at the point of change and policy-as-code.
- Distinguishing between Platform Team and Platform Customer Identity helps manage access and responsibilities efficiently.
- Decoupling customer identity from infrastructure is essential for managing access without overwhelming central teams.
- Claims-based User Authentication using OIDC provides flexible and scalable access control by separating identity and authorization.
- Transitioning to Attribute-Based Access Control (ABAC) allows for granular authorization decisions based on user attributes or claims.

# **5 Evolutionary Observability**

## **This chapter covers**

- Why observability is critical for both a platform and its users
- Providing observability as a service to platform users
- How Observability Platforms work and when they are needed
- Using Service Level Objectives (SLOs) to gain user confidence

Imagine that work on the platform starts at PE Tech with a backlog of stories, and things are going well. Prioritization of stories is leading to a steady stream of delivery, and with the observability-driven development practices that have been evangelized across the team, plenty of telemetry data can be used to diagnose and uncover issues. More importantly, the business has a great idea of the value platform efforts are returning right from the start. Across the engineering department, we are improving observability and seeing the benefits! More than quickly being able to diagnose and troubleshoot issues, leaders at the organization are starting to see that data can be correlated across applications and services to show how systems are performing across a whole portfolio, and they want to know more. They want to be able to quickly and easily define new queries that can be used to spot trends for areas that are succeeding well and those that are failing. The easiest way to do this quickly is by using spreadsheets. “Shadow IT” practices start popping up all over the organization as business leaders, finance, and even operations maintain their spreadsheets that are ingesting data into tables and charts. Teams are getting complaints that every time a data structure is modified, someone’s spreadsheet breaks, and they need time from the developers to understand why and how to fix it. The more spreadsheets that get built, the more these support complaints grow with every release because everyone has their version. Leaders have come to rely on these spreadsheets, so it’s always a high priority. Development starts to slow down because of the number of people who need to understand the changes. This is precisely the issue the platform and ODD practices were supposed to solve. Is it becoming a victim of its success?

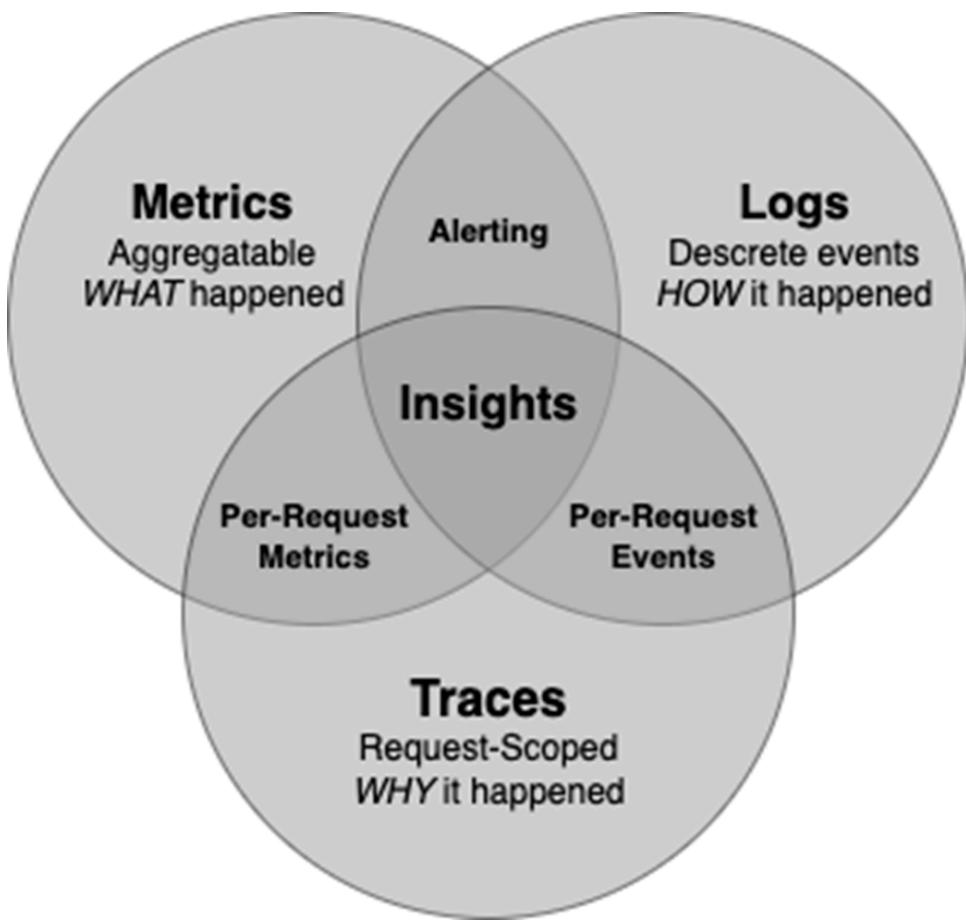
## 5.1 Why observability matters

As we have seen, good observability practices can increase the stability of a system by allowing the team supporting it to become aware of, diagnose, and fix issues quickly. It is also a powerful way to show value in any effort, proving to business stakeholders that any effort an engineering team takes can return value (customer-facing or not). This is because *data* can trump *opinions* on what is going well and what is not. True observability is more than just data and telemetry; it provides *insights*. To concretely define our usage of the term “observability”:

Observability is determining and explaining a software system's internal state and usefulness by gaining insight from its output data.

## 5.2 Observability is more than metrics and alerts

To gain insight from the observability data of any software system, we need more than just the metrics output by the infrastructure and applications running in the system. We should collect three distinct types of data: metrics, logs, and traces. Most systems will output some of these by default, and we should also aim to collect custom telemetry data as defined by our ODD practices. To know how we can get the most compelling insights, we should understand how we describe these telemetry types and how they can be used. In Figure 5.1, we describe the three core components of observability and how they intersect.



**Figure 5.1 Components of Observability** go beyond metrics to include logs and traces. These can be used to show what is happening in the system, why it happened, and how it got into its current state. Alerting and Logs are typically used to generate alerts, but correlating all three types can result in powerful insights into the system state.

### Metrics

Metrics are the point-in-time telemetry points typically aggregated over time and produced in high volume. When diagnosing a problem, the metrics will tell you *what* happened. This can be very useful for generating alerts if an unexpected event happens, such as an overloaded server or running out of capacity. Some examples include:

- CPU and Memory Usage
- HTTP Errors
- Disk Capacity

## Logs

Once you know about an event, you'll likely need to find out *how* the system got into that state. This is where logs become helpful. Logs are discrete events that happen as a process is executed and can be queried individually or as a set over time, but usually won't be aggregated. That means that combining the logs from multiple events in a summarized form has diminishing value as opposed to looking at it discretely. Data in logs can also be used for alerting when combined with metrics. Some examples include:

- A function was entered or completed.
- A request was sent to an external API, and a result was received
- The firewall blocked a network packet because of the source IP

## Traces

Knowing how a system got into a particular state is sometimes enough. Still, in modern systems, processes will usually cross multiple boundaries of applications and APIs as they are executed. When something happens there, we often need to know why an event occurred, and traces can help. Traces are events scoped to an individual request across multiple processes, and a correlation ID is used to join trace information across systems. An example trace could be:

Request received by the webserver -> Authentication token verified -> Request made to API ->  
Event sent to message bus -> etc....

## Insights

To get powerful insights from a system, we need to correlate all 3 of these telemetry types. For example, imagine that an alert is received indicating that a significant number of HTTP errors from a cart checkout have been returned to multiple users of your website for the last 20 minutes. You are responsible for building cart checkout feature and rush to find out what went wrong. You now query logs from the checkout system over the period the error codes were being returned and find out that you got an invalid response from another system that calculates the tax of an order. Still, it's unclear why that would happen. Now you can use the correlationID on one of those requests to trace the functions that were called and find out that six functions down a nested stack of calls, a process ran out of memory because the node it was running on didn't scale when it was supposed to. Without making these correlations quickly, you may have been debugging for hours only to discover it was an infrastructure issue!

### 5.2.1 Use cases for observability beyond applications

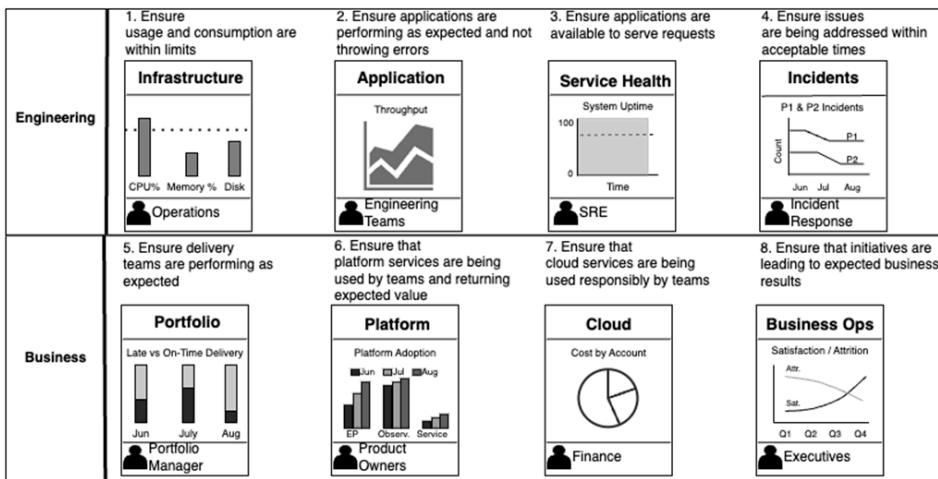
As an engineer, it's natural to consider observability regarding the telemetry data and insights needed to monitor and diagnose the infrastructure and applications your team supports. To ensure that your observability practices are most effective, you should also recognize that this data and the insights it can produce are helpful to many stakeholders across the business.

One of the common anti-patterns we have seen in the industry is the singular focus on application observability. In our opinion, this is a significant reason why the eventual experience for the end-users of your services is suboptimal. Instead, we strongly recommend looking at observability through the eight distinct but related lenses to ensure a better outcome.

Let us now look at how PETech understood the value of an expanded focus. Their initial focus, like many of the other organizations we know, was singularly on application performance. This helped their engineering teams troubleshoot the application issues but pretty soon they realized this wasn't enough. The customer complaints continued and the executive leadership was baffled to see that the much vaunted observability approach did not yield the results they were looking for - customer satisfaction about their products. After a deeper dive, the team recognized a critical gap in their approach. They found the following problems

1. Third-party tools used across the development and delivery ecosystem weren't observed adequately.
2. There were overnight processing of some operational tasks that had to run as scheduled tasks to do the processing. These were run as cloud services which, when failed provided significant impact on the customer's experiences.
3. There was the silly problem of some of the disks were filling up on two of the production servers which the monitoring always caught after the fact, creating an annoying customer experience.
4. The cybersecurity team had a completely different process that was not integrated into the central observability framework where the release management and senior leadership would hear about the security breaches. This was starting to have a reputation and credibility impact for the senior leadership.
5. PETech found that conversion rates from first time users to repeat users were dropping significantly. However, they found this during their monday morning review calls, by which time they had lost half the battle.
6. By standard practice, CFO received cloud usage reports on a monthly basis showing significant overruns. Instead, she wished there was a way to have the developers and SREs adjust the cloud usage scaffolding on a real-time basis without having to have her team get involved.
7. As PETech was expanding to Europe region, obtaining specific feedback on how GDPR privacy regulation were reported on a daily basis and ensuring compliance against that was becoming a critical requirement.

In Figure 5.2, we introduce the eight axes (seven listed above, in addition to application observability) that addressed each of the problems PETech encountered and we are sure you will too. Suppose you confine your observability efforts to just the applications and the infrastructure on which the applications run. In that case, you will miss the big-picture view of your eventual goal - an ideal customer experience while running your systems in the most cost-optimal manner.



**Figure 5.2 Observability data can be described across multiple engineering and business operations facets. Users and use cases are more than developers and operations personnel responsible for running the system. Stakeholders, security, and governance also have questions the data can answer.**

Observability can be described across eight facets, and recognizing them can inform the types of telemetry that should be produced. This data can be queried and aggregated across these facets for insights to drive engineering and business strategy decisions. Here are some examples of how engineering and business stakeholders can use each aspect of observability; you can likely think of many more.

**Infrastructure:** Telemetry on the hardware (physical or virtual) that runs the systems can generate insights such as usage, consumption, and failures. It is typically used by operations and DevOps personnel but is also helpful to system architects to ensure right-sizing.

**Application:** Telemetry on running applications. Engineering teams can use this to ensure the health of software systems and diagnose issues. Still, it is also valuable for product managers to determine whether new features are being used as expected or if a feature should be prioritized to develop a better user experience.

**Service Health** is data that indicates whether a service (which may consist of many applications and infrastructure resources) is running well. SRE teams typically use it to optimize runtimes and ensure stability. Product owners can also use it to prioritize stability issues on a backlog over new feature development.

**Incidents:** Data from ticketing systems or incident response workflows. Incident response usually uses incidents to measure team effectiveness. Engineering leaders can also use incidents to evaluate the success of a platform initiative designed to decrease incident response times.

**Portfolio:** Data to indicate effectiveness on portfolio delivery across an engineering function. This could include telemetry around deployment frequency, on-time feature delivery, or user story cycle times aggregated across teams. Team leads use this data to monitor effectiveness, and managers use it to identify bottlenecks and cross-team dependencies to inform team structure decisions.

Platform: Data will indicate the platform's usage and health. This could include team adoption rates or how often platform services are used. Product managers use it to determine backlog priorities, and the business can also use it to value the ROI of a platform initiative.

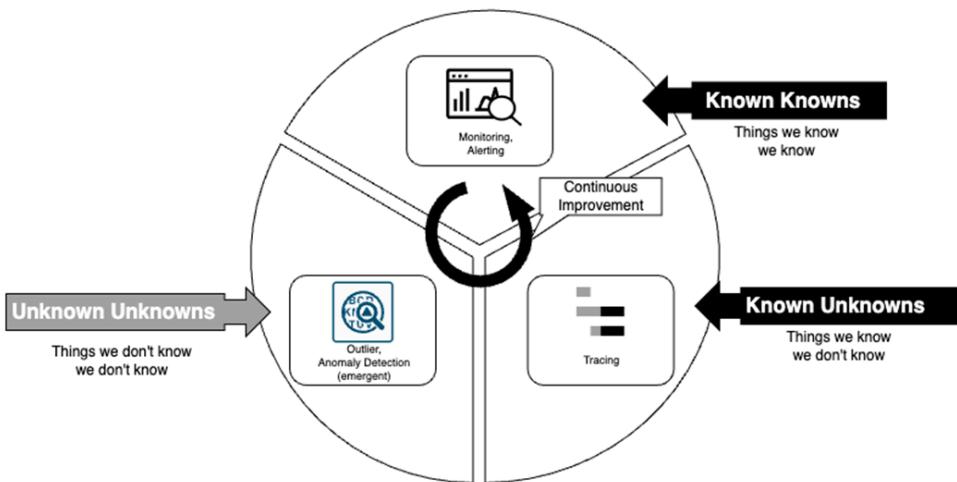
Cloud: Data on cloud usage and cost. Architects use it to meet runtime cost targets, and finance departments can also use it to calculate ROI on cloud costs across teams and environments.

Business Operations: Data on systems across business capabilities that indicate business value. Used by product owners to ensure that newly released features return expected ROI, and also used by leaders to monitor the health of the business.

### 5.2.2 What Does Good Look Like?

At PETech, when they started putting together the observability across eight axes, one of the first questions was how to interpret the data and, more importantly, prioritize what data should be observed and why. This section will discuss a simple technique to look at three potential data states, which will help you answer both of these questions.

Data by itself has no meaning. If you were told that CPU usage on a critical server runs at 75%, is that good or bad? It is impossible to know the answer to that question without *context*, meaning we need to understand what that server is doing and what we expect the meaningful load to be. If we have a Kubernetes-based platform, we may want to take advantage of scheduling to ensure each node runs at high capacity to get the most out of the infrastructure costs being incurred. In this case, it would be okay if the CPU was below 90%. In another case, we may have a workload that has bursts of activity regularly, and we need to ensure that, on average, the CPU is below 60%. In other words, we need to know more than whether a system is healthy and running; we need to understand what it means to *run well* and *as expected*. In Figure 5.3 below, we demonstrate how to look for different states to ensure the system has all the necessary signals to operate efficiently and correctly.



**Figure 5.3 Knowing what to look for when querying observability data is an evolving process. Monitoring is based on what you know a good state should look like so you can be alerted when something goes wrong. Emitting data that enables tracing allows you to investigate things you know could go wrong but aren't sure how. Accurate insights come from debugging unanticipated issues and, over time, growing the knowledge base of known knowns.**

When defining what good looks like for a system, there are three states that we need to be aware of to ensure we have the correct observability data to diagnose them.

#### Known-Knowns

These are the things we know can go wrong. In the above case, we might know that any CPU activity above 90% indicates that our Kubernetes cluster isn't scaling as expected. Based on load testing, we may also know that an API designed to automate the creation of new team projects in source control can only handle ten requests simultaneously. For the business, we may have defined that at least six teams must adopt a new platform service in the first month of release to consider it successful. In all these cases, we can and should define telemetry, queries, and alerts to indicate when a known good state is violated. We may also be able to design self-healing routines to remedy the system state quickly.

#### Known-Unknowns

We know these system states could cause a problem, but we aren't sure how. For example, we may know that a new node could fail to provision if the Kubernetes cluster needs to scale, but we don't know what scenarios could cause that. We may suspect that a call to an identity provider could return invalid data, but we do not know what that might look like or why invalid data was generated. In these cases, we can and should publish telemetry to observe these processes, and we may be able to create an alert to indicate something went wrong. Still, we may not be able to generate processes, queries, or dashboards that would allow us to remedy the issue quickly. However, we will know that something went wrong quickly and (hopefully) have the data needed to diagnose.

#### Unknown-Unknowns

These are the blind spots in the system where there is no recognition that a problem could occur. It may be a simply overlooked situation (although ODD practices should help) or an utterly unanticipated scenario. It could be that a user is reporting an error when deploying a workload that no one has seen before, or a service is crashing for no apparent reason. To handle these situations, we may be able to publish observability data that gives us reasonable confidence that we can trace an error. Still, in some circumstances, this may result in a system bug that needs more time to be tracked down.

In all of these cases, we should evolve practices such that the list of known use cases continues to grow. If there is an unknown-unknown situation, we should implement observability as part of the bugfix to make it a known-known situation in the future or recognize that a known-unknown state could occur and ensure the problem can be diagnosed and fixed much more quickly next time.

### **5.2.3 Viewing observability through a single pane of glass**

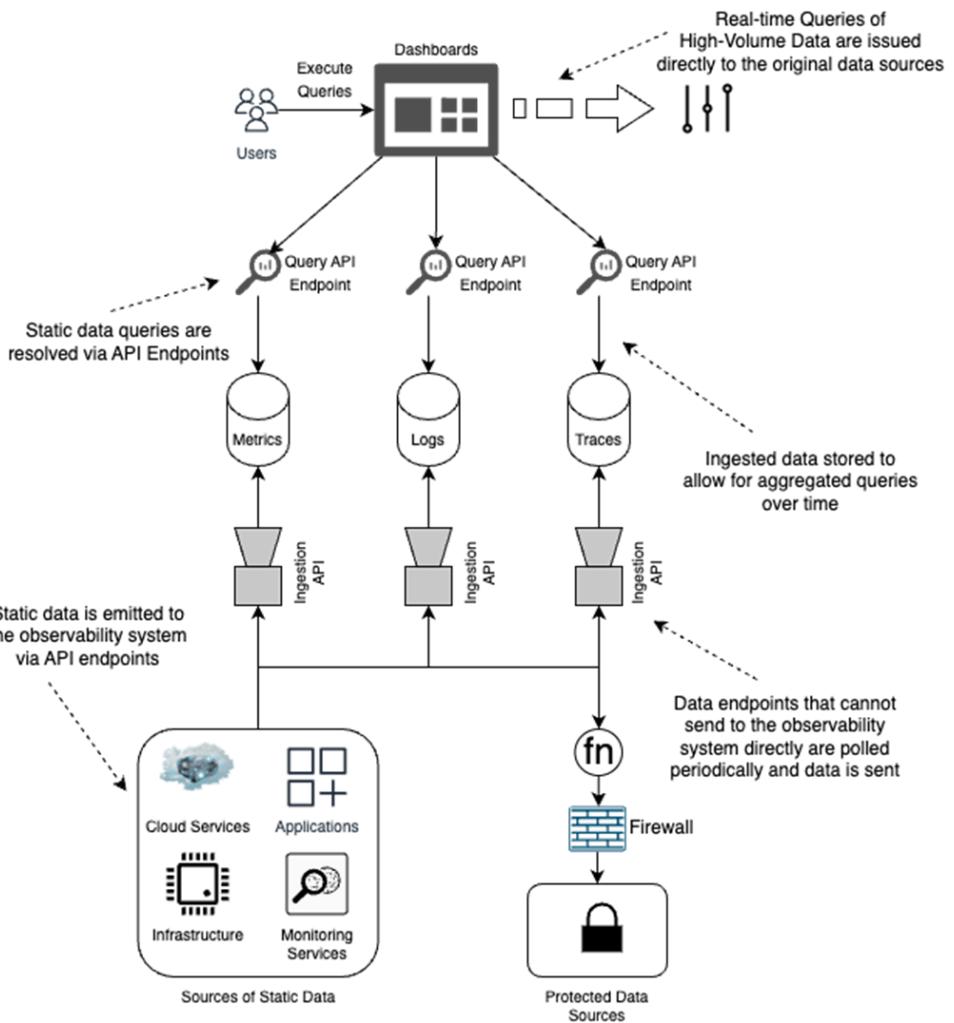
As we have seen, the true power of observability data is unlocked when different data types can be correlated, both within and across systems. Let's take PETech as an example again. PETech uses disparate tools for all aspects of observability (Hint: the eight axes), as described in Table 5.1.

**Table 5.1 Tools used by PETech across the eight different axes of observability.**

Axis	Tool Used
Infrastructure	Prometheus (metrics collection) + Grafana (Dashboarding)
Application	DataDog
Service Health	Pingdom
Incident Response	PagerDuty
Portfolio	JIRA
Platform	DataDog
Cloud Costs	Kubecost
Business	Tableau

Now, think about the impact on productivity if different stakeholder personas, such as developers, SREs, business owners, and others, have to look at multiple dashboards to understand what is going on in their overall ecosystem. This problem will be solved by having a single pane of glass (SPOG). By providing an easy experience to join these available data sources in a SPOG, observability practices will give a more significant benefit faster, making it much more likely your teams will be excited to adopt observability practices. This system will allow querying, correlation, and visualization of any observability data available from a central system, regardless of user persona. This does not mean all your observability data must be stored in a central, monolithic datastore. We can provide the data using multiple methods based on the data update type, volume, and frequency.

In Figure 5.4 below, we show a high level solution architecture of how to build a SPOG. It shows the solution components, data sources and how to bring the data together in a meaningful manner.



**Figure 5.4** By allowing observability data to be queried across data sources in a single place, generating insights across the entire business is much easier. Users can create dashboards and alerts based on these insights, allowing them to see the business's health and individual systems.

At the base of the architecture are the Sources of Static Data, which include various entities such as cloud services, applications, infrastructure components, and monitoring services. These sources generate static data like system metrics, application logs, and traces that provide insight into the system's operational state. This data is emitted to the observability system through API endpoints. Static data is crucial for historical analysis, capacity planning, and long-term system health monitoring. In scenarios where specific data endpoints cannot directly send their data to the observability system due to security restrictions or network configurations, they are periodically polled, allowing data to be collected securely. This polling mechanism is illustrated with a Firewall and Protected Data Sources, indicating that the observability system is designed to handle secure and protected environments.

The data emitted from these sources is processed through Ingestion APIs. These APIs serve as the gateways for data entering the observability system. There is a dedicated ingestion pipeline for each type of data—metrics, logs, and traces—to ensure the data is correctly processed and stored. The Ingestion APIs validate, format, and store the data in their storage backends. This structured approach to data ingestion ensures that metrics are aggregated and indexed appropriately, logs are parsed and stored efficiently, and traces are correlated for distributed tracing.

Each category of data—metrics, logs, and traces—has a corresponding storage component in the architecture. Metrics are stored in a time-series database optimized for efficiently retrieving numerical data points over time. Metrics provide quantitative data about the system's performance, such as CPU usage, memory consumption, and response times, enabling detailed performance analysis and anomaly detection. Logs are stored in a log management system that allows for aggregating and querying log data. Logs capture detailed information about system events and application behavior, facilitating root cause analysis during incidents. Traces are stored in a distributed tracing system that reconstructs transaction flows across multiple services, essential for understanding end-to-end latency and identifying performance bottlenecks in microservices architectures.

Users interact with the observability system through Dashboards, which provide a unified interface for querying and visualizing the collected data. The dashboard component is central to the architecture, allowing users to execute queries against the stored data. Query API Endpoints are provided for each category for static data—metrics, logs, and traces. These APIs enable storing data retrieval, allowing users to perform historical analysis, identify trends, and troubleshoot issues. The dashboard can be used to build visual representations of all the necessary data to create business impact.

In addition to querying static data, the observability system also supports Real-time Queries of high-volume data. This capability is crucial for responding to dynamic changes in the system's state and providing immediate feedback on system behavior. Real-time queries are issued directly to the original data sources, bypassing the stored data to ensure that the most current information is available for analysis. This real-time querying is particularly useful for monitoring system health, detecting incidents as they occur, and enabling rapid response to issues.

An architecture similar to the above must provide a robust and flexible observability system that supports static and real-time data analysis. By using dedicated ingestion pipelines, secure data collection mechanisms, and flexible querying interfaces, the system offers a solution for monitoring, analyzing, and optimizing your software systems. The design ensures that data is stored efficiently, queried flexibly, and visualized effectively, enabling users to gain actionable insights into their systems' operations.

## **EXERCISE 5.1: SET UP AN OBSERVABILITY SYSTEM FOR INFRASTRUCTURE MONITORING**

In this exercise, you will start by selecting an observability stack in your organization. You will also need a system for collecting metrics, logs, and traces.

- First, you should identify your observability goals - what you need to monitor and determine the scope of what you want to measure

- Then, consider using open-source components like *the Grafana LGTM stack, Prometheus (both for metrics), and Fluentd (for log aggregation)*, as they all have open-source versions.
- Since you are not monitoring applications in this exercise, you may not yet have to instrument your applications.
- Once you decide on the stack, install and configure them and collect the data for your infrastructure monitoring.

The expected deliverable for this exercise is a working dashboard that provides real-time metrics and logs. Traces are optional, but they are typically needed to identify how requests flow through the system, a far more detailed use case for most of the more straightforward problems that can be solved with real-time metrics and logs. However, if you wish to configure your tracing, we recommend trying out *Jaeger*, which allows distributed tracing.

## 5.3 Observability as a platform service

When it comes to observability platforms, the end-user experience is paramount. Industry-standard observability solutions are expected to provide a seamless and intuitive interface that enables users—from engineers to executives—to gain deep insights into the system's health and performance. The best observability platforms offer robust integration capabilities, comprehensive data aggregation, and a user-friendly experience that simplifies the complexity of monitoring and troubleshooting distributed systems.

A critical requirement for any observability platform is the ability to integrate effortlessly with the platform's existing authorization architecture. In modern enterprises, this typically involves integrating with an external Identity Provider (IdP) for single sign-on (SSO) and role-based access control (RBAC). End users expect a frictionless experience when accessing the observability platform, leveraging their existing credentials and permissions. An observability solution that supports seamless integration with external IdPs allows organizations to maintain consistent security policies and provides users with a unified access experience across all tools and services. This integration ensures that only those who were preauthorized can access all the data, enhancing security and compliance.

### 5.3.1 The end-user experience

Users will expect an observability platform that can be categorized by different lenses on what is being provided. The experience your users will look for will span many of these lenses simultaneously, but other situations will have different requirements.

An essential aspect of the user experience in observability platforms is supporting platform-wide aggregation of metrics, logs, traces, and other telemetry data. Providing isolated observability per-team can be tempting in complex, distributed service architectures. However, this approach carries significant risks. Isolated observability can lead to fragmented system views, where each team only has visibility into its services. This fragmentation makes understanding the overall system behavior challenging and increases the mean time to recover (MTTR) from issues. When incidents occur in a distributed environment, the root cause often involves multiple services interacting unexpectedly. Without a holistic view of the system, pinpointing the exact cause of an issue will be highly time-consuming.

Aggregating and correlating data across the platform reduces MTTR and improves the end-user experience. A well-designed observability platform aggregates data from all services into a centralized dashboard, allowing users to trace issues across service boundaries easily. This aggregated view helps teams quickly identify patterns, understand dependencies, and trace the flow of requests through the system. When an issue arises, users can seamlessly navigate from high-level metrics, like increased error rates, to specific traces and logs that reveal the root cause. This end-to-end visibility speeds up the troubleshooting process and empowers teams to proactively identify and resolve potential issues before they impact users.

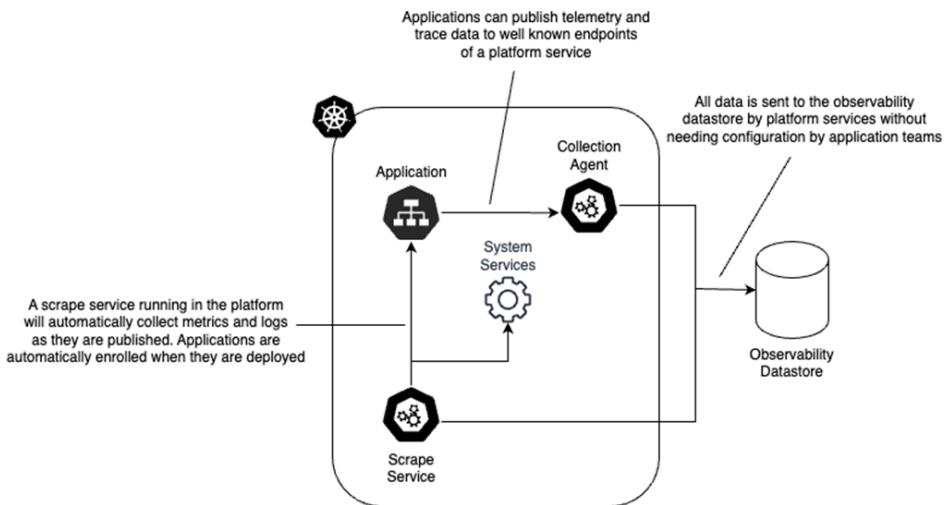
Your user's experience is also influenced by how well the observability platform can scale to accommodate the organization's growing needs. As the platform evolves and more services are added, the observability solution must continue to provide a unified and responsive experience. As the volume of metrics, logs, and traces increases, the platform should still be able to quickly process and present this data to users in an actionable format. Slow or fragmented observability tools can frustrate users and hinder their ability to maintain system health, especially during critical incidents.

Moreover, the platform should offer user-friendly features such as customizable dashboards, automated alerts, and intuitive querying interfaces. Users need to be able to create and share dashboards that present relevant data in a way that is meaningful to them, whether it's high-level system health indicators for executives or detailed service-level metrics for engineers. Automated alerts should be easy to configure and provide timely notifications when anomalies are detected, enabling teams to respond quickly to potential issues.

The end-user experience of an observability platform hinges on its ability to integrate seamlessly with the platform's authorization architecture, support platform-wide data aggregation, and provide an intuitive and responsive interface. By offering a holistic system view, the platform can reduce MTTR and enable users to effectively maintain the health and performance of complex distributed architectures. For organizations like PETech, investing in an observability platform that prioritizes these user experience aspects is crucial for ensuring that teams can efficiently monitor, troubleshoot, and optimize their systems, ultimately delivering a better product to their customers.

### 5.3.2 Automatic collection of customer data

Collecting telemetry data—metrics, logs, and traces—from user applications in a platform environment is a well-understood area with standard architectural patterns. When PETech tried to setup the automated collection, they first started with a notional diagram like what we have shown in Figure 5.5 below, where the three forms of elementary data - Metrics, Logs and Traces can be ingested. In this section that applications generate telemetry data that must be monitored and analyzed to ensure smooth operation and performance.



**Figure 5.5 There are three primary forms of telemetry that user applications will produce and need to be ingested: metrics, logs, and traces. By collecting these automatically via well-known endpoints, your users will have a much easier onboarding experience when getting started and will be unaffected if the backend technologies change over time.**

The platform employs a Collection Agent and a Scrape Service to make this process seamless. Applications are configured to publish their telemetry and trace data to well-known endpoints the platform provides. This means that as soon as an application is deployed, it can automatically start sending its telemetry data to these endpoints without requiring additional configuration by the application teams. This immediate integration dramatically simplifies the onboarding process for new applications, reducing the friction that developers often face when setting up observability.

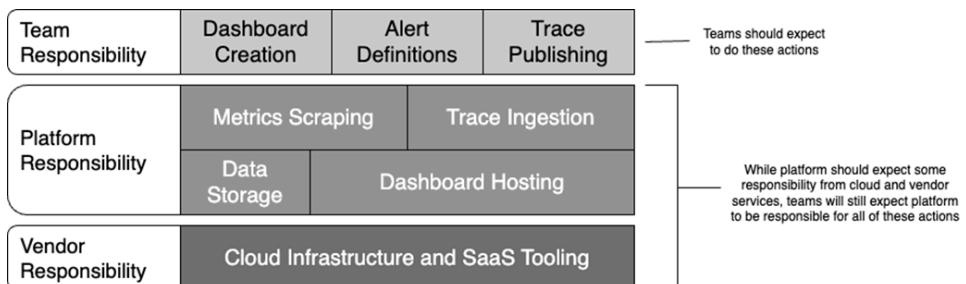
The Collection Agent is a mediator that gathers all the telemetry data emitted by the applications and system services. Meanwhile, the Scrape Service continuously runs within the platform to automatically collect metrics and logs. It enrolls applications into the monitoring system as deployed, ensuring that all new applications are automatically included in the observability pipeline. This automatic enrollment process means that developers don't need to manually configure monitoring for each new application, saving time and reducing the risk of human error.

All collected data is then sent to a centralized Observability Datastore. This centralization allows the platform to aggregate, store, and analyze telemetry data in a unified way. Because the platform handles the collection and routing of telemetry data, it abstracts away the complexities for application teams. They don't need to worry about the underlying mechanisms of how their data is gathered and stored. Furthermore, if the backend technologies evolve or change over time, this setup ensures that applications remain unaffected as they interface with the same well-known endpoints.

At PETech, the technical product manager asked whether the above setup is exclusive to building an observability platform from scratch. This approach can also be applied when using existing observability tools or platforms, whether they are open-source or commercial products. The key idea here is the automation and standardization of telemetry data collection, which can be achieved through configurable and extensible mechanisms in many modern observability solutions.

### 5.3.3 Who needs to respond when things need attention?

Figure 5.6 illustrates a shared responsibility model that PETech should adopt to implement and manage an observability platform effectively. This model is crucial for setting clear boundaries on who is responsible for different aspects of the observability infrastructure. By defining these roles, for example, PETech can reduce friction between teams, the platform team, and external vendors, ensuring a more seamless and efficient observability experience.



**Figure 5.6 A shared responsibility model is critical to reducing friction when teams adopt any platform service. The platform cannot be expected to abstract everything for users, but it is equally important to define who should respond when there is an issue with managing user expectations.**

Let us look at the layers in the shared responsibility model above.

At the top layer, Team Responsibility includes tasks like Dashboard Creation, Alert Definitions, and Trace Publishing. This means that individual application or service teams within PETech are responsible for setting up the specific dashboards they need to monitor their services. They should also define alerts relevant to their operational context—such as setting thresholds for response times or error rates—and publish traces to track the flow of requests through their services. While the platform provides the tools and frameworks for observability, it is up to each team to utilize these tools effectively to gain insights into their applications. By making this a team responsibility, PETech ensures that the people closest to the services are configuring the monitoring and alerting, resulting in more meaningful and actionable insights.

The Platform Responsibility layer is where PETech's platform engineering team comes into play. This team is responsible for managing the core aspects of the observability infrastructure, including Metrics Scraping, Trace Ingestion, Data Storage, and Dashboard Hosting. In practice, the platform team sets up and maintains the mechanisms that collect metrics and traces from various applications and services. They ensure this data is ingested into a centralized data store, which can be aggregated, queried, and analyzed. Additionally, the platform team hosts and manages the observability tools, such as Grafana or Prometheus, making them accessible and easy for all other teams. By handling these responsibilities, the platform team abstracts much of the complexity involved in gathering and storing telemetry data, allowing service teams to focus on interpreting and acting on the data rather than worrying about the mechanics of how it is collected and stored.

The bottom layer is Vendor Responsibility, which includes Cloud Infrastructure and SaaS Tooling. This reflects the reliance on external vendors for foundational services such as cloud infrastructure (e.g., AWS, Azure, Google Cloud) and SaaS observability tools (e.g., Datadog, New Relic). For PETech, vendors are responsible for ensuring the reliability, scalability, and availability of the underlying infrastructure and tools on which the observability platform depends. The vendors handle aspects like data center operations, network reliability, and the health of the SaaS services. By leveraging vendors for these foundational services, PETech can offload the infrastructure's operational burden, allowing its platform team to focus on higher-level observability tasks.

## **EXERCISE 5.2: CORRELATING DATA WITH A DEMO APPLICATION**

In this exercise, you will build on exercise 5.1.

- Identify a demo application.
- Start your observability stack
- Send the data from your demo application.
- Check and make sure that the data is showing up on the observability dashboard.

## **5.4 Observability platforms**

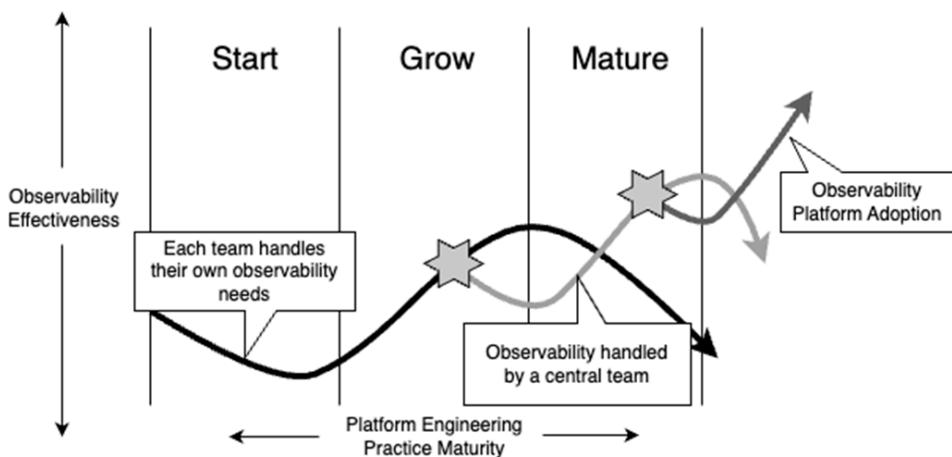
As discussed in section 5.1.4, PETech has used several tools to manage its observability problem. However, for them, just like it might be for you, it was an arduous journey to get where they got to with standardization. About eighteen months back, they had different scrum teams using various tools in an utterly non-standardized manner. Most of the tools were used from a monitoring perspective. The primary change they had to bring about was a unified platform centric approach to observability, based on the disparate monitoring tools. We would encourage you to consider the same approach in your teams too. In platform engineering, an observability platform is a set of tools and systems designed to help engineers understand what's happening inside their software and infrastructure. Think of it as a monitoring system on steroids. While traditional monitoring tells you if something is wrong (like a smoke alarm going off), an observability platform helps you figure out what is happening, why, and how to fix it before your customer sees the problem. If you think about it, you see monitoring together with or after the customers know the situation, where you have lost half the battle. Observability lets you see all of that before them, and you can help fix these problems ahead of time!

Unexpected problems can occur in a complex digital environment where multiple services and applications work together. An observability platform collects data like metrics (numbers showing how the system is performing), logs (detailed records of what's happening inside the system), and traces (maps showing how requests travel through different services). This data is then presented in a way engineers can easily understand, using dashboards and alerts.

This platform is crucial for engineering because it provides the necessary visibility to ensure that all the underlying systems and services run smoothly. If a problem occurs—like a slow application response or a system failure—the observability platform helps engineers quickly find out where the issue is happening and what caused it. This means they can fix problems faster, improve the system's performance, and provide a better experience for the users. An observability platform is like having a detailed map and diagnostic tool for your entire software environment, ensuring everything runs as expected.

#### 5.4.1 Evolving to the need for an observability platform

Let us now look at the journey of observability in an organization like PETech as it evolves from the early stages of platform engineering to a mature state in Figure 5.7. It shows how the approach to observability changes over time and why investing in a centralized observability platform becomes increasingly important as the organization grows.



**Figure 5.7 When the adoption of platform engineering services starts, there may not be a need for a fully functional observability platform distinct from the engineering platform. Over time, however, the needs of a growing number of stakeholders, teams, and systems will make investing in this a more feasible option.**

In the Start phase, observability is usually handled by individual teams. Each team at PETech might set up its tools and processes to monitor their specific applications or services. This approach works when the organization is small or teams operate relatively independently. At this stage, PETech can quickly get some level of observability without much overhead. However, because each team uses different tools and methods, the overall effectiveness of observability across the entire organization may be limited. There can be gaps in visibility, and it's challenging to get a unified view of the system, especially in a distributed architecture. This fragmented approach might lead to difficulties in diagnosing issues across multiple services.

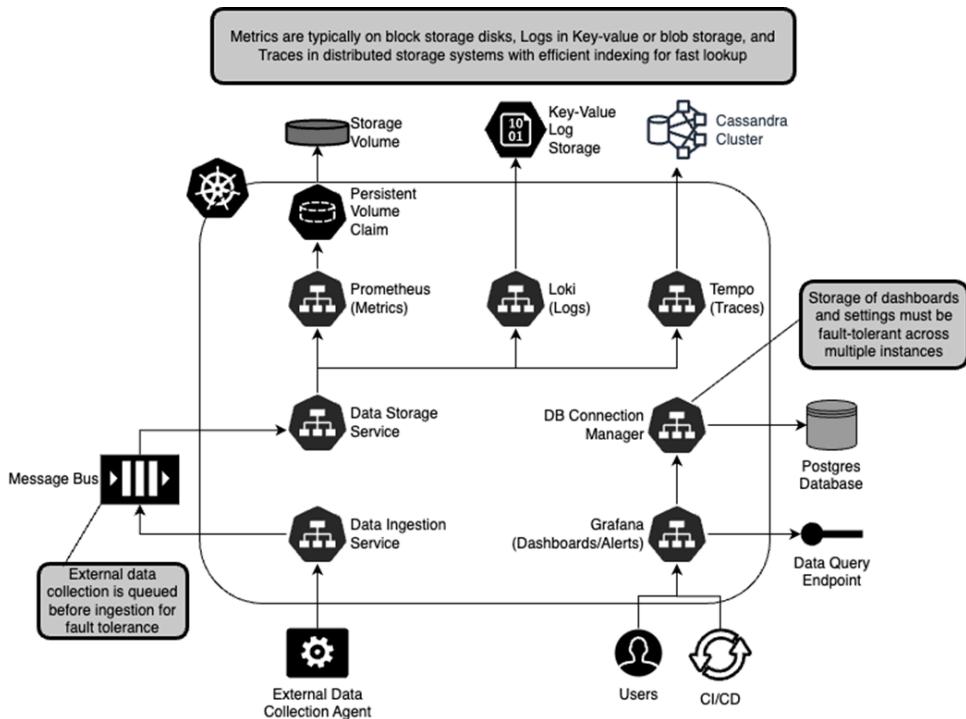
As PETech moves into the Growth phase, the complexity of its platform engineering practices increases. More services are added, and the interactions between them become more intricate. At this point, the limitations of each team handling observability separately become apparent. The lack of a unified observability approach can lead to a longer mean time to recover (MTTR) during incidents because it's hard to trace problems that involve multiple services. Recognizing this challenge, PETech begins to centralize observability efforts. A central team takes responsibility for setting up and maintaining a unified observability platform. This team provides standard tools, practices, and data storage for metrics, logs, and traces, making it easier for all teams to monitor their services consistently. This centralization improves the overall effectiveness of observability, as PETech can now get a holistic view of the entire system.

Finally, PETech has fully adopted a centralized observability platform in the *Mature* phase. The observability platform becomes a core part of the organization's infrastructure, providing a single source of truth for monitoring, troubleshooting, and optimizing all services. The platform is integrated with the organization's workflows, making it easier for teams to understand the health and performance of the system. With this mature approach, PETech can quickly detect, diagnose, and resolve issues, significantly reducing MTTR and improving the reliability of its services. The central observability platform is now sophisticated enough to handle the needs of many stakeholders, teams, and systems.

Investing in a centralized observability platform becomes more feasible and beneficial over time, allowing PETech to handle complexity more efficiently and provide a better, more reliable user experience.

### **5.4.2 Architecture of an observability platform**

The following diagram in Figure 5.8 presents a comprehensive architecture of an observability platform in distributed systems. This architecture enables the collection, processing, storage, and visualization of metrics, logs, and traces, providing a robust solution for tracking the health and performance of applications and infrastructure.



**Figure 5.8 Most observability platforms, whether OSS or commercial, will use a similar high-level architecture. Specific technology components will be different depending on scale, but understanding this footprint will be important when considering build vs buy options.**

At the forefront of the data flow is the External Data Collection Agent, the primary entry point for gathering metrics, logs, and traces from various external sources. These sources could include applications, infrastructure components, or network devices. The data collected by this agent is crucial for building a complete picture of the system's operational state. Once gathered, this data is forwarded to the Message Bus for further processing.

The Message Bus acts as a buffering layer in this architecture ensuring fault tolerance. It queues incoming data, temporarily storing it until the system is ready to process it. This buffering capability is essential in preventing data loss, especially during peak times or when the downstream systems are temporarily unavailable or overwhelmed. The message bus smooths out the data flow by decoupling the data collection from the processing components, ensuring the ingestion process is resilient and scalable.

Once data is buffered in the message bus, the Data Ingestion Service takes over. Its primary responsibility is to consume the queued data, format it appropriately, and prepare it for storage. This service is pivotal in structuring the incoming data into a format that aligns with the requirements of the storage systems and querying tools that will later analyze this data. By separating the ingestion process from data collection, the architecture allows for greater flexibility and scalability in processing and storing data.

The next critical component is the Data Storage Service, the conduit for routing the processed data to the appropriate storage backend. Depending on the data type—metrics, logs, or traces—the storage service directs it to the corresponding storage solution. For example, metrics data is routed to Prometheus, a time-series database for storing high-resolution metrics data. Prometheus is often paired with block storage managed via Persistent Volume Claims to ensure data persistence even across container restarts or migrations.

Logs are stored using Loki, a log aggregation system designed for high throughput and efficient log storage. Logs are stored in a Key-Value Log Storage system, which can efficiently handle large volumes of log data. This system often utilizes distributed storage backends like a Cassandra Cluster for scalability and reliability. Logs are indispensable for debugging and tracing issues in a distributed system, providing detailed records of system events and application behaviors.

For tracing data, the architecture utilizes Tempo, a distributed tracing system that stores traces in a distributed storage system like Cassandra. Traces are crucial for understanding the flow of requests through the various services in a distributed architecture, helping to pinpoint bottlenecks, latency issues, and system failures. Using a storage solution that offers efficient indexing and fast lookup capabilities, Tempo allows for rapid querying and analysis of trace data, which is essential for real-time monitoring and troubleshooting.

The architecture also includes a Storage Volume managed through Persistent Volume Claims to provide reliable, persistent storage for Prometheus and other components. This ensures that even when the system restarts or the container is rescheduled in a Kubernetes environment, the stored data remains intact and accessible. This persistence is vital for maintaining a continuous and reliable view of system performance over time.

Grafana is integrated into the system as the primary tool for visualization and alerting. To provide a unified system monitoring interface, it connects to various data sources, such as Prometheus for metrics, Loki for logs, and Tempo for traces. Users and CI/CD pipelines interact with Grafana to create dashboards that visualize the collected data, set up alerts for specific conditions, and query the underlying storage backends for deeper analysis. Grafana's flexibility and extensibility make it a critical operational monitoring and decision-making component.

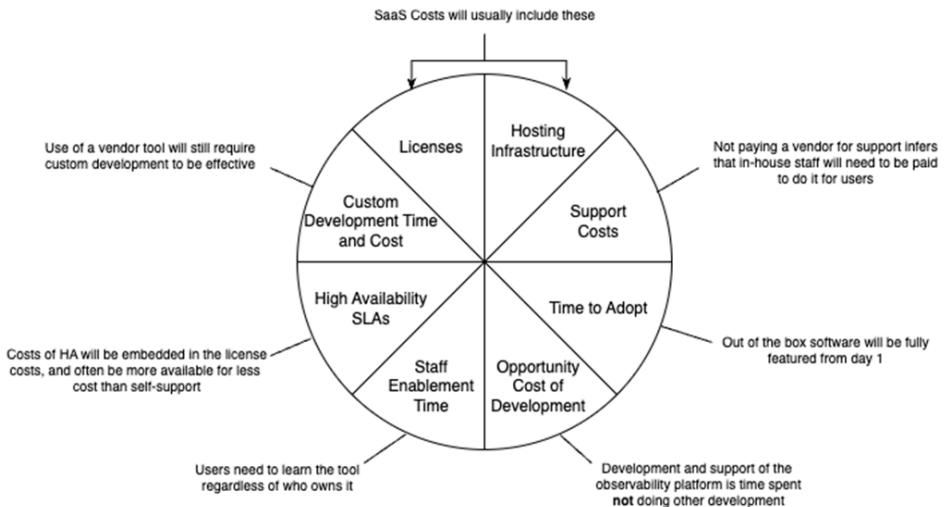
To support Grafana's operation, the architecture includes a DB Connection Manager that interfaces with a Postgres Database. This database stores the configurations, dashboard settings, and alert rules Grafana uses. By using a relational database like Postgres, the architecture ensures that these configurations are stored fault-tolerant, capable of surviving failures and scaling across multiple instances if necessary.

Finally, the architecture provides a Data Query Endpoint. This endpoint is the access point for users and CI/CD systems to query stored data. Whether for real-time analysis, historical data review, or automated responses to certain conditions, this endpoint facilitates flexible and efficient access to the telemetry data.

This telemetry pipeline architecture is a well-orchestrated system designed to provide comprehensive monitoring and observability for distributed systems. It efficiently collects, processes, stores, and visualizes data, offering a robust framework for understanding and managing system performance, health, and reliability. Each component—data collection to storage, visualization, and querying—is tailored to handle specific data types and operational requirements, ensuring a scalable, fault-tolerant solution.

### 5.4.3 Should you build or buy?

When deciding whether an organization like PETech should build its observability platform or buy an existing one, all the factors that go into the total cost and effort must be considered. Figure 5.9 breaks down these considerations into several key components that give us a clearer picture of what's involved in each approach.



**Figure 5.9 Finding the TCO of buying or building an observability platform, or any commercial software, is much more than just license costs and infrastructure hosting. Time to value, support team sizes, and opportunity costs of what your team members are doing are equally important.**

Let us now break down the above diagram step by step. The first one is the licenses. If PETech decides to use a vendor-provided observability tool, they must pay for software licenses. While this adds to the cost, it often comes with the benefit of having a fully-featured solution right out of the box. This means PETech can start monitoring its systems immediately without waiting for a custom-built solution to be developed. On the other hand, building an in-house platform could eliminate licensing fees, but PETech would still incur development costs, which can sometimes end up being just as expensive, if not more.

Then, there's Hosting Infrastructure. A bought solution usually comes with its own hosting or at least recommended hosting infrastructure. The vendor typically takes care of scaling, maintenance, and upgrades, which offloads a lot of operational headaches. If PETech builds its own observability platform, it will need to consider where and how to host it. This involves thinking about the hardware, cloud services, and the ongoing costs associated with running and maintaining the infrastructure.

Support Costs is another critical factor. Buying an observability platform often includes support from the vendor, which can be invaluable, especially when issues arise. The vendor's support team usually has deep expertise in the tool, which means faster problem resolution. However, relying on vendor support comes with its own price tag. If PETech chooses to build the platform internally, it will save on vendor support costs but will need to have its own support team in place. This means training in-house staff or hiring specialists, which can also be costly.

The Time to Adopt is an important consideration as well. Pre-built observability solutions are typically designed to be implemented quickly. PETech could get up and running in a matter of days or weeks, depending on the complexity of its environment. On the other hand, building a custom observability platform is time-consuming. It could take months to develop and fully implement, during which time PETech may not have the visibility it needs in its systems. This delay could lead to missed opportunities for optimization and longer downtime during incidents.

One of the often-overlooked costs is the Opportunity Cost of Development. If PETech's engineering team is busy developing and supporting a custom observability platform, that's time they're not spending on other critical projects, like building new features for PETech's core products. This can have a ripple effect on the company's growth and innovation. Buying a solution means this time can be spent elsewhere, adding more value to the business.

Staff Enablement Time is another key factor. Regardless of whether PETech builds or buys an observability platform, staff will need time to learn how to use it effectively. However, with a bought solution, the training is often well-documented and supported by the vendor, which can speed up the enablement process. For a custom-built platform, PETech will need to invest time and effort into creating internal documentation and training programs, which can slow down the overall adoption.

Now, let's talk about High Availability SLAs. Observability platforms need to be highly available to provide reliable insights. When PETech buys a platform, the high availability (HA) guarantees are often baked into the vendor's service-level agreements (SLAs). This means PETech can count on the vendor to ensure the platform is up and running around the clock. Building an in-house platform means PETech will need to design and implement its own HA strategy, which can be complex and costly. However, some argue that this also gives more control over the system, as PETech can tailor the HA requirements to its specific needs.

Custom Development Time and Cost are other aspects on which build vs. buy decisions often hinge. Using a vendor tool doesn't mean it's plug-and-play without any customization. PETech will likely need to invest in custom development to tailor the tool to its specific workflows and requirements. However, this is usually far less time-consuming and expensive than building an entire platform from scratch. In contrast, building an observability platform gives PETech full control to create a solution that perfectly fits its needs, but the time and cost involved in such a project can be significant.

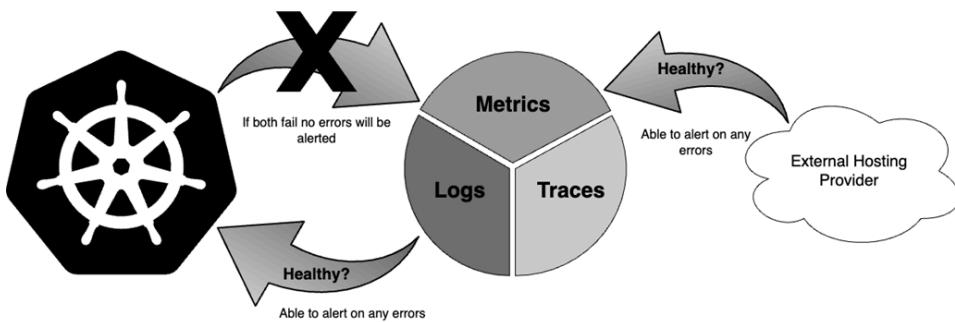
The decision for PETech boils down to a trade-off between cost, time, and control. Buying an observability platform can get PETech up and running quickly, with support, HA guarantees, and a wealth of pre-built features. However, it comes with recurring costs like licenses and support fees. Building an in-house platform offers more control and potentially lower ongoing costs, but it requires a significant investment in both time and money both in the short term and long term.

For many organizations like PETech, the best approach might be a hybrid one: starting with a bought solution to quickly establish observability practices and then gradually customizing or building out specific components as needed. This allows PETech to strike a balance between the speed and convenience of a vendor solution and the flexibility and control of a custom-built platform.

#### 5.4.4 Cross-platform observability

Organizations sometimes take for granted the importance of observing their engineering platforms, like Kubernetes, from an external system rather than relying solely on internal monitoring. If your house is on fire, it is better for you to get out and taking a look as opposed to asking your dog.

In the context of PETech, imagine that they have a complex platform running on Kubernetes, which collects metrics, logs, and traces to monitor the health of their applications. If PETech only relies on the observability tools running within this same Kubernetes cluster, they run a significant risk: if Kubernetes itself goes down, so do the observability tools. This means PETech would lose visibility into what caused the failure and would not receive any alerts indicating the problem because the tools meant to provide those alerts are offline. In Figure 5.10, we show you the approach to observing your observability platform through external means.



**Figure 5.10 Observing your engineering platform should be done external to the platform itself. Otherwise, you risk not being alerted properly about a total outage because the observability system would also be down! Similarly, your observability platform is an equally critical system component and should also be observed externally.**

The abovementioned approach suggests using an External Hosting Provider or an external monitoring service to prevent this situation. Observing the platform from the outside, PETech ensures they can still receive alerts even if their internal systems fail. For example, let's say PETech sets up an external monitoring service like Datadog or PagerDuty that checks the health of the Kubernetes cluster and the observability platform itself. If Kubernetes or the internal observability tools (collecting metrics, logs, and traces) go down, the external service can detect the issue and send an alert. This external monitoring is a safeguard, ensuring that PETech is always aware of critical system outages, even when the internal tools are not functioning.

The notional diagram above shows three key areas—Metrics, Logs, and Traces—which are usually the core components of an observability platform, as we discussed earlier. The arrows and annotations illustrate that these components must be checked for health status. For instance, if PETech's Kubernetes cluster and observability platform are working correctly, they should be able to provide alerts on errors. However, if both fail simultaneously, no errors will be alerted, creating a blind spot. Therefore, using an external service to monitor these aspects ensures that PETech is always in the loop, receiving alerts about outages or system failures, even if their primary monitoring tools are down.

In simple terms, this highlights a critical practice: don't put all your eggs in one basket. Observability tools are crucial, but they also need to be monitored independently to ensure that PETech can respond quickly to any critical issues, minimize downtime, and maintain the health of its platform.

#### **5.4.5 Strategies to Drive Adoption**

Multiple strategies to drive platform adoption can be applied in different scenarios. Evaluate the organization's needs to ensure adoption against the friction that will be imposed on teams when they move to use the platform to find the right balance of enforced vs. voluntary usage.

Driving the adoption of observability in both development teams and the overall business requires a thoughtful approach that balances the benefits of enhanced monitoring with the potential friction that may arise from introducing new tools and practices. This friction could manifest as initial resistance to change, concerns about additional workload, or the need for retraining. Here are several strategies to encourage adoption while minimizing these potential sources of resistance:

1. Start with Education and Awareness  
One of the biggest challenges in adopting observability is helping teams understand why it's essential. You can begin by educating developers, operations teams, and other stakeholders about the benefits of observability. Hold workshops, lunch-and-learn sessions, and training to explain how metrics, logs, and traces can help detect and resolve issues faster, improve system performance, and enhance user experience. When people see observability as a tool that makes their jobs more accessible rather than just another requirement, they are more likely to embrace it.
2. Provide Easy-to-Use Tools and Automation  
To reduce friction, provide teams with easy-to-use observability tools. For example, set up automated metrics collection and log aggregation so developers don't have to configure these for each service manually. Offer pre-built dashboards and templates that teams can use out of the box. If the tools are intuitive and save time, developers will be more inclined to adopt them voluntarily. Automation also ensures that observability practices are applied consistently across the organization without burdening individual teams.
3. Highlight Quick Wins and Success Stories

- Showcase how observability has led to quick wins within the organization. For example, share stories where a team quickly identified and resolved a production issue thanks to detailed metrics or where a performance bottleneck was uncovered using traces. These are the kind of quick wins that can be achieved with observability. Publicize these successes through internal newsletters, team meetings, or intranet pages. When teams see real-world examples of how observability can impact business outcomes, they'll be more motivated to incorporate it into their processes.
4. Offer Support and Make It a Collaborative Effort  
Adopting observability can be a significant change for some teams, so it's crucial to provide support throughout the transition. One way to do this is by creating a central platform engineering team or a group of observability champions. These teams can assist other teams in setting up and using the observability platform, providing guidance, troubleshooting issues, and sharing best practices.  
Encourage a culture of collaboration where teams can share best practices, dashboards, and alerts. This makes adoption smoother and aligns with the shared responsibility model we provided earlier.
  5. Balance Enforced vs. Voluntary Usage  
Decide where observability should be mandatory and where it can be optional. For example, for mission-critical services that directly impact customers, enforce a baseline level of observability with specific metrics, logs, and alerts. This can be done through platform-level integrations that automatically apply observability settings to these services. For less critical or internal services, allow teams to opt in and tailor the observability setup as needed. By striking the right balance, you ensure that critical areas are adequately monitored while giving teams the autonomy to manage their observability practices.
  6. Integrate Observability into the Development Workflow  
Make observability a natural part of the development and deployment workflow. Integrate it with CI/CD pipelines so that new services automatically include observability instrumentation when deployed. Include observability requirements in the definition of "done" for features, ensuring that monitoring, logging, and alerting are considered from the start. When observability is baked into the development process rather than being an afterthought, teams are more likely to adopt it effectively.
  7. Incentivize Adoption  
Offer incentives to teams that actively use the observability platform and contribute to its improvement. This can include recognition through awards, spotlighting teams that have demonstrated best practices, or even tying observability adoption to performance goals. When teams see tangible rewards for adopting observability, they are more likely to participate enthusiastically.
  8. Demonstrate Business Value

- Tie observability to business outcomes to get buy-in from leadership and business stakeholders. Show how improved observability has reduced downtime, faster recovery times, better performance, and a smoother customer experience. Use metrics to demonstrate how these improvements translate into financial benefits, such as higher customer satisfaction, reduced operational costs, and increased revenue. When the business understands the value of observability, securing resources and support for its adoption becomes easier.
9. Iterate and Adapt  
Adoption is not a one-time effort; it requires continuous improvement. Gather team feedback about what works and what doesn't, and use this input to refine the observability platform and practices. Adjust the strategies as the organization evolves, scaling the observability platform to accommodate new services, tools, and requirements.
  10. Encourage a Culture of Observability  
Lachere observability is essential to delivering high-quality software. Promote that observability is a tool and a mindset emphasizing proactive monitoring, learning, and continuous improvement. Encourage teams to view observability as a shared foster rather than a task a specific team owns. When observability becomes part of the organizational culture, it leads to more reliable, performant, and resilient systems.

By implementing these strategies, PETech can drive the adoption of observability in a way that aligns with both development and business goals, reducing friction and enhancing the overall effectiveness of its platform.

The ability to generate comprehensive reports with suggestions for enhancements and remediation of systems based on observability data is a powerful application of GenAI. By combining templated reports with AI-generated suggestions, proactive measures can be taken when critical functions start behaving outside expected baselines.

In chapter 11 of this book, we will discuss GenAI's enhancement of observability in great detail. However, we want to introduce these concepts here first. We see five axes to consider when reviewing the available observability tools and how they use GenAI.

1. Anomaly Detection and Root Cause Analysis
2. Predictive Analytics & Proactive Remediation
3. Automated Incident Response and resolution
4. Security Observability and Threat detection
5. Automated Log Analysis and Insights

All major observability tool vendors are incorporating GenAI features across all these axes. However, your success in using a tool or a combination of tools depends on your understanding, as a platform engineer, how these work and how to apply that knowledge to your environment.

## EXERCISE 5.3: CASE STUDY: EVALUATE THE TCO OF CONTINUING TO SUPPORT YOUR OBSERVABILITY SYSTEM LOCALLY

*In this exercise, we will revisit a specific case study related to PETech. The goal of this exercise is for you to make appropriate decisions concerning build vs. buy decisions. Your task is to pick the correct option. You are expected to write a short justification for your option of choice.*

*Option #1: In this option, the platform engineering team at PETech decides to build a custom observability platform as they came to the conclusion that there are a lot of custom requirements that they could not find in an COTS solution. It involved using open-source tools such as Grafana LGTM, and Prometheus among others. They collected the requirements and decided to build an MVP to get quick feedback. Based on the user feedback, they invested in two full-time developers and a technical product manager to work on the observability platform. Their tasks included the whole lifecycle - define the product features working with the engineering team, building, testing and deploying the products in a continuous manner and ensuring adoption of this platform for all the stakeholders within the organization*

*Option #2: In the second option, PETech decides to buy a third-party SaaS observability platform from Chronosphere with fluentd. This platform came with built-in scalability, cloud-native application integration, comprehensive features for about 80% of what PETech engineers needed. The benefit the platform engineering team found was that they did not need a separate team to maintain this and was able to democratize the decision making. Moreover, they were able to go to market (have all of 80% of the requirements fulfilled) within a span of 2 weeks. They did so by investing in 6 sprints of work for three developers to integrate and rollout Chronosphere with the help of the vendor recommended, third party experts.*

*Which option would you pick? Why?*

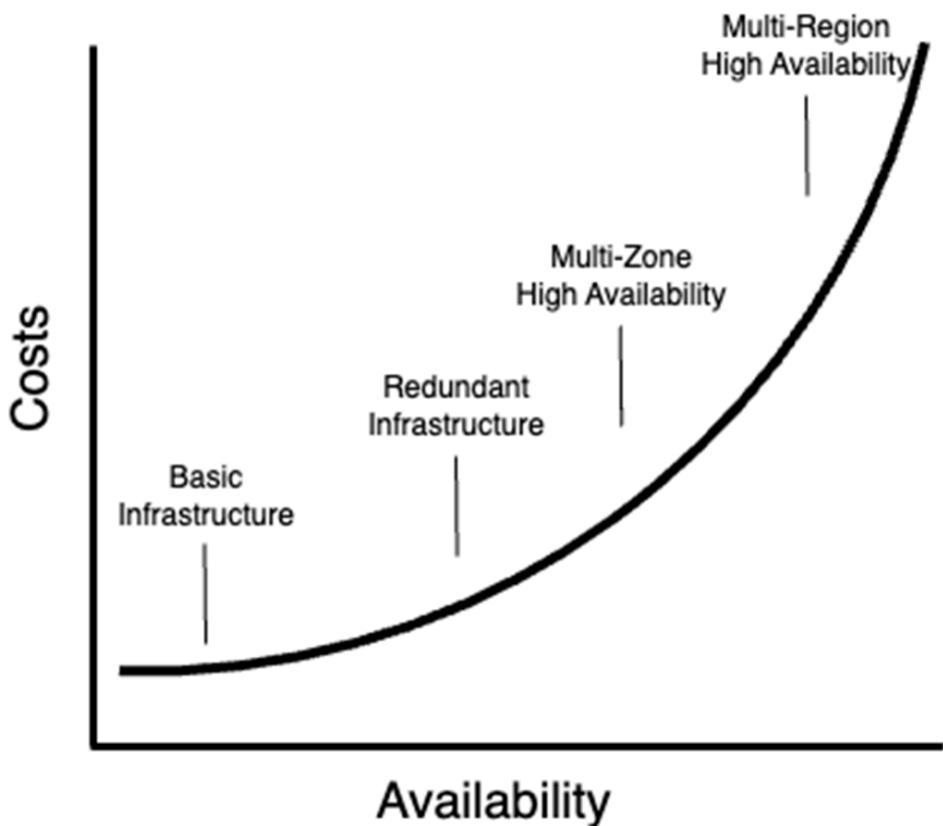
## 5.5 Managing expectations with published SLOs

In this section, we will talk about how to use observability to gain your customer's trust. Specifically we will talk about Service Level Indicators (SLIs), Service Level Objectives (SLOs) and Service Level Agreements (SLAs) and why those signals, goals and contracts, respectively matter. We will conclude this section by talking about doing SLOs as code, which is a common way to do set the objectives in the industry today.

### 5.5.1 Gaining customer trust

In Figure 5.11 below, we demonstrate the typical cost benefit analysis of availability vs cost. Availability refers to the availability of your service to your customers and cost refers to the investments you have to make the availability levels you promise your customers. As you can see the costs increase exponentially as your availability increases.

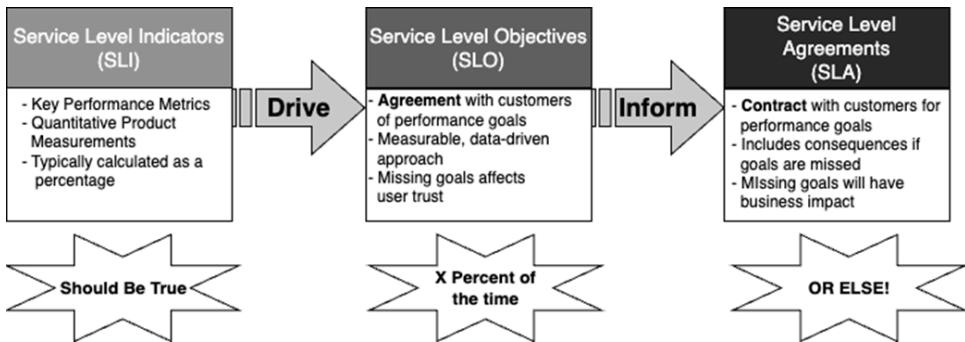
When PE Tech encountered this dichotomy, they first started with modeling the costs based on the cloud infrastructure scaling and the personnel costs, that is needed by factoring the expenditure for multi-zone and multi-region availability. Then they compared this cost against the product revenues. By doing so, they were able to come up with a number that said that if they have to increase their availability of their services from 2 9's (99.99%) to 3 9's (99.999%), the additional costs were in the order of 30%. This also demonstrated to them a drop in profit margin in their product lines by around 10%. Based on this data, they decided to stick to their existing 2 9's level of availability.



**Figure 5.11** While the uptime of a system is one of the most critical measurements to ensure users will trust putting their critical workloads on it, the cost will increase quickly with more availability. Ensure you understand what support more “9s” means and if it’s vital for a given system component to justify the investment.

### 5.5.2 SLIs, SLOs and SLAs

Figure 5.12 explains the concept of *SLIs* (Service Level Indicators), *SLOs* (Service Level Objectives), and *SLAs* (Service Level Agreements) and how they relate to one another. Let us now look at how PETech would handle these three concepts.



**Figure 5.12 SLIs inform SLOs that can be used to notify and publish SLAs. Like engineering platforms, most internal systems will not use SLAs and publish SLOs to their internal teams. This is critical to ensuring adoption because teams must know they can depend on an engineering platform for their workloads.**

SLIs (Service Level Indicators) are pivotal metrics that gauge a service's performance. At PETech, an SLI could be the speed at which their platform responds to user requests. For instance, if the response time is consistently 200 milliseconds, that's an SLI. It's a crucial tool that aids PETech in comprehending the quality of its service.

SLOs (Service Level Objectives): SLOs are targets based on SLIs. They set expectations for what the service should achieve. For PETech's platform team, an SLO might state that the platform should respond within 200 milliseconds 99.9% of the time. This is a way to ensure the service consistently meets standards. If the platform takes longer to answer, it tells the team to investigate and fix potential issues. SLOs are crucial for maintaining trust within the company.

SLAs (Service Level Agreements): While SLIs and SLOs are primarily internal, SLAs are formal customer agreements. They outline what happens if the service doesn't meet customer expectations. For example, PETech might have an SLA with its enterprise clients stating that if the platform response time exceeds 200 milliseconds for more than 0.1% of requests in a month, they will offer a discount or some other compensation. SLAs are less common for internal services, like PETech's engineering platform.

In PETech's case, the engineering platform team focuses on SLIs and SLOs to keep their internal teams happy. They don't typically use SLAs internally, as the consequences are more about improving internal trust and efficiency rather than facing formal penalties. By consistently meeting their SLOs, the platform team at PETech ensures that all internal teams can depend on the platform to handle their workloads smoothly.

### 5.5.3 SLOs as Code

In any modern platform engineering approach, particularly one modeled after PETech's practices, observability is pivotal in ensuring that Service Level Objectives (SLOs) are theoretical benchmarks and practical, attainable goals. For the system's customers—whether internal development teams, business stakeholders, or end-users—trust in the platform is derived from its ability to meet these predefined SLOs consistently. Observability enables this by offering deep insights into the system's performance, behaviors, and anomalies. This transparency fosters trust by allowing platform operators and consumers to track how well the platform adheres to its promises. For instance, in PETech's case, they implemented a multi-layered observability framework that monitored system health metrics and provided actionable insights to optimize performance continuously. By ensuring that the platform's behavior is consistently visible and measurable, PETech established a reliable and predictable service environment, building a solid trust foundation with its users.

When PETech redesigned its platform, observability became a central feature. This was crucial in ensuring that workloads are observable by default, distinguishing the responsibilities between the platform team and its end-users. In Figure 5.13, we show how PETech published an SLO dashboard for their engineering platform services. Incorporating self-service observability features made it easier for developers to onboard their applications without extensive manual intervention. This automated onboarding process embedded observability hooks into every deployed workload, providing instant feedback on application health, performance, and compliance with the platform's SLOs. By making observability a native part of the deployment process, PETech empowered its developers to take ownership of their workloads while maintaining clarity around the platform team's responsibilities. This default observability framework allowed PETech to keep a clear boundary between platform and application management, promoting a culture where platform teams focus on maintaining the core platform's stability and scalability. This emphasis on the platform team's role instills a sense of security in their responsibilities, knowing their efforts are crucial to the platform's success. Meanwhile, end-users (developers) are responsible for their application's performance. This separation of concerns was critical to streamlining operations and enhancing system reliability.



**Figure 5.13 Publishing an SLO dashboard for engineering platform services is an easy way to show success for the platform meeting its objectives. It also becomes a place where teams can quickly check when diagnosing an issue to ensure it isn't a problem with the underlying systems.**

In the figure shown above you can see a typical SLO dashboard designed to monitor the performance of an engineering platform that you can use as a guidance in your own organizations. On the left side, it lists critical metrics such as the target SLO, the current error budget usage rate, and the remaining error budget for the month, providing a clear view of how much of the platform's tolerance for errors has been consumed. It also includes a 30-day overview of the error budget and status indicators for warning and critical alerts, helping teams quickly gauge the platform's health. To the right, various visual elements like line graphs and burn rate charts illustrate the platform's performance over time, highlighting trends or anomalies. This comprehensive dashboard enables platform teams to assess whether the platform is meeting its objectives at a glance and serves as a valuable tool for diagnosing issues efficiently, distinguishing whether problems stem from the platform itself or elsewhere in the system.

#### **EXERCISE 5.4: USE PROMETHEUS DATA TO CREATE SLOS**

In this exercise, you will first define SLOs using the data collected by Prometheus. The SLOs have to be aligned to your business needs, and sufficient justification should be provided for the SLOs.

##### **Objectives:**

1. Understand the importance of SLOs in a business context and how they are a crucial part of maintaining system reliability.
2. Align technical metrics to business needs by selecting appropriate metrics from Prometheus data that reflect business objectives.
3. Justify SLOs based on collected data by writing down the reasoning behind their chosen SLOs aligning with business context and user expectations.
4. Use Prometheus monitoring data to formulate SLOs conduct performance analysis and metric-based decision-making.

**Deliverables:**

1. List of selected Prometheus metrics relevant to the provided business objectives.
2. Defined SLOs (2-3) based on the chosen metrics, with specific targets (e.g., 99.9% uptime, <200ms latency).
3. Justification document explaining why each SLO was selected and how it supports business goals.
4. Prometheus queries used to extract the relevant data for monitoring the SLOs.

## 5.6 Summary

- Observability is critical for engineering platforms themselves as well as the products these platforms are supporting.
- Observability is not monitoring or looking at metrics, logs, and traces. Instead, it goes beyond that by predicting a system's internal workings by examining its external behaviors.
- Unknown-unknowns are the actual valuable pieces of information you need to be effective at observability.
- Building an observability system includes these five critical steps (1) Understanding your observability goals (2) Buying a tool or building one that provides the necessary framework to achieve these goals with a well-defined and predictable user experience framework (3) Instrumenting your code (4) Ensuring automated and verifiable data collection from all the sources (5) Establishing a transparent shared responsibility model
- It's important to understand that building a world-class observability platform is an evolutionary process, not an immediate achievement.
- Building a platform is not enough. Getting the cultural aspects right to ensure your developers adopt the platform is challenging but crucial.
- Observability is critical to providing SLOs that will establish trust in the system for its customers.
- Workloads deployed on the platform should be observable by default with self-service.

# ***6 Building a Software-Defined Engineering Platform***

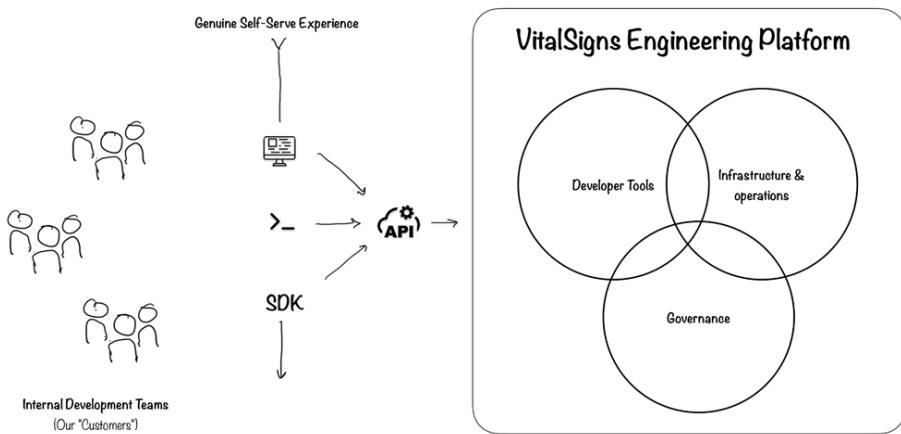
## **This chapter covers**

- Starting to Build our own Example Engineering Platform
- Infrastructure Pipeline Orchestration Practices
- Bootstrapping software-defined Infrastructure Automation
- ManagingCloud Account Automation Identities and Permissions

VitalSigns.online is a fictional healthcare tech startup based in North America. They offer various mobile and web services that help people track their health vitals and share this info with their doctors when paired with consumer electronic devices. Their web services have always been open to third-party developers and business partners, and they're keen to keep and grow this feature. VitalSigns plans to roll out even more services in the coming years. They're focusing on building their tech as APIs, aiming to create individual health-data collection apps and combined experiences quickly. This approach is about providing doctors with better data and helping people achieve better health outcomes without frequent office visits.

The company has seen incredible success and growth in the four years since it started and expects to have over a hundred developers soon. But without a clear strategy beyond a mix of tech silos and a DevOps team, developers at VitalSigns are now spending half their time on lead-time planning, coordinating with other teams to get DNS entries, firewall rules, storage, compute capacity, monitors, alerts, pipeline changes, and everything else needed to build, deploy, and operate their software, often under tight deadlines. Maintenance and operational issues are a constant headache and aren't seen as adding much value. Unsurprisingly, product incidents are rising, leading to frustrated customers and higher support costs.

Imagine we are a team within VitalSigns tasked with creating a better solution to these challenges.



**Figure 6.1 We want to create an internal product that provides a genuinely self-service experience where developers (our internal customers) can imagine, design, build, release, and operate their applications with agility, high engineering quality, greater operational resiliency, and confidence in meeting compliance requirements, yet without all the usual engineering friction they usually experience. In other words, we will deliver an Engineering Platform.**

In the next couple of chapters, we will dive into applying platform engineering practices to create the foundational parts of an engineering platform for our imaginary company. These foundational components are crucial for any effective platform. You'll see how platform domains relate to the pipelines we build, how we can extend the Kubernetes control plane for more value, and what a self-serve user experience should look like.

## 6.1 Start Building Our Own Example Engineering Platform

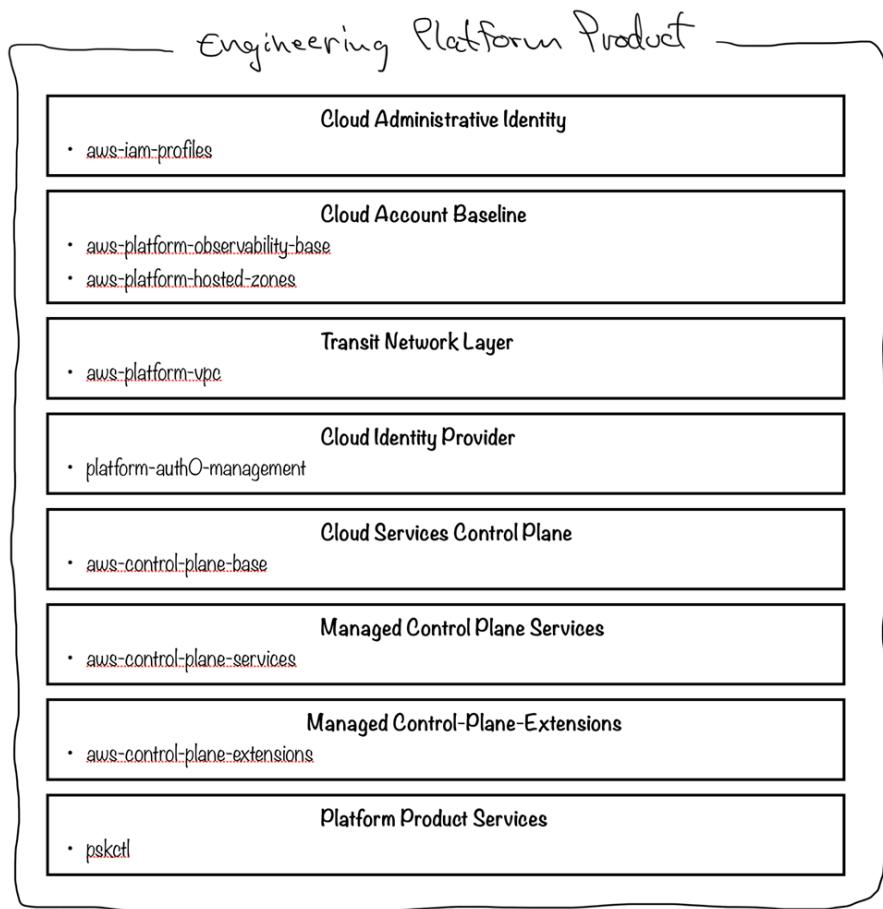
Let's assume this will be a brand-new product. While this is a common approach for learning exercises, there are plenty of good reasons for an enterprise to consider doing the same when building an engineering platform. The most pragmatic reason is described best by Gall's Law:

A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple system. [\[1\]](#)

By now, nearly every organization has been using many, if not all, of the various technologies that will go into an engineering platform. They have teams scattered everywhere using Kubernetes, creating infrastructure using Terraform, using Git, and “doing DevOps.” These implementations are often difficult to adopt because they’re either too focused on the needs and preferences of a small group of users, or they were developed without user input and are instead optimized for the needs of the team providing the technology. In other cases, while the implementation is meant for general use, it is managed across many different traditional IT silos where API access and self-serve experiences were not originally considered and by teams without the resources or experience to evolve. Changes need a lot of planning, and long delays are normal. In all these scenarios, user experience is rarely among the top priorities.

Organizations trying to create a developer platform while sticking to a traditional IT or their original DevOps model spend a lot of time and money, only to realize they are not achieving their goals and have to start over with a unified team in a greenfield setting.

With effective product ownership and platform engineering practices, a single team can successfully manage the internal domain boundaries within the product while they are initially integrated and an MVP version is created. Then, as the product evolves and greater velocity in the platform features is needed and scale becomes a factor, sub-domains can be handed off to other similarly organized teams effectively since the architecture has the right low-friction, loosely coupled boundary between the various parts of the platform product.



**Figure 6.2** The engineering platform product domains (which we talked about in Chapter One) also show the basic dependencies and almost exactly the order in which we'll set up the product infrastructure pipelines. This diagram shows the pipelines we will build in each domain to create the foundation of our engineering platform at Vital Signs.

But first, let's talk about the prerequisites.

## 6.2 Prerequisites to Getting Started

**NOTE** Our example platform will be built using AWS as the cloud infrastructure provider. Be conscious of the cost as you start these platform-building exercises. With careful management, such as de-provisioning resources, clearing data storage when not in active use, limiting the work to a single platform environment, and other similar strategies, you may be able to stay within the AWS free tier, but it can be challenging. A Kubernetes cluster supporting a service mesh and other platform technologies requires instances larger than micro. Presently, even at the small scale of a personal platform, fully sustaining the various resources of a two-environment platform 24/7 can run from \$600-800/mo or higher. Because these are learning exercises, you don't need 24/7 uptimes, and you can get these costs dramatically lower using the aforementioned careful management. Still, potentially significant costs may be involved, and consideration must be made at the start regarding access to the necessary resources. If you are applying these principles at your place of work in the actual delivery of an engineering platform, then the cost has already been budgeted. Alternatively, many organizations fund limited use of cloud resources for skills improvement and learning, so you may have the necessary access to Cloud provider services through your employment or educational institution. If you do not have access to cloud provider resources, many aspects of the engineering practices within the exercises can be explored locally using tools such as Minikube.

What resources do we need to get started?

Categories	Used in Example exercises
 Cloud vendor account	
 Distributed source version control	
 Backend location for terraform state files	
 Secrets store	
 Pipeline orchestration tool	

**Figure 6.3 Resources we will need to begin building our engineering platform product foundation.**

Besides the cloud accounts where the platform infrastructure resides, you will notice that these tools are also tools the platform product includes for use by platform customers. As platform engineers, we will use these tools ourselves to deliver the platform.

Not every potential developer tool is needed initially, so which tools are needed to bootstrap an engineering platform effectively?

- Source code version control
- Secrets store
- Infrastructure state store
- Pipeline orchestrator

Starting from scratch, we have a bootstrap challenge. How can we deploy the first configuration in a software-defined manner using source control, managed secrets, state store, and a pipeline if we must first deploy these tools before using them? Right away, you can see the accelerating impact of using SaaS tools as the solution to the bootstrap challenge.

The reference code examples [\[2\]](#) in the *Effective Platform Engineering* GitHub organization will demonstrate several highly effective tools.

- GitHub
- 1Password
- Terraform Cloud
- CircleCI

Using these tools is not required to apply the principles in this book, though it will allow you to get the most from the sample exercise solutions.

## GETTING STARTED WITH THE EXAMPLE TOOLS

In an Enterprise setting, each tool would typically be integrated using an SSO solution. We will talk more about how that fits into the product experience in the section on Customer Identity Provider. But for now, let's go ahead and set up access to the initial tools as an individual user. The respective tool's product documentation provides detailed instructions for performing the following steps.

If you do not already have one, create a free personal account on [github.com](https://github.com). Then, create a free-tier GitHub Organization for our imaginary VitalSigns company, in which we and all the VitalSigns platform developers will be members. Create a GitHub team called `platform-team` to represent our product delivery team for the VitalSigns exercises. Later, we will use GitHub Teams and team membership to manage access permissions. Add yourself to this team. Create a personal access token (PAT) and be sure to upload your personal SSH keys, and enable support for signed commits.[\[3\]](#)

Create a free-tier Terraform Cloud organization. Go to the settings area (from the left-hand menu) within the organization, create a team, and add yourself to the team. From the team settings page, also generate a team API token.

Create a free-tier CircleCI organization and link it with our GitHub organization. Generate an access token.

Lastly, 1Password offers individual plans for less than \$3 per month. Create a dedicated 1Password vault for these exercises then go to the Developer options and generate a service account credential with read and write permissions.[\[4\]](#) Or, if you are using some other secrets management tool, also have the access credentials for pipeline usage available.

Store all of the above access tokens in this vault.

Most of the SAAS tools used in the example exercise solutions offer a free tier adequate to cover the exercises in this book or very affordable personal options, and alternative tools will often be discussed. Occasionally, we will use alternative tools to demonstrate the differences among effective choices within the example exercise solutions.[\[5\]](#)

For any tool where you need an access token to use in automation, such as in our pipelines, if the tool doesn't allow you to create something like a team or organizational level token, where anyone on your team can manage it, then you are left with needing to create a *personal* access token. This introduces a problem. What if you leave the company or take on a new role with a different team? Either of these events can cause your personal access token to be revoked, and any automation that depends on the token will break.

The two most effective ways of dealing with this situation are:

**Service Accounts.** Sometimes called machine users, these are identities created within the appropriate system in the same way as human users, except that no single person has control over the identity. It becomes a team resource with the username and password stored in the Team secrets store for management. Often, systems provide an actual feature designed to support this type of User. We will use an example in AWS later in this chapter. Just for these exercises, creating and using a personal token when needed is fine.

**OpenID Connect Tokens.** Many tools and most cloud resources, such as AWS, also provide a means of establishing direct trust between other tools or cloud resources. This approach requires more behind-the-scenes automation to create a self-service experience for Platform users, but it is an option.

### 6.2.1 Developer Tools Selection Criteria

The tools you choose aren't just neutral decisions—they can either support good platform engineering practices or get in the way. The best place to start is by talking to your customers to understand their needs and preferences. But when it comes to making the final choice, it's important to include platform engineering considerations as part of your criteria to ensure you build and maintain a solid self-serve experience.

## Choosing Tools



Smaller, focused tool, exceptional in its implementation and interoperates easily

Strongly domain-bounded and can be implemented with replacement in mind

Qualified SaaS option that offers no or low cost access for testing and experimentation before purchase

Has an API based on modern standards with good documentation that provides functional examples and does not require the use of a specific programming language

The API provides access to all application functionality

Data stored in the product is secure at rest and in transit, and readable and writable through the API

Can be integrated into our platform's user authentication and authorization strategy

Visible community

If self-managed, can be fully installed and maintained using software-defined practices.

large, monolithic product that tries to meet many different (or even all) developer needs

No or limited ability to test and experiment with tool before purchase

Requires manual configuration by Users

No or limited API access, or the API is based on out-of-date or otherwise unusual technology

Poor data security or API access to data maintained within the tool

No ability to make support our platform's user authentication and authorization strategy

Poor documentation, no visible community

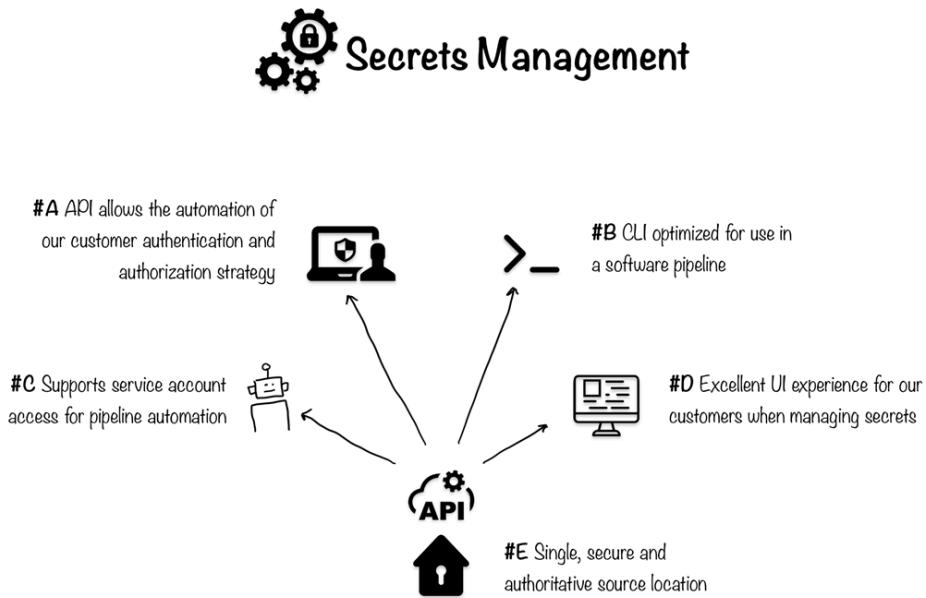
Must be deployed and maintained by the buyer, without support for a software-defined implementation

**Figure 6.4 Regardless of which tool we are examining, there are general criteria we should carefully consider in determining whether or not the tool will integrate effectively and help create the experience we want for our platform.**

Where available, using high-quality, secure SaaS development tools is one of the most accelerating and long-term efficiency choices available, paying dividends not just at the start but continuously over the Platform's life. Time spent deploying, operating, securing, or maintaining any of these tools is time taken away from doing strategically valuable work. Companies routinely underestimate the cost of self-managed or poorly administered tools.

### ADDITIONAL CRITERIA FOR SPECIFIC KINDS OF TOOLS

In addition to these general criteria, each tool we implement has some additional attributes that will make it much easier to integrate and maintain a self-serve user experience.



**Figure 6.5 We are using the same tools we will integrate within the Platform for our Customer's use. To be able to create the truly self-service experience we want, in addition to the general criteria, we need the following from our secrets management tool.**

**#A** While this may seem somewhat obvious, it can be difficult to achieve in practice. For instance, since we are building our platform on AWS, you may ask, what about using AWS Secrets Manager? Recalling the discussion of customer identity in Chapter One, we want an independent customer identity model. Think of the complexities of using only AWS IAM to map individual secrets in Secrets Manager based on a user's team membership that is maintained in an external source. In practice, you must create a custom API to provide this experience. However, several tools are available that more closely align with our authentication and authorization goals.

**#B** What do we mean by a CLI optimized for use in a software pipeline? Pipelines are not the only places we will interact with secrets, nor will a CLI be the only method. But, pipeline usage is critical. At a minimum, we need a command line method for setting environment variables based on values in a file and injecting secrets into template files.

## SETTING ENVIRONMENT VARIABLES FROM FILE CONTENTS

This is the most common pipeline use case. A clean, file-based method of sourcing secure information into a shell process should exist.

For example, using the 1Password CLI, you can inject secrets defined in a file into a shell process in a single command.

```
$ op run -env-file my_app.env -- bash_script.sh
```

Needed secrets would be defined in the file my\_app.env as follows:

```
export SNYK_TOKEN=op://vault-name/snyk/api-token
export TFE_TOKEN=op://vault-name/terraform-cloud/team-api-token
export SLACK_BOT_TOKEN=op://vault-name/slack/post-bot-token
```

## TEMPLATE INJECTION

There should be direct support for populating template files with secrets directly from the secrets manager.

For example, a common method of providing credentials is through a credential file. Creating a template of the required file that can be populated and written to the correct location should be a single step. An example using Terraform Cloud would be:

```
$ op inject -i terraformrc.tpl -o ~/.terraformrc
```

Where the contents of the --input file are:

```
credentials "app.terraform.io" {
  token = "{{ op://vault-name/terraform-cloud/team-api-token }}"
}
```

Many widely used secrets management tools offer pretty poor support for the use cases above. Unsurprisingly, this has inspired some open-source projects to solve this problem [\[6\]](#).

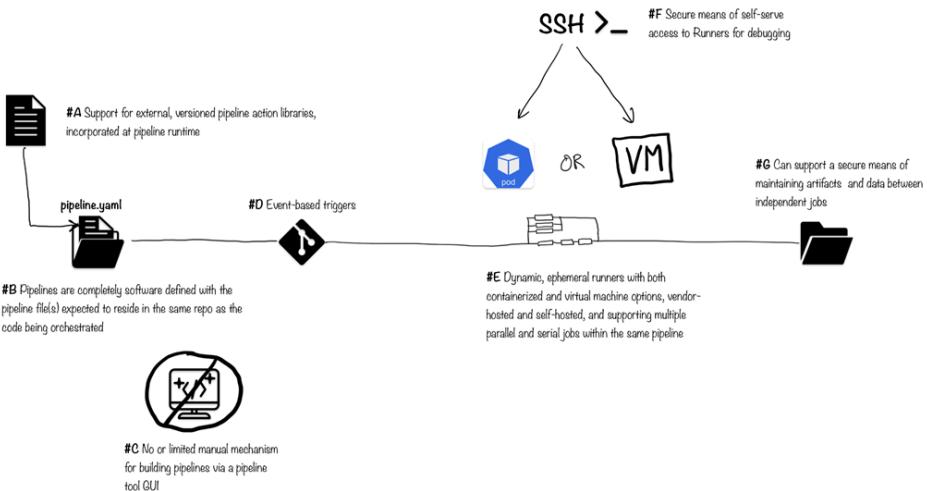
So long as the secrets management tool meets the general *software selection criteria* listed at the start of this section, it is not difficult to create and maintain a simple CLI that provides the desired experience. It is worth the effort if the tool or an open-source alternative does not provide the right experience.

**#C** Nearly all interactions with the secrets manager will be through automated integration, which will require some form of machine-to-machine credential. Usually, this means generating an API token or similar service account credential.

**#D** Development teams will manage their own secrets. While this mainly involves using the secrets in integration and deployment pipelines or configurations, the interface for users to add or update secrets is also essential. Using a tool with a good interface means you won't need to custom-build one.

**#E** There should be a single source of truth for secure values. Pipeline steps, deployment events, secrets services deployed onto clusters, and all other automation should pull secrets from a single location. Not only does this reduce the opportunities for error, but it greatly simplifies sustaining secure configuration and good security practices such as automating the rotation of credentials.

## 📍 Pipeline Orchestration



**Figure 6.6 Our pipeline tool will determine the quality of the continuous integration and delivery experience and needs specific additional attributes beyond the general selection criteria.**

**#A** Most modern pipeline tools have this feature. In CircleCI, these are called Orbs. In GitHub Actions, they are shared Actions and Workflows. These shared code libraries are the key means of standardizing common activities or performing all the non or cross-functional requirements. This must be a capability of the pipeline code itself rather than an extension to the pipeline server or runners. In other words, references to the shared pipeline code occur in the pipeline code itself, which lives in the repository with the infrastructure or application code. Modules or plugins that are instead loaded or installed on an orchestration server or runner are not the same thing and result in brittle, high-maintenance pipelines.

Most popular pipeline tools encourage community contributions to their growing library of version pipeline code (such as Orbs or Actions). Using these resources involves the same security practices needed for any shared library. With few exceptions, the most valuable Orbs will be those created in-house to support your workflow needs directly. The internal development and management of customer pipeline libraries must also follow traditional software development practices such as integration tests and versioned releases.

**#B** Teams must own their own pipeline code. Pipelines may be required to include remote Orbs or Actions to support cross-functional requirements, yet the CI pipelines triggered by source code changes must live in the same repository as the code being tested. Where teams using the same language and architecture can easily use a common set of build or test steps, make those steps available through shared code referenced by the individual repository pipeline. Developer ownership is also a requirement for deployment pipelines. The essential outcome is that customers (developers) are never blocked waiting for another team to make needed changes to their pipeline, nor should unexpected breaking changes get pushed into their pipelines. Changes in shared pipeline code must be introduced through version changes. Teams intentionally adopt the new version and will not be surprised even if the new version includes a breaking change. If a team needs a change in a shared pipeline resource and it's not happening fast enough, they can fork the shared code and make the changes themselves rather than remain blocked.

**#C** While not strictly necessary, this is an example of how the lack of a feature can effectively constrain complexity while not limiting effectiveness. For the same reasons we don't want people going into the AWS console to make infrastructure changes, it is a poor practice to make manual pipeline changes. Where no manual option even exists, there is one less problem to manage.

**#D** Optimally, triggering a pipeline via the pipeline tool should be limited to retries. This is not an absolute requirement, but triggering a pipeline within the pipeline tool itself should be treated as an antipattern. Confidence in continuous deployment automation is built upon the assumption that specific events will always occur in a specific order.

**#E** Slow pipelines mean wasted time. Installing packages and configuring runners as a stage in a pipeline is where significant time may be recovered through pre-configured runners. Runners (or Executors as they are called in CircleCI) are where our pipeline jobs, steps, and commands occur. Having the environments already set up with the tools, testing frameworks, or any other packages the pipeline will need means the pipeline does not need to take the time to download and install those things. If you can shave 5-30 minutes off every pipeline, imagine how much time you could get back across a large organization with hundreds or thousands of developers, most of whom expect their pipelines to run multiple times daily. As a feature of your engineering platform, maintain a set of shared runners that includes a common base runner that has all the packages that need to be included with every kind of pipeline (such as the secrets manager CLI), along with language or context-specific runners that have build, test, and reporting tools. And include Runner starter kits that customers can use to build upon the shared images to create fully customized runners that do not require any package installation or other configuration at pipeline runtime.

**#F** A secure means of enabling customers to access pipeline runners for debugging is a significant time-saving feature. A pipeline step working locally but failing in the pipeline is a very common event. So long as feature #E is true, you can build your own solution should the tool not already offer this capability, though time is saved if you don't need to.

**#G** Though not as frequently needed as #F, this is still valuable and, if already provided, will save the effort of building a custom solution.

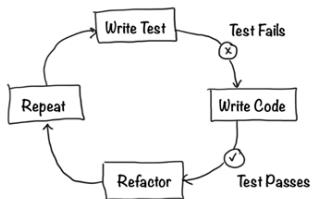
## EXERCISE 6.1: ASSESS A PLATFORM DEVELOPER TOOL ACCORDING TO THE SOFTWARE SELECTION CRITERIA

Select a developer tool from one of the *prerequisite tool* categories. This can be a tool used in the example solutions or another tool you are interested in using. Use the general tool selection criteria and the additional requirements, if they apply, to assess whether the tool could be effectively integrated into a platform.

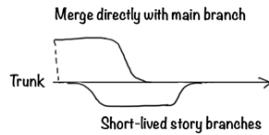
### 6.3 Infrastructure Pipeline Orchestration Practices

Before starting our first pipeline, we should define the platform engineering software development practices that make up a well-architected infrastructure pipeline.

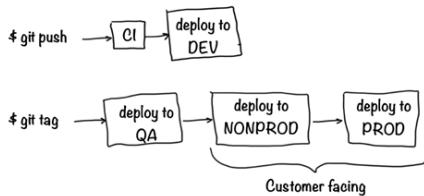
## #A Use test-driven development (TDD)



## #B Use trunk-based development (TBD)



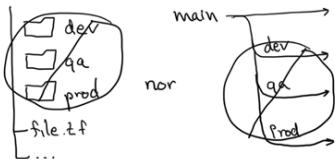
## #C Test and release pipelines follow a set path to production



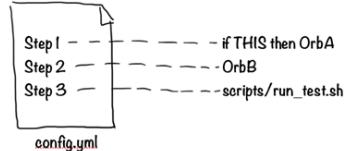
## #D Only one value is maintained in pipeline tool's ENV values

Environment Variables	
Environment variables are available to any job that request Environment Variables documentation.	
Name	Value
OP_SERVICE_ACCOUNT_TOKEN	*****JSJ9

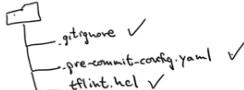
## #E Keep infrastructure code DRY



## #F Build dumb pipelines and smart scripts



## #G Use local code protection and quality practices



**Figure 6.7 Well-architected Infrastructure-as-code pipelines are no different from well-architected regular software pipelines. This is why we prefer the term 'software-defined infrastructure' to differentiate between simply using an IaC framework like Terraform and the software engineering practices that are a part of sustainable, high-quality software developer lifecycle management of an engineering platform product.**

**#A** In an infrastructure pipeline, the tests are less about confirming that the infrastructure SDK performs as expected and more about acting as a general smoke test after deployment and (more importantly) part of a nightly check to detect configuration drift caused by a manual change.

**#B** Preferably, the Platform team engages in pair programming and pushes directly to trunk. Short-lived story branches used to organize changes or to take advantage of the Pull Request process built into GitHub are fine, but you should still be merging changes frequently main, at least daily and preferably more often.

**#C** CI in this context refers to static code analysis, and deploying to DEV means the first infrastructure test environment. Naming the first environment *development* can create confusion, as this test environment is only for the builders of the engineering platform. It is not an environment a platform customer ever accesses. Sandbox might be a better name for the first environment.

Requiring a fixed, recurring release path to production is essential for maintaining the environmental stability needed for your customer to engage in continuous deployment successfully. Continuous deployment of application software depends upon the health and ‘sameness’ of production and nonproduction environments, in addition to the rigor of the release process and automated testing. Successful continuous deployment of infrastructure has the same requirements.

Remember that every customer-facing environment is production from the platform delivery team’s point of view, just as every customer’s AWS account is a production environment from AWS’s point of view.

**#D** Don’t store all the secrets a pipeline needs in the pipeline tool’s ENV value service. Pipelines pull in values from the secrets manager as needed. This practice will make us much more successful in the automated management and rotation of all forms of access credentials, reduce the security risks of human error through the repeated duplication of stored values, and significantly reduce the complexity of moving to either a different secrets manager or a different pipeline tool. The screenshot in the diagram is of a value entered into a CircleCI context that is associated with the GitHub Team *platform-team*. Only the team’s credential for accessing their secrets is retained in this secure location.

**#E** None of our infrastructure code should be duplicated into folders or long-lived branches to account for environments. The minimal differences between environments are accounted for only through different configuration parameters (tfvars in our case). This is a standard practice in general software development, coming from years of painful learning (in many cases). Don’t be tempted to reintroduce such strategies with infrastructure code as though it will be exempt from the consequences. It won’t.

**#F** In other words, the pipeline or workflow triggered by the git event should contain only ordering logic. The individual steps of the pipeline will call external, versioned pipeline code (such as CircleCI orbs or shell script libraries) or local scripts for logic unique to the pipeline. This pipeline architecture has two important benefits. First, the *smart* shared code provides an effective, sustainable means of sharing standardized pipelines, keeping pipeline code DRY across teams, and enabling developers to pass nonfunctional compliance checks successfully. Second, the relatively *dumb* pipeline code preserves a realistic ability to change out the pipeline tool should more effective or affordable options arise. (This is one example of *preserving domains of change* as described in the foundational architecture chapter.)

**#G** Use .gitignore, pre-commit hooks that run lint and style checks with each commit, and all the usual repository-level code lifecycle practices.

If you have worked in software development, you will recognize that this list is pretty standard for software pipelines. Perhaps it is because software-defined infrastructure is relatively recent, but it is common to find infrastructure engineers dismissing many of the hard-learned lessons of software development, not realizing that these apply to infrastructure code. The GitOps movement that began a few years ago has been a mixed bag of good and bad practices and has led many people to have to re-learn all the lessons traditional software practitioners learned over the years of working with version control and pipeline orchestration. Managing code in a version control system and triggering automated build and deployment from changes in the source code is a well-understood practice. Labeling the practice as *GitOps* in an infrastructure context does not bring something new to the domain.

There's one more thing about infrastructure code that's worth mentioning early on. When you're building an engineering platform product, the architecture, engineering choices, and implementation need to focus solely on optimizing for the platform itself. Due to the history of DevOps and infrastructure-as-code initiatives in many companies, there's often a culture of evolving each bit of Terraform code to support every possible use case. But many of those use cases will have nothing to do with the engineering platform. Some will need to stick around indefinitely but can't fit into self-serve patterns, or they're specific to just one team. Trying to create an architecture and code base that supports both an engineering platform and a bunch of unrelated use cases is not only impractical, but it will soon result in the pace of delivering capabilities and features for the platform as slow and cumbersome as the traditional IT process it's meant to replace.

## CHOOSING AN INFRASTRUCTURE-AS-CODE FRAMEWORK

In a cloud setting, infrastructure code should be declarative by design. Why?

With very few exceptions, a cloud vendor's APIs are intentionally architected to be declarative. For example, we don't control how AWS sets up software-defined networks, deploys an RDS Postgresql database on an AWS-managed server (EC2 instance), or handles most tasks in any step-by-step way. When we interact with cloud vendor services, we just give them configuration values that get stored in some database we never see. Behind the scenes, an API does the actual setup and configuration based on the parameters we have provided. The frameworks provided by the cloud vendors themselves demonstrate this characteristic. Not surprisingly, many modern infrastructure-as-code ("IaC") frameworks are declarative by design.

Compare this to earlier IaC frameworks that had their start when the primary activity (often the only possible activity) was configuring virtual servers. (E.g., Chef, Puppet, Ansible) Certainly, there was an end-state in mind, but the underlying purpose of the framework was to manage the steps required to arrive at a desired configuration. When this is still needed, then use tools well suited to those *imperative* configuration requirements.

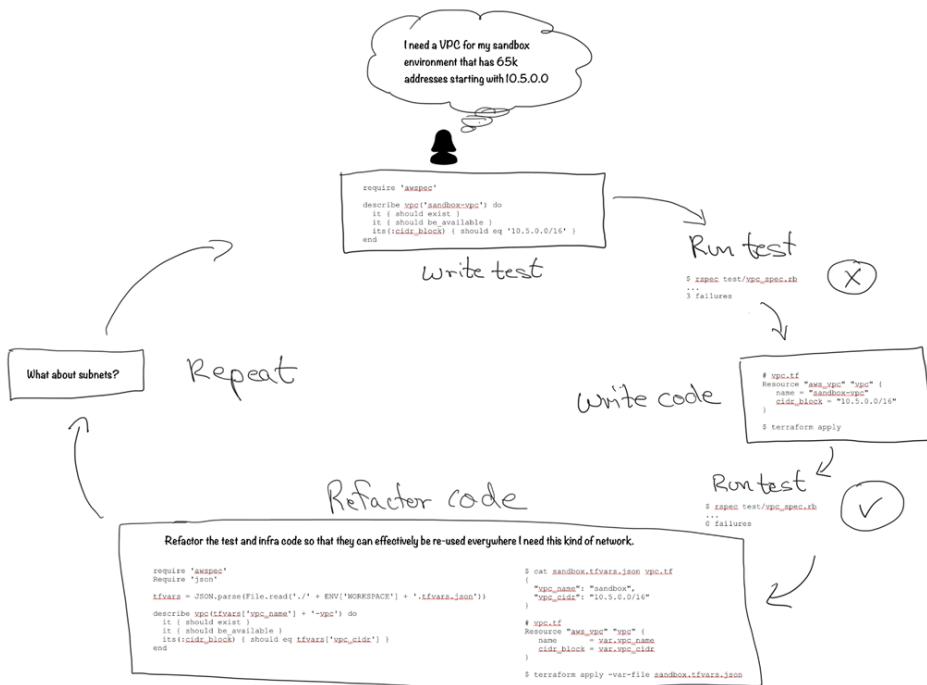
But, particularly with the rise of cloud-native architectures, the vast majority of IaC tasks involved in using and managing cloud vendor infrastructure are highly declarative in nature. The optimal framework provides the most human-readable format for passing structured data to the cloud vendor's APIs while introducing the least processing overhead (wait time) beyond the vendor APIs' wait time.

We will use Terraform for our direct vendor infrastructure configuration as it scores high marks against our criteria.

When we use Terraform to build and maintain an internal product, there's one key difference compared to how a typical DevOps or Site Reliability Engineering (SRE) team might use it. DevOps teams must focus on making their Terraform code flexible enough to handle a wide range of use cases. One week, they might need to provision a network (VPC) for one team with specific configurations, and the next week, it's a completely different setup for another team. Their priority is to support various requests efficiently, so they structure their Terraform code to make their job easier.

On the other hand, when we're building the internal components of an Engineering Platform, we're creating resources for a much more opinionated use case. The Platform team is responsible for maintaining and evolving the platform over its entire lifetime. If we had to design our Terraform modules or resource definitions to support every possible use case across the company, every change would need extensive testing. The structure of our Terraform code would shift to accommodate scenarios unrelated to our product, slowing down changes and making evolution within the code much more difficult and probably rare. The concern is there even when we choose to use a Terraform resource maintained by someone else. Most of the officially maintained modules related to cloud infrastructure are meant to be generic and provide access to all the configuration available for the resource. This makes it a low risk choice to use. But when assessing outside modules, in addition to security considerations, try to understand if there is some level of opinionation within the module that, should the goals of the maintainers change, would negatively impact our product.

### 6.3.1 Test-Driven Development of Infrastructure Code



**Figure 6.8 In terms of the workflow pattern used by engineers, Test-Driven Development (TDD) in infrastructure code is the same as other software. Write the tests. Run the tests. The tests are failing at this point. Then, start writing and applying the infrastructure code until the tests pass. Finally, refactor the code for dryness, readability, performance, etc.**

We said in the previous section that infrastructure is usually highly declarative. The action is happening inside the vendor APIs. But if this is the case, how much value is there in writing tests? You're not writing the code that technically performs the actions. You can write integration tests to confirm the infrastructure configuration, but isn't that just testing whether or not the AWS API works as advertised? True. If that were all testing inside of infrastructure pipelines could tell us, then it wouldn't be especially useful.

However, integration tests for infrastructure code are still valuable.

First, exhaustive integration testing is very effective when building and maintaining a reusable module, as in a Terraform module. A traditional integration configuration test is the proper test to assess the module's internal workings. In this case, while the module also just accepts a set of configuration parameters, the specific resources provisioned are an opinionated decision of the module creator, not the cloud provider. As the developer of a reusable module, you will create a CI pipeline that applies the module in a dedicated cloud account or project with integration tests to confirm that your module correctly provisions the desired resources and configuration you intend.

Second, the TDD workflow provides an effective means of demonstrating story acceptance criteria. For example, the story might be that the subnet used for managed nodes in the control plane clusters supports the use of Karpenter, and an established naming convention will apply. In this case, while the AWS-provided VPC terraform module can be counted on to successfully create and configure the VPC based on the parameters you provide, you will nonetheless want to confirm that tagging of the subnet uses the correct naming convention. The snippet below is an example of how the subnet parameters are provided to the VPC module.

```
...
private_subnets      = var.vpc_private_subnets
private_subnet_suffix = "private-subnet"
private_subnet_tags   = {
  "kubernetes.io/cluster/${var.cluster_name}" = "shared"
  "Tier"                                     = "node"
  "karpenter.sh/discovery"                   = "${var.cluster_name}-vpc"
}
```

An effective test is to confirm that the required tag values match the naming convention pattern, automating the proof that the acceptance criteria were met.

Third, integration tests can be practical as recurring tests aimed at early detection of configuration drift caused by actors outside the infrastructure pipeline. In a way, infrastructure code has the shortest potential shelf-life of any software because no matter what the code configures, there is always a manual way to change the infrastructure. Someone makes a change through the console, and as a result, the actual state of the infrastructure does not match the desired state. There's very little chance of that with my Java app. Recurring configuration tests can be effective at detecting unexpected changes that can eventually become breaking.

Finally, don't overlook the long-term benefits of getting into the habit of writing tests first. Writing the tests first forces you to work through your understanding of the intended outcome of the infrastructure code you are about to create. While creating the actual code (terraform in our case), minor, unintended changes in direction, assumptions you didn't realize you were making, and even confirmation bias in favor of the implementation approach adopted can all creep into your thinking with the result that tests written after primary development are statistically more likely to return false positives. In other words, the tests Pass, yet the logic of the test is flawed and should have resulted in a Fail based on the actual desired outcome.

Several tools exist for infrastructure configuration testing. Let's look at an open-source example for AWS used in the diagram above. [Awspec](#)<sup>[7]</sup> is an extension to the Ruby RSpec<sup>[8]</sup> framework that incorporates the AWS Ruby SDK to support comprehensive testing and has predefined assertions for the most commonly used resources.

One of the first things we will do in an upcoming section is create AWS IAM Roles. Here is an example of testing for one of the built-in AWS roles, AdminUsersRole.

```

require 'awspec'

describe iam_role('AdminUsersRole') do
  it { should exist }                                #A
  it { should have_iam_policy('AdministratorAccess') } #B
end                                                    #C

```

#A `iam_role` is a pre-defined assertion in `awspec`. I only need to specify a role name.

#B This example tests for two things. First, does the named role exist?

#C And second, does it have an attached policy called `AdministratorAccess`?

InSpec [\[9\]](#) is another tool originating from Rspec and now supports multiple clouds. The following example demonstrates a test for a Google Cloud project's existence and active state.

```

describe google_project(project: 'my-gc-project') do
  it { should exist }
  its('lifecycle_state') { should cmp 'ACTIVE' }
end

```

Notice the similar *Rspec* structure.

## **EXERCISE 6.2: CREATE AND RUN A CONFIGURATION TEST AGAINST A BUILT-IN AWS ROLE**

Use your personal, administrative-level AWS credentials and run the above `iam_role` check against the AWS account you use for the exercises. Then, change the policy name to misspell it to test for a policy we know doesn't exist and see how Rspec reports the failure. (Exercise code samples are available in the companion code.)

### **6.3.2 Static Code Analysis**

Like general programming languages, static code analysis of IaC code is an effective means of maintaining general code quality and sustainability. However, as discussed in the TDD section, IaC is highly declarative, so things like test coverage, cyclomatic complexity, and other more traditional imperative code checks are not particularly feasible or valuable. However, this still leaves some valuable analysis that can be performed.

1. Syntax
2. Style and best practice conventions
3. Static security analysis

There are dozens of paid and open-source tools in this space. Whichever tool we use, we want early feedback, locally and in our CI pipeline.

Let's look at a specific Terraform example for each category.

1. Terraform [\[10\]](#)
2. Tflint [\[11\]](#)

### 3. Trivy [12]

The Terraform CLI validation and fmt commands provide syntax validation and canonical formatting checks.

Tflint is a linting framework for Terraform that warns about deprecations, validates cloud vendor parameters such as instance types, and provides feedback around common best practices and naming conventions. The plugin architecture enables customization for the cloud provider we are using. Create a local file named .tflint.hcl with the following contents and run tflint --init to configure for AWS.

```
plugin "terraform" {
  enabled = true
  preset  = "all"
}

plugin "aws" {
  enabled = true
  version = "0.30.0"
  source  = "github.com/terraform-linters/tflint-ruleset-aws"
}
```

Trivy is a multipurpose, open-source security scanning tool created by Aquasec. It can scan various infrastructure technologies and frameworks, including Terraform files. The trivy config command can scan Terraform files for security issues.

Install these three tools and then complete the following exercise. The footnotes provide links to instructions for installing on your operating system.

## EXERCISE 6.3: PERFORM STATIC ANALYSIS OF TERRAFORM CODE

Create a file called main.tf with the following contents.

```

resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0"
  tags = {
    Name = "main"
    Pipeline = "https://github.com/my-org-name/my-repo-name"
  }
}

resource "aws_security_group_rule" "my_sg_rule" {
  type = "ingress"
  from_port = 0
  to_port = 65535
  protocol = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
  security_group_id = "sg-123456"
}

```

1. Using the Terraform CLI, validate the syntax and canonical formatting of main.tf
2. Correct the error and re-run the test
3. Using the Terraform CLI, apply canonical formatting to main.tf
4. Using Tflint, check the code style and formatting for best practices
5. Make the recommended changes until Tflint no longer returns warnings
6. Run a Trivy scan main.tf
7. Finally, correct the security issues in main.tf. For the MEDIUM alert, rather than adding a VPC Flow Log configuration, research the Trivy documentation for instructions on how to add an inline comment to ignore that particular security check.

Before we leave the topic, let's implement the local git commit-hooks recommendation from #G in Diagram 6-7. The Python package pre-commit [13] is an effective utility for managing local git commit hooks. Like the integration tests, the static analysis scans we examined above are steps that we want to include in our CI pipeline. Static tests can be applied locally against changed files with each commit for faster feedback rather than waiting for the pipeline to fail.

In the typical Terraform pipeline, an effective commit scan will include:

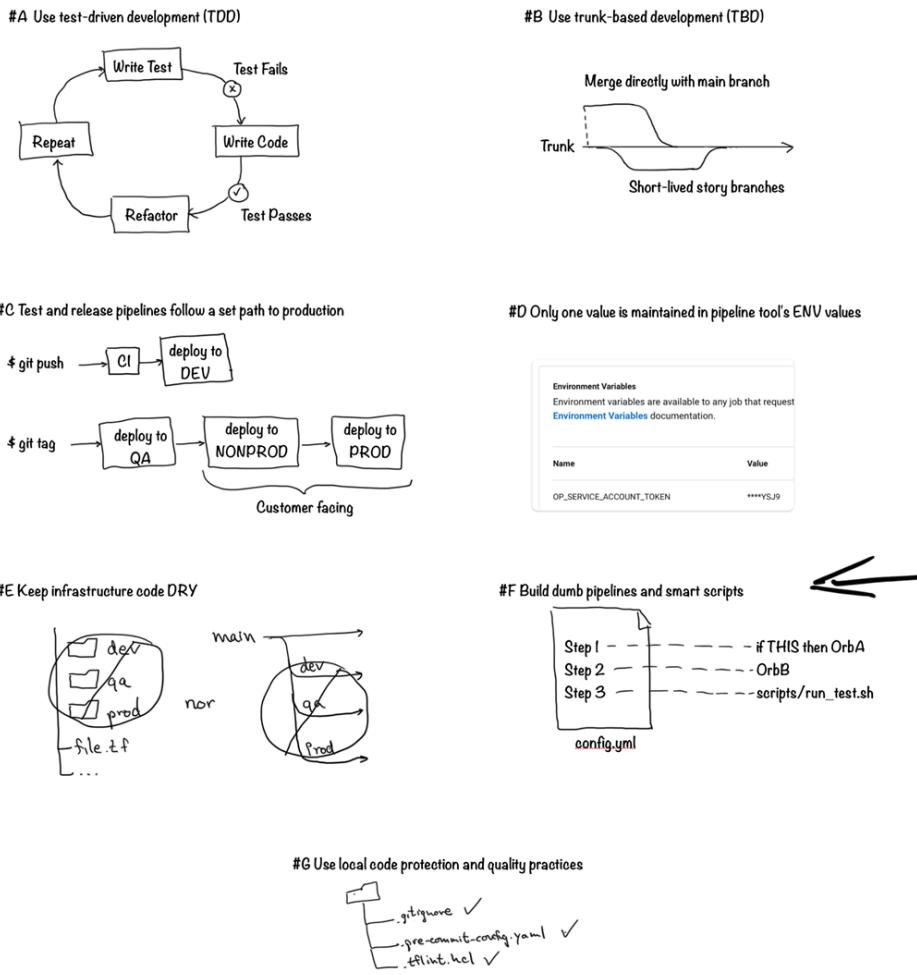
1. Static code analysis scans (same as will be applied in the pipeline) [14]
2. Scan for secrets inadvertently committed [15]
3. Basic git syntax and configuration standards [16]
4. Basic syntax scans for structured file types commonly found in Terraform repos

## **EXERCISE 6.4: IMPLEMENT STATIC ANALYSIS THROUGH COMMIT HOOKS**

Create a new git repository locally and add the uncorrected main.tf file from Exercise 6.3.

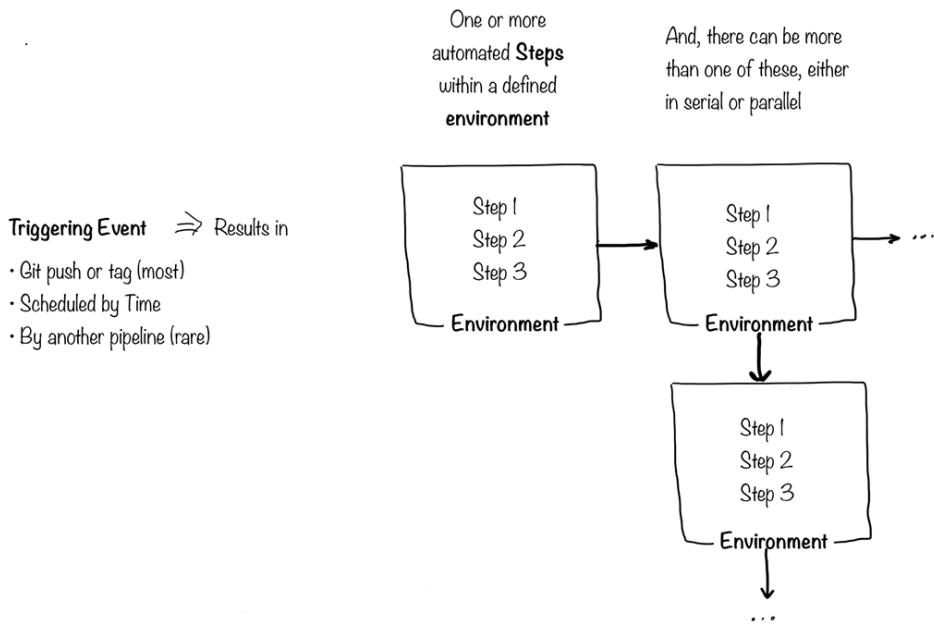
Using the Python package pre-commit, apply pre-commit hooks that perform each of the above four common scans for terraform pipelines.

### 6.3.3 Reusable Pipeline Code



**Figure 6.9 So far, in this section we've talked about seven essential infrastructure pipeline practices, discussed how to choose an IaC framework, and done a deeper dive into test-driven development of infrastructure code and performing static code analysis. Now, let's discuss reusable pipeline code.**

An effective, sustainable pipeline architecture is built around a strategy described as “dumb pipelines and smart code.” This is a play on words from the term “smart endpoints, dumb pipes” that has become well-known in the microservice development community.



**Figure 6.10** We intentionally want to constrain the complexity of our pipelines. **TRIGGERS:** A change to the source code in our repository should be the only reason the vast majority of our pipelines are triggered. Running a build or integration test on a routine schedule (nightly) can be a helpful Continuous Integration practice. Pipelines triggering other pipelines outside the same repo is nearly always a bad idea and probably indicates that code between those repos is being coupled in unhealthy ways. **STEPS:** In CircleCI, a Command is a series of Steps, and a Job is Steps with a specified Executor (the environment). In GitHub Actions, an Action is a series of steps, and a Workflow is Steps with the Runner definition.

In CircleCI, the pipeline defined in the config.yml file is *triggered* any time there is a change to the repo, including when a tag is applied. Each job listed in the workflows can also have a filter added to limit the job to a more specific change. The following filter will limit a job to run only when a new tag is added:

```
filters:
  branches:
    ignore: .*/
  tags:
    only: .*/
```

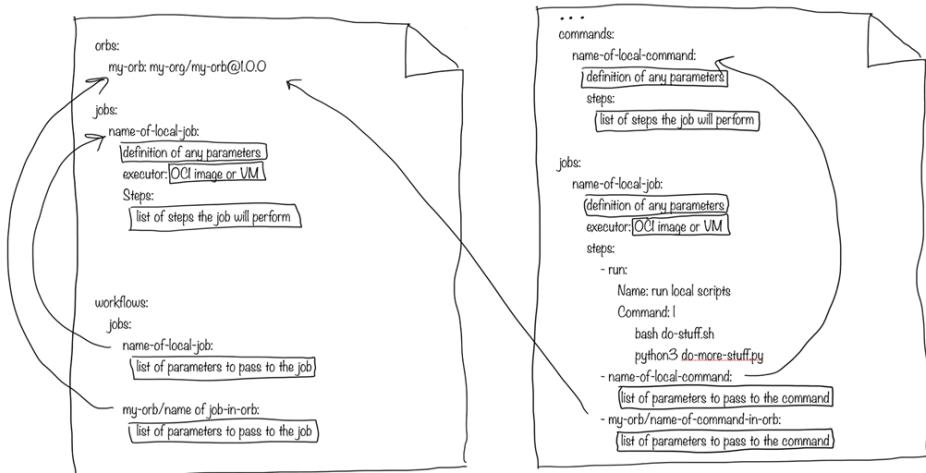
Jobs can also include a list of one or more other jobs that must be completed first. We can define a series of jobs that run in order upon a trigger or a list of jobs that will all run at the same time. Since many jobs in a pipeline can share the same trigger, we can use a Yaml anchor to avoid duplicating this code in multiple places. The following example is a filter that limits a job to run only when a change is pushed to the branch called main, ignoring all tags.

```

on-push-main: &on-push-main
  branches:
    only: /main/
  tags:
    ignore: ./*/
...
workflows:
  jobs:
    my-job:
      filters: *on-push-main

```

We want to keep our local pipelines as dumb as possible. Here, local means the pipeline code that lives in the repository with the code it manages. This pipeline should focus only on when to run (triggers) and the order of events. Wherever possible, specifics about “how” to perform a step should be defined in shared, versioned libraries of pipeline code or similarly shared language libraries such as bash or python. If the step details are unique to the pipeline, they should come from local scripts. Likewise, when the logic of a job or command is effectively unique to the pipeline, then local jobs and commands can be defined.



**Figure 6.11** In CircleCI, you list all the jobs you want the pipeline to run beneath the `workflows` key. You can refer to a job that has been defined right in the same config.yml file, or you can refer to a job defined in an Orb that will be pulled from the orb registry at pipeline runtime. Jobs contain a list of steps to perform. The individual steps can run command line instructions. But a step can also be the name of a Command, defined in the same pipeline or in an Orb. Commands are also a list of steps. Jobs and commands can have required and optional parameters.

Any pipeline may have some steps that are unique to it, but most of what an infrastructure pipeline does depends on the IaC framework, the cloud vendor where the infrastructure is set up, and the tools used for testing, scanning, fetching secrets, compliance, and so on. This is no different than an application pipeline. If you decide to build an API with Python, most of what your pipeline will do is determined by that choice and by the specifics of the deployment environment for your API.

Because so much of what the local pipeline does will be common to the underlying type of code and environment, building and maintaining shared pipeline code libraries (Orbs in CircleCI) will be a key part of what we do as the engineering platform product team. In practice, even with hundreds of teams using the platform, there will typically only be dozens of shared Orbs or Actions. These common libraries are also how we will manage the many non-functional (or cross-function) requirements related to work normally done in the pipeline. Also, by sticking to this practice, we can prevent implementing a different pipeline tool from becoming so costly that it is impractical. This is an important feature of the product. And even though the first pipeline libraries we create in building the VitalSigns platform are for our own team's benefit, most will be equally useful to our eventual customers.

With so much depending on these shared libraries, how they are managed matters. Pipelines are software. Assume that good software engineering practices will have as much value when applied to pipeline code as other types of software. When creating our *Orbs*, we must apply software development principles such as linting, testing, versioning, etc.

In addition, as you read in chapter 4 in the section on compliance, we want to keep the local pipeline code owned and managed by the developer and use shared libraries as the means by which we keep pipelines DRY.[\[17\]](#) across common workflows and provide developers the tools to meet non-functional requirements such as compliance. If we want developers to have the flexibility that comes from owning their own pipelines, how do we minimize the challenges that can come from using shared pipeline resources? If a shared job or command is too rigid or simplistic, developers will soon stop using it. Likewise, if pull-requests are the only means of customization, the frustration of waiting on approvals and the challenges of every change needing to support all users of the shared code will end in the same result.

Versioning can address the issue of how changes from the Orb owner can be released in a lower friction manner. Users have time to understand any changes and can adopt in a way that prevents them from being blocked. But this is not the only strategy.

Additional strategies include:

- Liberally include injection points throughout the job or command for custom steps.
- Provide overrides so the developer can customize the default parameters for tools or processes within the Orb. Validate the parameters for needed guardrails.
- Allow tool or package version overrides wherever possible. Validate the parameters for needed guardrails.

Here are examples of each of these.

### Injection points for custom steps:

A parameter to a CircleCI job or command can be defined as the type steps. Now, any valid list of steps can be passed as a parameter to the job.

Assume we are creating a shared job that performs the Plan phase of a Terraform pipeline. The steps of the job could start out looking something like this:

```
Steps:
- checkout
- setup_remote_docker
- run:
  name: terraform init
  working_directory: << parameters.working-directory >>
  command: |
    terraform version
    terraform init
- run:
  name: terraform plan
  working_directory: << parameters.working-directory >>
  command: |
    terraform plan -var-file=<< parameters.terraform-var-file >> --out tfplan.binary
    terraform show -json tfplan.binary | jq '.' > << parameters.terraform-plan-
outfile >>
- when:
  name: run checkov scan of terraform source files
  condition: << parameters.checkov-scan >>
Steps:
- checkov:
  working-directory: << parameters.working-directory >>
  checkov-additional-args: << parameters.checkov-additional-args >>
  terraform-plan-outfile: << parameters.terraform-plan-outfile >>
```

Let's add the following parameters to the job definition:

```

after-checkout:
  description: Optional steps to run after checking out the repository code.
  type: steps
  default: []
after-init:
  description: Optional steps to run after running terraform init.
  type: steps
  default: []
after-plan:
  description: Optional steps to run after running terraform plan.
  type: steps
  default: []
after-complete:
  description: Optional steps to run after the entire plan job is complete.
  type: steps
  default: []

```

Within the Orb job steps add the following:

```

Steps:
  - checkout
  - setup_remote_docker
  - when:
      name: Run after-checkout custom steps
      condition: << parameters.after-checkout >>
      working_directory: << parameters.working-directory >>
      steps: << parameters.after-checkout >>
  - run:
      name: terraform init
      ...

```

And do the same for each of the parameters at the respective location among the other job steps. Now, when using this shared job a developer can include whatever additional steps they want, at any of the potentially useful locations within the job. Suppose they need to populate the contents of their varfile at runtime because some of the values are the result of computations the pipeline will perform. With the after-checkout parameter, they can create a local command that performs all the steps and pass it to the plan job to be performed right after checkout. No change is needed to the shared job, nor do they have to build into the shared job a feature that only they need and account for any negative impacts to other users.

Keep this in mind when evaluating pipeline software. Generally, modern tools will have some means of passing or referencing custom pipeline steps from a shared job or command. For example, with Github Actions, when creating a shared workflow you can have steps that reference specifically named workflows from the calling repository.

### Overrides to tool parameters:

Look back at the terraform init and terraform plan steps in the above job. Let's change those steps:

```
- run:
  name: terraform init
  working_directory: << parameters.working-directory >>
  command: |
    terraform version
    terraform init << parameters.terraform-init-additional-args >>

- run:
  name: terraform plan
  working_directory: << parameters.working-directory >>
  command: |
    terraform plan \
      <<#parameters.terraform-var-file>> -var-file=<< parameters.terraform-var-file
>> <</parameters.terraform-var-file>> \
      <<#parameters.terraform-plan-additional-args>> << parameters.terraform-plan-
additional-args >> <</parameters.terraform-plan-additional-args>> \
      --out tfplan.binary
    terraform show -json tfplan.binary | jq '.' > << parameters.terraform-plan-
outfile >>
```

In this version, the developer can pass additional parameters to the init step. There are many reasons they might need to do this. What if someone made a manual change to the AWS resource through the console and the result was the need to use the --upgrade or --migrate-state flag?

Also, note that the new version of the terraform plan step doesn't assume that a varfile is being passed. It is an option. This allows us to take advantage of a very effective varfile template strategy. There are a number of situations where the ability to populate the contents of a varfile while the pipeline is running is necessary. What if we needed some sort of secret value? Or, what if parameter needed was a global value maintained in a remote location, such as a certificate. When we are pre-processing varfile contents, the version of the file maintained in the repo is now a kind of template. At pipeline runtime, we populate the dynamic values within the file and use the result as the varfile for the actual terraform actions. And in this situation we also want some assurance that this resulting file never appears accidentally in the repo. We can prevent git from including these files by adding the \*.tfvars extension to our .gitignore file. At runtime, we fetch the template varfile from a folder, populate it, and write it to the working directory as an .auto.tfvars file. Terraform will automatically include the file, and you do not need to manage the -varfile parameter in so there is less chance for a typo.

The new plan step also has an additional-args parameter so users can self-manage their own unique circumstances should they need it. If there are terraform plan arguments that specifically must not be used, such as overriding the backend state location, add validation logic as a step.

**Overrides to tool versions:**

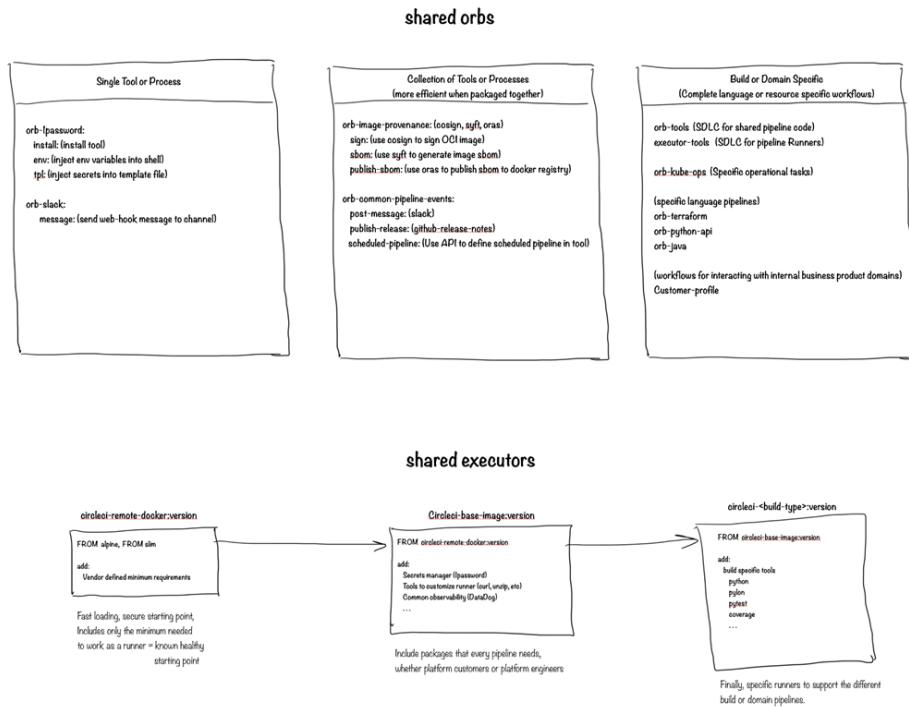
Another common point of friction is where Orb versions are either not providing minor and patch updates to tools in a timely manner, or a new Orb is released that has updated tools or packages, but is breaking for a small subset of users. In both situations, allowing developers to specify a different version, whether newer or older, is often the only adjustment needed for them to use the current version of the Orb.

In tracking the issues and impacts among teams in large organizations over several years from using shared pipeline libraries, we have found that the above three practices significantly reduced the majority of the long-term management issues. It isn't possible to remove 100% of the challenges from shared pipeline code; any more than you can remove 100% of the challenges from using regular programming language libraries. But these practices will make the experience dramatically better.

Users of the platform should also be encouraged to submit pull requests against Orbs as they find improved ways for the Orb to function or to increase its flexibility. But users of shared orbs should not need to make pull requests to keep from becoming blocked. The Orb maintainer will not always be able to respond in a timely manner or may reject the proposed change for legitimate reasons. Because the source code of the Orb is visible to Users, if at any point a platform user needs a change in the behavior of an Orb for whatever reason, they will not be blocked waiting for the owner of the Orb to make the change on their behalf as they can clone the Orb or include the Orb directly in-line within their pipeline and then modify as needed.

**STANDARD PIPELINES FOR MAINTAINING SHARED PIPELINE CODE**

Building and maintaining reusable pipeline code or runners is very much like any other kind of software development. CircleCI maintains an official orb designed to support the development lifecycle of shared orbs.[\[18\]](#)



**Figure 6.12 Specific tools or actions are the most basic type of shared pipeline code. But don't stop there. The most accelerating shared pipeline libraries are those that are optimized to cover the entire lifecycle for specific resources and language builds, including all the internal cross-functional requirements along with the language CI needs. The smaller, single-tool, or domain collections are valuable in assembling into the larger workflows and also providing platform users with supported and standardized ways of using platform resources within custom portions of their pipelines. Along with the shared libraries, we want to maintain the underlying pipeline runners, which are preconfigured with expected packages.**

The above orbs and executors are a good representation of the kinds of pipeline resources you will typically need to build and use as part of getting started building an engineering platform. The exercises in this book will use open-source orbs and executors. When you see later examples referencing these OSS resources, review the source code repositories to understand how to create an orb or executor build pipeline.

### 6.3.4 Private Executors (Runners)

Cloud vendors' API endpoints are exposed on the public internet. By following zero-trust networking engineering practices, every capability we include in our engineering platform, including how customers expose their APIs and UIs in each environment, can be effectively secured for similar public-facing access. This is the practice we recommend. But, it is also true that most organizations require all services and infrastructure to be exposed only within internal, private networks unless and until it must be made available to customers. Despite the value that can be demonstrated by removing the network from the security equation, we should still take some time to talk about private runners when using SaaS-based pipeline tools since this is the means of dealing with this challenge.

Private executors run on our cloud resources and are, therefore, a type of self-managed resource, unlike the executors hosted by CircleCI. We still want the same efficiency of use. This means we don't want executors that sit idle wasting resources, nor do we want pipelines waiting for executors to become available because they are a limited resource.

### EXERCISE 6.5: EXPERIMENT WITH SELF-HOSTED RUNNERS

Let's do a simple experiment with private runners using our pipeline tool, CircleCI.

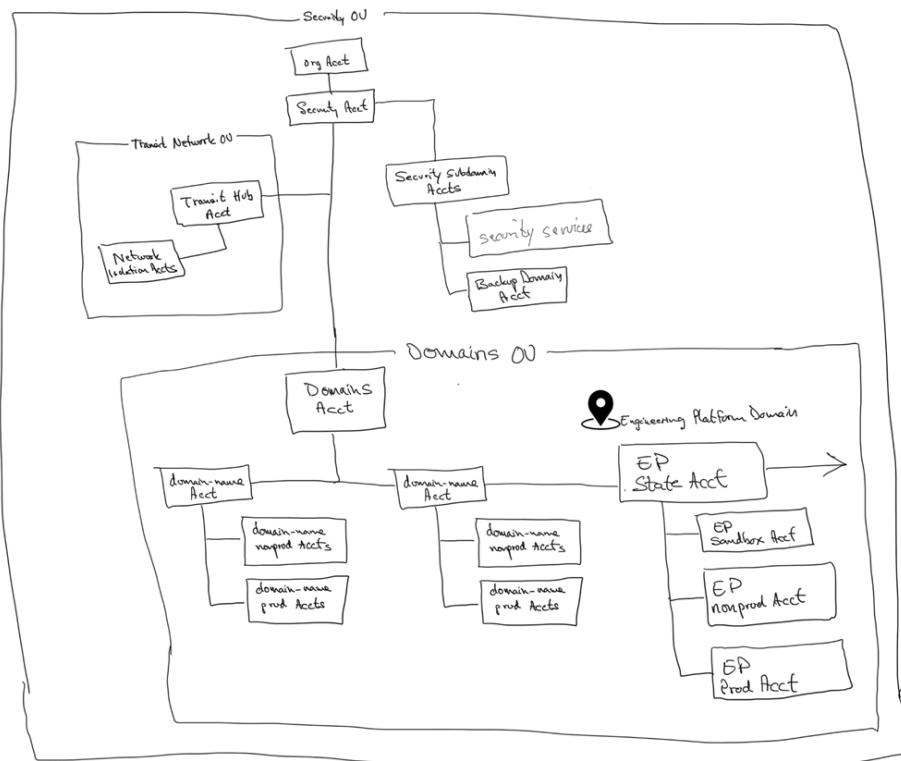
Start a local instance of Kubernetes on your laptop and use the container-agent service provided by CircleCI to create a private runner resource that will watch for CircleCI projects configured to use it and run the pipelines on containers started within the local cluster.

For this exercise, we will need the following tools: CircleCI cli.[\[19\]](#), Kind.[\[20\]](#), Kubectl.[\[21\]](#), Helm [\[22\]](#). These can be installed on MacOS using the homebrew package manager as follows. See the tool documentation for installation on other operating systems.

```
$ brew install kind kubectl helm circleci
```

## 6.4 Cloud Administrative Identity

The Cloud Administrative Identity domain encompasses the top-level cloud provider organization and account structure, along with the identities and permissions for the service accounts used in the foundational infrastructure automation.



**Figure 6.13 An Engineering Platform exists within your organization's cloud structure like any software product. This diagram shows a typical, well-formed AWS account structure with common OUs and clear product domain boundaries. The engineering platform should be assigned its own domain account, acting as a top-level account that can hold specific product-wide configurations, with sub-accounts for platform test environments and non-production and production contexts. These are cloud vendor-specific. In AWS, these would be Accounts. In Google Cloud, these would be projects.**

A well-formed cloud vendor organizational structure will reflect the organization's business and technology domain capabilities. A platform engineering team has four distinct needs at the start of product development.

## TOP-LEVEL DOMAIN ACCOUNT

First, the product will have a certain amount of product-wide configuration. One example is where engineering platform-specific governance policies are applied at the cloud vendor level. If there is a top-level account, then such policies can be applied in a single location with confidence that if additional accounts or sub-accounts are added in the future, the policies will consistently be applied. Another example is a configuration that is maintained on a product-wide basis. Service accounts, WAN policies, or, in AWS, settings maintained in Parameter Store are examples of where there can be a value or an object that must have a single location or instance yet also have product-wide (or cross-account) implications. A dedicated top-level domain or **state** account is an effective means of addressing this issue. It is not uncommon for many of the top-level items mentioned here to be managed by other teams in your organization. So long as it is recognized that the delivery team for an engineering platform is an administrative-level team that must have administrator-level permissions to manage the cloud accounts in which their product will be created, this can be manageable.

## PLATFORM PRODUCT TEST ENVIRONMENTS

An engineering platform will create environments used by the platform engineers delivering the product as development and test environments. The platform's internal customers never use these; they are solely for the development needs of the platform engineering team. It can be confusing to refer to this account as the *Dev* account since it is not where an internal platform customer has their *Dev* environments. We often refer to this account as the *Sandbox* account.

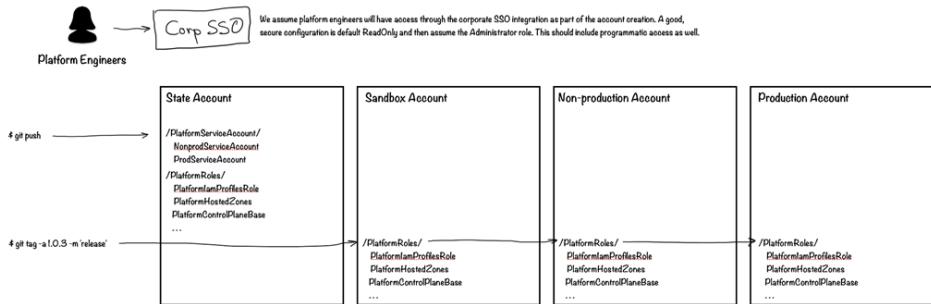
There will also be many account-level settings incorporated into an engineering platform, and isolating Platform test accounts from the internal customer-facing accounts is critical to enable the platform engineers to experiment, develop, and test without impacting platform users.

## INTERNAL CUSTOMER NON-PRODUCTION AND PRODUCTION ENVIRONMENTS

An engineering platform separates non-production developer environments from production environments for governance and production stability risk management. From a product perspective, all internal customer-facing environments should be considered *Production* by a platform engineer. This is no different from our expectations when using AWS, GC, or any cloud infrastructure provider. While we may deploy cloud infrastructure that we consider *development* in a cloud account, we do not expect the cloud provider to maintain or support that infrastructure any differently. The only thing we expect from AWS is their *production* context.

Finally, the basic structure of accounts described here is the starting point for an engineering platform product. Increasing scale and increasing complexity caused by regulatory or contractual issues can create the need for greater complexity than described here. A later chapter will describe several architectural strategies for managing such complexity.

### 6.4.1 Service Accounts and Permissions



**Figure 6.14** The first pipeline we create will manage the roles that our service accounts will assume to enable a pipeline to make the desired changes. Roles will be the same across all the engineering platform product accounts. In our example, we will also manage the IAM User service accounts. These exist only in a single AWS account.

In AWS, roles are used to define a set of permissions. Each of our infrastructure pipelines will need a matching role that grants the permissions necessary to do what the pipeline will attempt to do. Roles are assumed by the service account identity used in the pipeline. In other words, service accounts are defined only as having permission to assume roles.

Since we will be using the Kubernetes API as the general control plane of the engineering platform, a relatively limited number of terraform cloud infrastructure pipelines are needed. Many resources will be maintained via pipelines, but most will be services, controllers, and operators deployed to Kubernetes. We will use these operators and controllers to create a means for platform customers to create the infrastructure elements our platform supports rather than additional terraform workflows.

It is common to have less than a dozen pipelines using Terraform to configure cloud resources, even within a highly scaled-engineering platform. No matter how few, these pipelines nonetheless need dedicated roles that enable the pipeline to manage just the resources for which they are responsible.

In these VitalSigns exercises, we will assume that we need to manage the creation and management of service accounts and credentials. The simplified exercise example solutions assume we are using only two accounts.

As we start creating service accounts, it is worth noting that many SAAS pipeline tools now offer OIDC integrations with the major cloud vendors. Defining an application identity for a pipeline tool still generates an associated ID and secret. From the integration owners' point of view, you are still creating an identifying credential in either case. And frequently, one where the system doesn't offer an effective means of credential rotation. This also creates a completely new layer of automation complexity when applied broadly within the Platform for Developer use of the pipeline tool. Yet, the credential required to create this trust is used much less frequently. There are established, secure means of protecting automation service account credentials, and they are still the most broadly used industry standard. The option for system-to-system OIDC trust doesn't render the prior model insecure. Either one can be an appropriate choice.

An effective approach when launching an engineering platform with a single team is for the team to have just two service accounts used by their infrastructure pipelines. One for non-production use and one for production. The identities exist only in the State account. Use IAM Groups, one for nonproduction and one for production, to define in which accounts the service account may assume roles and add the service account to the appropriate group.

We are ready to get started with the first infrastructure pipeline for our VitalSigns engineering platform. However, we can't start using git-push to trigger the first change since a pipeline will require an identity and a role that does not yet exist.

First, we will need to bootstrap, from our workstation, the minimum configuration necessary to enable our roles and service accounts pipeline to start managing these resources directly. Once those resources exist in the cloud account, we can add a pipeline to apply and manage the resources we just created and those that will be added to this pipeline as we go.

The minimum resources we need to put in place are:

1. An IAM Role for the *roles and service accounts* pipeline that enables it to manage itself plus all the other needed pipeline roles
2. Two IAM Users to be the Nonproduction and Production service account identities
3. The initial credentials for our two service accounts

We will use the AWS-managed terraform module for IAM resources.[\[23\]](#) As we create these resource definitions using modules, apply the following practices:

- Always pin modules, providers, and tools to a specific version to avoid unexpected breakage.
- Use a dedicated role\_path for the roles created. It can greatly simplify ongoing management of pipeline roles when they are all kept in a single location.
- Place the IAM User on a dedicated path for the same reason.
- Name the backend state for each terraform workspace with a combination of the pipeline repository name and workspace name. This makes it much easier to interact with the state store in a predictable way programmatically.
- Adopt a consistent tagging strategy. This is an organizational need as much as an engineering platform requirement. We will want to have a tagging strategy that lets us track and allocate costs. At a minimum, there should be a product identifier and a reference to the pipeline that orchestrates this code.

- Use terraform variable validations wherever practical to help prevent errors in tfvar values.
- Keep terraform dry. Refrain from duplicating the code in folders or branches to deal with environmental differences. Differences (apart from the values passed in the tfvars file) between environments should be considered a bad idea that we will surely regret.

## EXERCISE 6.6: BOOTSTRAP OUR NONPRODUCTION AND PRODUCTION AWS ACCOUNTS WITH THE INITIAL SERVICE ACCOUNTS AND PIPELINE ROLE

Create a new repository called aws-iam-profiles. Create the following files and folder structure, and use the concepts we discussed above to add the file contents that can be used to bootstrap our accounts. In these exercises, we will only be using two AWS accounts.

```
aws-iam-profiles/
  environments/
    nonprod.auto.tfvars.json.tpl
    prod.auto.tfvars.json.tpl
  service-accounts.tf
  aws-iam-profiles-role.tf
  variables.tf
  versions.tf
```

Use your own credentials and administrator role permissions to apply the resources to the appropriate accounts with the matching tfvars.

Hint: Remember, we use Terraform Cloud for our backend state store. By default, a new workspace created in Terraform Cloud will be in *remote* execution mode; however, we will be operating in *local* execution mode. Local mode means we are not using the Terraform Cloud's additional paid features. Set our Terraform cloud organization default to local mode. The Terraform orb we are using can manage the setting on a per-workspace basis, or you can set it within the Terraform cloud UI if you prefer.

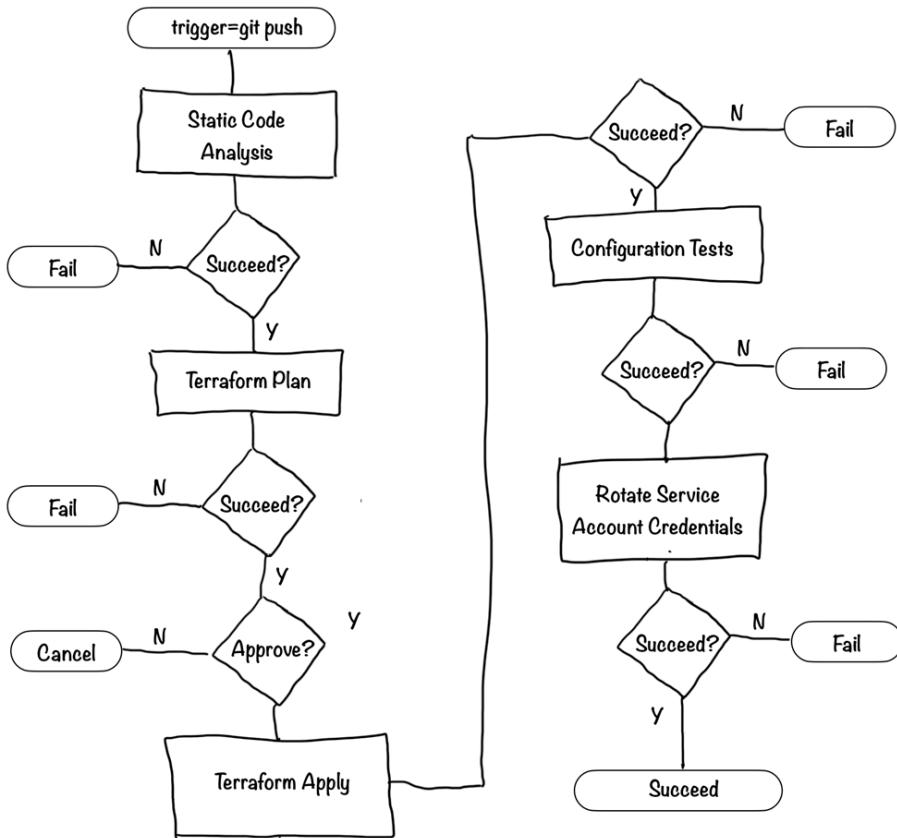
Also, since you are using your credentials and permissions as part of this bootstrap, comment out the `terraform assume_role` details in the `aws provider` definition. We can uncomment those lines once the service accounts are set up and credentials are available.

## BOOTSTRAPPING SERVICE ACCOUNT CREDENTIALS

After we have applied this configuration directly, the role that the pipeline in this repo needs exists in each account, and the service account identity the pipeline will use also exists in the correct account. Before implementing a pipeline for this repo that uses one of our new service account identities, we need to generate the associated access credentials and store these in our secrets management service. This pipeline will manage these credentials, routinely rotating the credentials in AWS and storing the current values in the secrets store. But the first time our pipeline runs, it will try and fetch the credentials from the secrets store, so we will need to manually create and store those credentials the first time for the pipeline to succeed and thereby take over the automated management of the service account credentials.

Use the AWS Console or CLI to generate credentials for each service account and store these in our secrets store.

With that done, we can start on the first stage of our `platform_iam_roles` pipeline.



**Figure 6.15** Let's walk through the logic of the first stage in our pipeline. We want to take these steps whenever a change is pushed to this repository. This is a familiar Terraform flow. Terraform will generate a plan showing expected changes after the static code analysis, including a security scan. If it looks good, we want to apply the changes to our state account and compare the resulting actual configuration against our desired state. Finally, since we also store our service accounts in the state account, we will take this opportunity to perform a resilient credential rotation.

Create the file config.yml inside the .circleci folder and add the following snippet before starting the pipeline exercise.

```

---
version: 2.1
orbs:
  terraform: twdps/terraform@3.0.1          #A
  op: twdps/onepassword@3.0.0
  do: twdps/pipeline-events@4.1.0
globals:
  - &context <my-team>
  - &executor-image twdps/circleci-infra-aws:alpine-7.5.0
on-push-main: &on-push-main                  #C
branches:
  only: /main/
tags:
  ignore: /*/
commands:
workflows:

#A Use these three orbs in creating our pipeline. The terraform orb[24] includes workflow jobs for static analysis, plan, and apply. The 1Password orb[25] provides commands for fetching secrets from our 1Password vault, populating environment variables, or creating credential files. The pipeline-events orb[26] contains a collection of common commands. In particular, consider using the bash-functions command to make available local scripts for assuming roles in AWS and writing values into a 1password vault.
#B Use a YAML anchor to define a couple of global values. In this case, the CircleCI Context that contains our team's 1Password service account credential and the runner image we want to use for the jobs. To create a context in CircleCI, select contexts from the organizational settings. Create a new context named to match our team name in GitHub. Add a single environment variable called OP_SERVICE_ACCOUNT_TOKEN and place our 1Password service account token in the value. This is the only value we will store within CircleCI. This is a secure location, but managing values through this feature is time-consuming and error-prone. All other values we need in the pipeline we will fetch at pipeline runtime. The OSS runner contains all the packages and dependencies for our terraform pipeline[27] and is part of a series of examples of maintaining pre-configured runners. The source code for these Orbs and Executors is open source, and you can review it to see exactly what the job or command does. The YAML key globals is not actually used by CircleCI pipelines, but since it conforms to standard YAML, you can include additional data even though the pipeline processor itself will not do anything with it.
#C Define an anchor to describe the trigger for the first workflows. In this case, when a change is pushed to the main branch only and ignore tags.

```

## EXERCISE: CREATE THE CI AND DEVELOPMENT TEST PIPELINE FOR IAM SERVICE ACCOUNTS AND ROLES

Now we are ready to complete the rest of the first stage of our pipeline according to the workflow diagram above and test it against our nonprod account. Complete the above starting setup by adding the workflows and commands needed into sections “commands” and “workflows”. Wait until the next exercise to work on the credential rotation step. Don’t forget to uncomment the lines from `versions.tf` that cause Terraform to attempt to assume the needed IAM Role.

When using the static-code-analysis, plan, or apply jobs from the Terraform orb, remember that you can pass commands as a parameter and they will be run at the place you specify in the job. If we discover the need to set up some credentials or inject values into a tfvars template, we can easily do that without changing the orb directly.

When you are ready to start triggering the pipeline, go to our CircleCI organization, find the aws-iam-profiles repo among the projects, and start building. When you click set up project you will be asked whether you want to add a config.yml or use one already in the repo. Choose the option to use the pipeline you have created on the main branch. This step of setting up a project in CircleCI only happens when first connecting CircleCI to a repository.

You can see in this pipeline the impact of using versioned pipeline code steps or jobs. In this case, the terraform orb we used provides most of the functionality we need for a typical terraform pipeline, including hooks that let us provide custom commands at various points in the workflow. The orb provides pre-configured steps for calling the static analysis tools we have decided to use. In the future, should the technical details for interacting with the tool change or even a change in the tool itself, we can release a new version of the orb for users to adopt and afford a window of time for users to adopt before deprecating the earlier pattern.

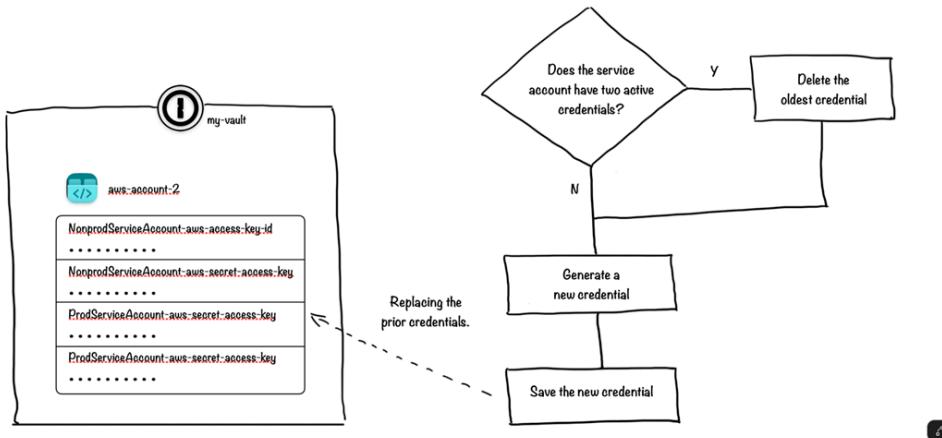
## AUTOMATED ROTATION OF SERVICE ACCOUNT CREDENTIALS

The last step in the first stage of our pipeline is to rotate the credentials of the service accounts. It isn't difficult to call the AWS api to generate new credentials for an IAM user, but if we generate new credentials and update the values in the secrets store while there are pipelines or other service account workloads using those credentials at that moment, those jobs will break. We need a way to ensure that a key being changed during a routine rotation is still usable in automation, at least to the extent necessary to prevent this race condition.

A common practice is using a two-key pattern and performing a rotation at least twice within the desired frequency. What does this mean in practice?

IAM > Users > NonprodServiceAccount

SUMMARY		
ARN arn:aws:iam::123456789123:user/ PlatformServiceAccounts/ NonprodServiceAccount	Console access Disabled	Access key 1 AKIAR***** - Active Used 6 days ago. 13 days old.
Created April 26, 2023, 21:40 (UTC-05:00)	Last console sign-in -	Access key 2 AKIAR***** - Active Used 3 hours ago. 6 days old.



**Figure 6.16** A service account has two valid credentials at any given time. Only the latest credential is maintained in the secrets store, and the prior keys remain valid until the next rotation to prevent workloads and orchestration from failing due to the race condition of the key being changed while a job is already in process. New jobs or pipelines launching always fetch the credentials from the secrets store and are therefore using the latest credentials. If you define an automated rotation to occur every seven days, the active key is never more than seven days old, and both keys will be rotated every 14 days.

Support for this automated identity flow is one of the reasons IAM Users can have two keys. This could be done using the AWS CLI. But there are a couple of steps, and if we use the AWS CLI we will need to put these steps into a script. We only need to do this in this one pipeline, so that is not necessarily a problem. We could also create a shareable utility that can be used in any pipeline. Let's use the Python package `iam-credential-rotation`.[\[28\]](#).

The default method for the tool requires a single parameter: the IAM path in which the service accounts can be found. Recall we used the path `/PlatformServiceAccounts/` for the two service accounts we defined. Assuming you have appropriate credentials, the following command will apply the workflow from Diagram 6.15 for every IAM user found on the path we pass as the parameters. The resulting new credentials will be written to stdout.

```
$ iam-credential-rotation PlatformServiceAccounts
{
  "NonprodServiceAccount": {
    "AccessKeyId": "AKIARKL*****",
    "SecretAccessKey": "bCFqIBZUo*****"
  },
  "ProdServiceAccount": {
    "AccessKeyId": "AKIARKLI*****",
    "SecretAccessKey": "cVSkOhunYxS*****"
  }
}
```

## EXERCISE: ADD A CREDENTIAL ROTATION STEP TO THE FIRST STAGE PIPELINE

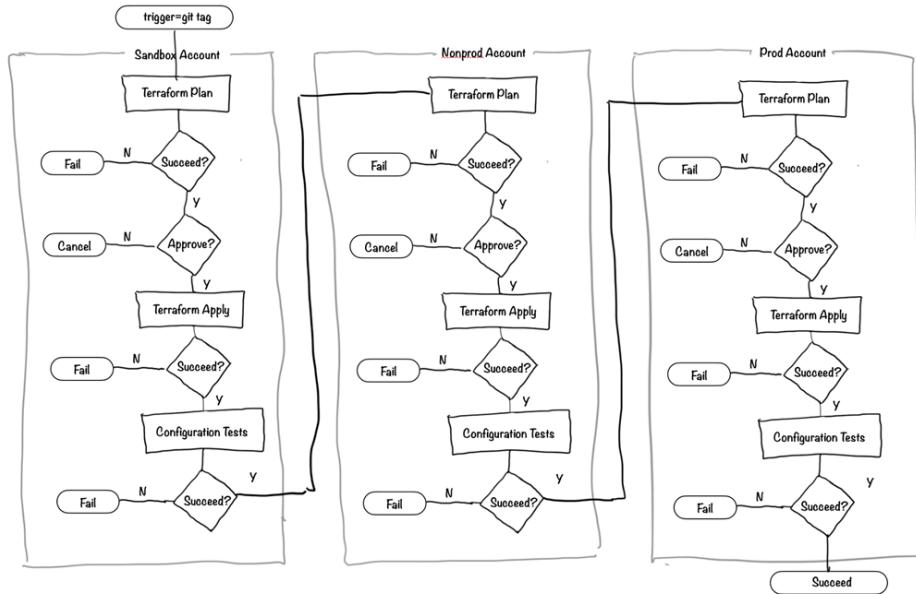
Add a credential rotation step to the after-apply parameters of our terraform apply pipeline step. This will come after the AWS integration tests.

```
After-apply:
- aws-integration-tests:
  account: nonprod
- rotate-service-account-credentials:
  account: nonprod
```

Now, define a local command that will use the `iam-credential-rotation` utility to rotate our service account credentials and store the resulting new credentials in our 1Password vault, overwriting the existing service account credentials.

## RELEASE OUR CHANGES TO PRODUCTION

With every step in the first stage of our pipeline running successfully, we need a way to release this change to production. We trigger our release pipeline with a semantic tag to our latest commit.



**Figure 6.17** In the four-account strategy described at the start of this chapter, our service accounts are deployed to the STATE account only, and it is also in this account where we perform the initial configuration test of the Roles. Recall that our pipeline roles are deployed into each account. They are identical in each account. These roles are only assumed by our platform's dedicated infrastructure service accounts. And only the ProdServiceAccount is able to assume roles in the Prod account. Our release workflow would look like this, with three approval steps where an engineer can validate the Terraform plan results and decide to let the changes be applied to the next account. Any failure along the way will cause the entire release workflow to fail. That is the outcome we want. If we discover any environmental difference, the solution will be a code change, and all changes must go through this path to production to ensure that every instance of the product is built from the same code.

## EXERCISE: CREATE THE RELEASE PIPELINE FOR IAM SERVICE ACCOUNTS AND ROLES

We need a trigger that will watch for new tags.

```

on-tag-main: &on-tag-main
Branches:
  ignore: .*/
Tags:
  only: .*/
  
```

Now add a workflow for the release job in the pipeline. We only need to deploy to a single account since we only use two accounts instead of four in our simplified example. This means we only need to add the first column from diagram 6.17.

## ADDITIONAL PIPELINE STEPS

What if we were to add some form of notification to our pipeline? We could add a command to send a message to a team chat channel. In practice, such messages can have value but have the same limitation as a monitoring alert. The message must be actionable, or it will become ignored. Sending a Fail message when a pipeline fails can be actionable. But then our pipeline tool will tell us when a pipeline fails, and most often, a pipeline is triggered because we made a change and will be watching for the pipeline results. And there are less complex ways of adding an additional pipeline status indicator than making this the general responsibility of the underlying pipeline code [\[29\]](#). Yet, there can be situations where a specific notification will be actionable and is worth adding to a pipeline.

The pipeline-events orb we are using includes a command for sending messages to Slack. We could notify our team of production changes just by adding these lines to the after-apply parameters:

```
- do/slack-bot:
  channel: engineering platform events
  message: New release of aws-iam-profiles
  include-link: true
  include-tag: true
```

How do we notify customers about new feature releases, incidents or outages, and events in general? A good experience will involve multiple channels but subscribing to a chat channel is something most customers prefer.

A standard practice for a release pipeline is to generate release notes. Anyone impacted by the changes can then read details about precisely what has changed. The pipeline-events orb includes a job that uses the github-release-notes utility (gren) to automatically generate a release and compile the completed issues and commit messages since the last release. Let's include this job as part of the release workflow.

```

- do/release:
  name: generate release notes
  context: *context
  on-tag: true #A
  additional-args: "--data-source=commits" #B
  Before-release:
    - op/env:
      env-file: op.prod.env #C
  Requires:
    - apply prod changes
  filters: *on-tag-main

```

#A Release notes will include issues and commit messages since the last release.

#B Include all commit messages in release notes.

#C Add our GitHub token to the op.prod.env.

#D The release note job should be contingent upon a successful release.

```
export GREP_GITHUB_TOKEN={{ op://my-vault/svc-github/access-token }}
```

When using git tokens, we would definitely prefer to be able to create and use a GitHub service account to generate the token so that it is not tied to a specific user, at least in theory. However, this is not as straightforward as it used to be since GitHub requires all users to have MFA.

The final step we will add before leaving this section is a regularly scheduled run of our integration tests. Recall from the section on TDD of infrastructure code that a recurring run of these tests can help catch unexpected changes made outside of our infrastructure code.

We should add a workflow that schedules a job each week to run our integration tests *and* rotate the service account credentials. The pipeline-events orb has a command that makes setting these schedules in CircleCI easy.

```
workflows:  
  . . .  
  
    schedule weekly integration tests and sa credential rotation:  
      jobs:  
        - do/schedule-pipeline:  
            name: weekly integration test and sa credential rotation  
            context: *context  
            scheduled-pipeline-name: weekly-iam-profiles-jobs  
            scheduled-pipeline-description: |  
              Weekly, automated run of integration tests  
              and iam-credential-rotation  
            hours-of-day: "[1]" #A  
            days-of-week: "[\"SUN\"]"  
            Before-schedule:  
              - op/env:  
                  env-file: op.prod.env  
            filters: *on-tag-main
```

#A Job runs every Sunday at 1:00am.

#B When the ProdServiceAccount can access both Nonproduction and Production resources, then we need only schedule a single recurring job. If we did not permit the Prod sa to access Nonprod resources then we would have to schedule two jobs.

Add the definition for this scheduled job.

```

weekly iam profiles jobs:                                #A
  When:
    equal: [ scheduled_pipeline, << pipeline.trigger_source >> ] #B
  Jobs:
    - recurring-integration-tests:                                #C
      name: AWS integration test on Nonprod account
      context: *context
      account: nonprod
    - recurring-integration-tests:                                #C
      name: AWS integration test on Prod account
      context: *context
      account: prod
    - rotate-credentials:                                       #D
      name: Rotate service account credentials
      context: *context

```

#A Though we are using the same name as the scheduled-pipeline-name parameter we passed when scheduling a trigger, the pipeline name is just a label displayed in the UI when the trigger occurs.

#B A triggered pipeline is effectively the same as either a git-push or a git-tag in terms of how our filters will respond and by default, is treated like git-push. In other words, when this pipeline is triggered because of a schedule, it will behave as though it were caused by git-push. This means that when we start using the scheduled pipeline feature, then individual workflows that should run only when scheduled or only run when actually caused by a code change need some additional configuration. The configuration you see here is saying 'only run when the trigger source is actually because of a scheduled\_pipeline.'

#C Run the integration tests on all the accounts configured by the pipeline.

#D Rotate the service account credentials. Since the pipeline runs once a week, this means the credentials are fully rotated every two weeks.

We just defined two jobs that our workflow will trigger each week. But we haven't added these local jobs to our pipeline. In the jobs section, add these two jobs.

```

jobs:
  recurring-integration-tests:
    description: |
      Recurring job (weekly) to run pipeline integration tests to detect aws
      configuration drift
    docker: #A
      - image: *executor-image
    parameters:
      account:
        description: nonprod or production account configuration
        type: string
    steps:
      - checkout
      - setup_remote_docker
      - set-environment: #B
        account: << parameters.account >>
      - aws-integration-tests: #C
        account: << parameters.account >>

rotate-credentials: #D
  description: Recurring job (weekly) to rotate PSK service account credentials
  Docker:
    - image: *executor-image
  steps:
    - checkout
    - setup_remote_docker
    - set-environment:
      account: nonprod
    - rotate-service-account-credentials:
      account: nonprod

```

#A This is a job not just a command, so we need to define the executor to be used by the job. We can use the same executor as the Terraform job. The only jobs we have used up to this point have been defined in the Terraform orb. When you look at the source code for this orb you will see that the job definition includes an executor configuration.

#B Just like in our Terraform apply job, we need to set up our credentials

#C We use the same aws-integration-test command, performing the same tests, as we did in the terraform apply job.

#D This job is similar to the integration-test job. We are calling the same rotate-service-account-credentials command that we did in the first stage of our pipeline. We just need to add the additional configuration that a job requires.

And finally, because our scheduled pipeline is treated like a git-push trigger, we need to add the configuration to our first stage so that it only runs on an actual change and not every time our scheduled pipeline runs.

```
deploy service accounts and roles to state account:
When:
Not:
  equal: [ scheduled_pipeline, << pipeline.trigger_source >> ]
Jobs:
  . . .
```

Sending a Slack message and scheduling a CircleCI pipeline also require two more credentials to be added to our op.prod.env file.

```
export SLACK_BOT_TOKEN=op://my-vault/svc-slack/post-bot-token
export CIRCLE_TOKEN=op://my-vault/svc-circleci/api-token
```

With all the pieces in place, we have a fully software-defined lifecycle for our services accounts and needed pipeline roles.

With each new platform infrastructure pipeline we create, we will start by adding a new role in this aws-iam-profiles repository to provide the permissions the new pipeline needs.

## 6.5 Summary

- Start with a simple system that evolves into a complex system.
- Traditional IT silos and DevOps models have limitations in achieving platform goals.
- Advocate for a unified team approach in a greenfield setting to manage domain boundaries effectively.
- Effective product ownership and platform engineering practices are essential for long-term success.
- Tools must align with effective platform engineering practices and customer needs.
- High-quality, secure SaaS tools are preferred for long-term efficiency.
- Tools should be easy to integrate, support pipeline use, and provide a single source of truth for secure values.
- Encourage developer ownership of pipeline code to minimize manual intervention.
- Optimize pipeline performance and provide secure debugging access for developers.
- Define platform engineering practices for well-architected infrastructure pipelines.
- Focus on continuous deployment automation, secrets management, and avoiding duplicated code.

- Adopt declarative Infrastructure-as-Code (IaC) frameworks like Terraform for cloud-native architectures.
- Ensure pipelines are versioned, shared, and optimized for maintainability and scalability.
- Infrastructure is highly declarative, with actions happening inside vendor APIs.
- Integration tests for infrastructure code are valuable despite not directly writing the action code.
- Integration tests are crucial for confirming reusable modules and providing the intended resources.
- Tests can demonstrate story acceptance criteria, like naming conventions or specific configurations.
- Recurring tests help detect configuration drift caused by manual changes outside the pipeline.
- Writing tests first clarifies desired outcomes and reduces the risk of flawed test logic.
- Use tools like Awspec and InSpec for infrastructure configuration testing.
- Static code analysis is vital for maintaining IaC code quality, focusing on syntax, style, and security.
- Tools like Tflint and Trivy validate best practices and identify security issues in Terraform code.
- Shared pipeline code libraries (e.g., CircleCI Orbs) should allow flexibility for developers.
- Pipelines should focus on triggers and event order, with specifics handled by shared code.
- Injecting custom steps, overriding parameters, and allowing tool version overrides improve pipeline flexibility.
- Managing shared pipeline resources effectively reduces long-term challenges in large organizations.
- Users should be encouraged to submit pull requests but can also clone and modify Orbs as needed.
- Building and maintaining reusable pipeline code requires applying software development principles.
- Private executors (runners) handle pipelines in internal, private networks, ensuring security and efficiency.
- Manage cloud organization structure, identities, and permissions for service accounts in foundational infrastructure automation.
- A well-organized cloud structure should align with business and technology needs.
- A top-level domain account is needed for product-wide configuration and governance policies.
- Platform engineering teams need dedicated test environments, often called Sandbox accounts, separate from customer-facing environments.
- Separate non-production and production environments are crucial for governance and stability.

- Service accounts and roles in AWS or similar providers require specific permissions defined by Roles, which are assumed by pipeline service accounts.
- Infrastructure pipelines need dedicated roles for managing specific resources, even if only a few exist.
- Initial service account setup involves bootstrapping with minimum configuration before pipelines can manage resources autonomously.
- Use Terraform with best practices like version pinning, role path organization, and consistent tagging.

- [1] John Gall (1975) Systemantics: How Systems Really Work and How They Fail. p. 71
- [2] <https://github.com/orgs/effective-platform-engineering/repositories>
- [3] [https://github.com/effective-platform-engineering/companion-code/blob/main/chapter-6/configuring\\_github\\_for\\_signed\\_commits.md](https://github.com/effective-platform-engineering/companion-code/blob/main/chapter-6/configuring_github_for_signed_commits.md)
- [4] <https://developer.1password.com/docs/cli/reference/management-commands/service-account/>
- [5] Alternative SaaS options for secrets management and Terraform state that have a free tier you could explore are doppler.com and tfstate.dev
- [6] <https://github.com/tellerops/teller>
- [7] <https://github.com/k1LoW/awspec>
- [8] <https://rspec.info>
- [9] <https://docs.chef.io/inspec/>
- [10] <https://developer.hashicorp.com/terraform/install>
- [11] <https://github.com/terraform-linters/tflint>
- [12] <https://github.com/aquasecurity/trivy>
- [13] <https://pre-commit.com/>
- [14] <https://github.com/antonbabenko/pre-commit-terraform>
- [15] <https://github.com/awslabs/git-secrets>
- [16] <https://github.com/pre-commit/pre-commit-hooks>
- [17] **Don't Repeat Yourself.** In most cases, duplicate code, even pipeline code, is difficult to maintain over time. When changes are needed it is difficult to make sure they happen in every duplicated location.
- [18] <https://circleci.com/developer/orbs/orb/circleci/orb-tools>
- [19] <https://circleci.com/docs/local-cli/>

- [20] <https://kind.sigs.k8s.io/docs/user/quick-start/#installation>
- [21] <https://kubernetes.io/docs/tasks/tools/>
- [22] <https://helm.sh/docs/intro/install/>
- [23] <https://registry.terraform.io/modules/terraform-aws-modules/iam/aws/5.40.0>
- [24] <https://github.com/ThoughtWorks-DPS/orb-terraform>
- [25] <https://github.com/ThoughtWorks-DPS/orb-1password-connect>
- [26] <https://github.com/ThoughtWorks-DPS/orb-pipeline-events>
- [27] <https://github.com/ThoughtWorks-DPS/circleci-infra-aws>
- [28] <https://pypi.org/project/iam-credential-rotation/>
- [29] <https://ccmenu.org>, <https://github.com/build-canaries/nevergreen>

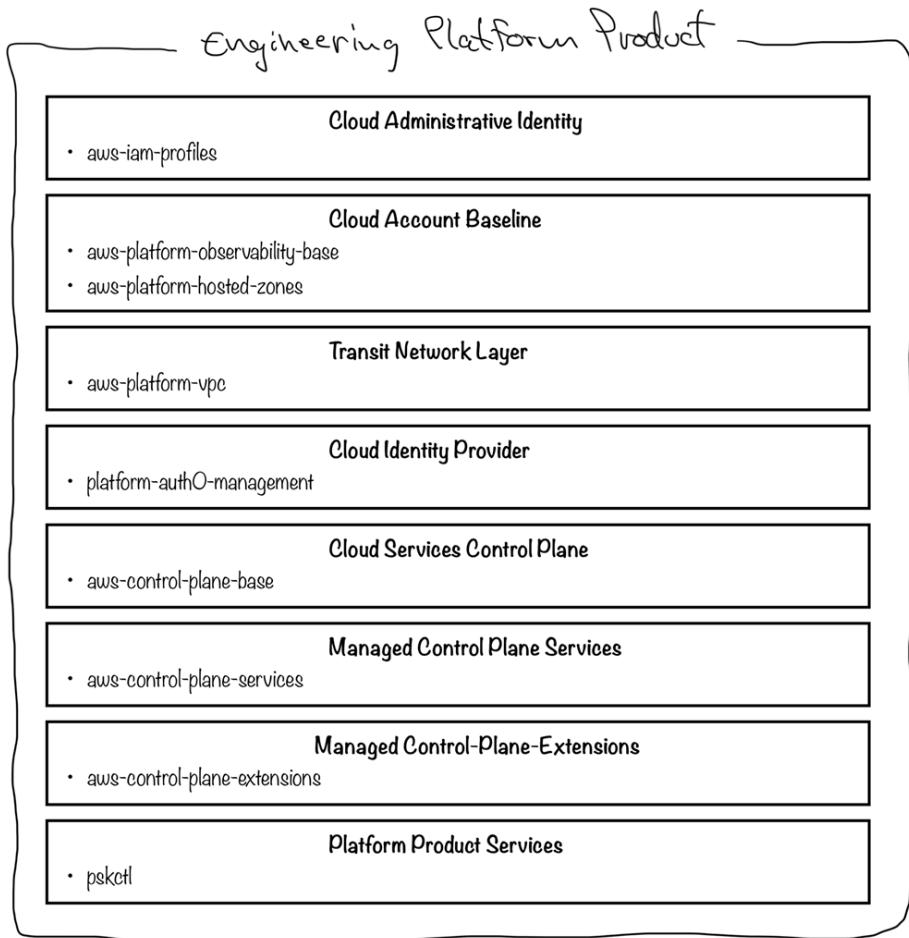
# ***7 Platform Control Plane Foundations***

## **This chapter covers**

- Managing Cloud Account Baseline Settings
- Defining the Transit Network Layer
- Separating Customer Identity
- Deploying the Cloud Service Control Plane

In chapter 6, we started building the VitalSigns engineering platform, gathering the prerequisite resources and bootstrapping the initial pipeline. We began with the same developer tools we will provide our internal customers, like source control, secrets management, and pipeline orchestration. Now, we can proceed to the foundational components of the platform. The overall product goal for our engineering platform is to provide VitalSigns developers access to all the resources they need to build, release, and operate software independently, without the usual engineering friction. Every component of our platform needs to be resilient. This goes beyond merely including redundancy to minimize the impact of failure. We want our services and the applications that our customers build and deploy to have a level of self-healing. If the actual state of something in our platform is not as we expect - is not in the desired state - we want the platform to be able to correct this. Kubernetes provides us with this capability. When talking about Kubernetes, most people are familiar with its ability to redeploy a failing service, move services off of failing virtual machines, or scale up or down the number of instances of a service based on the amount of traffic. Yet, the Kubernetes API also allows us to tap into this orchestration logic and apply it in other ways to extend the API. This is why we refer to Kubernetes as the Control Plane. It is a key component among those that form the foundation of our engineering platform.

## 7.1 Cloud Account Baseline



**Figure 7.1** In Chapter 1, we discussed the importance of domain-driven design and the platform product domains. The aws-iam-profiles pipeline we created was part of the Cloud Administrative Identity product domain. We now continue to the Cloud Account Baseline domain and create the account-level baseline resources.

This domain might feel similar to the last one as it also deals with cloud account-level settings. Just like with Roles, these configurations are set up once per account. But in reality, AWS account identity, authentication, and authorization have such a broad impact on the platform engineering team's ability to work effectively it often becomes one of the first areas to need its own dedicated domain team. Because of this, administrative identity becomes its own domain. The rest of the account-level resources can be treated as a domain as well, and this is what we've named the Cloud Account Baseline domain. We will discuss three of the primary areas.

### 7.1.1 Account Baseline Security Scanning

Account-level baseline security configuration can be effectively managed by a separate security team so long as the security leaders recognize the platform accounts as infrastructure-provider accounts and the platform engineers building the platform as the natural administrative owners of those cloud accounts. In this case, the baseline security configuration should be based on the configuration to which all cloud accounts within the enterprise must conform.

Examples include:

- Permission that limit VPCs to connect only to the enterprise transit-gateway network
- Permission boundaries that require Public networks only to contain load balancers integrated into a distributed configuration management architecture such as AWS Firewall Manager.
- Permission boundaries that restrict access to cloud services that are not allowed based on enterprise technology standards
- Resources to automatically ingest all account-level logs into a centralized log aggregator
- Resources to automatically ingest specific logs that the security team is interested in to a centralized location

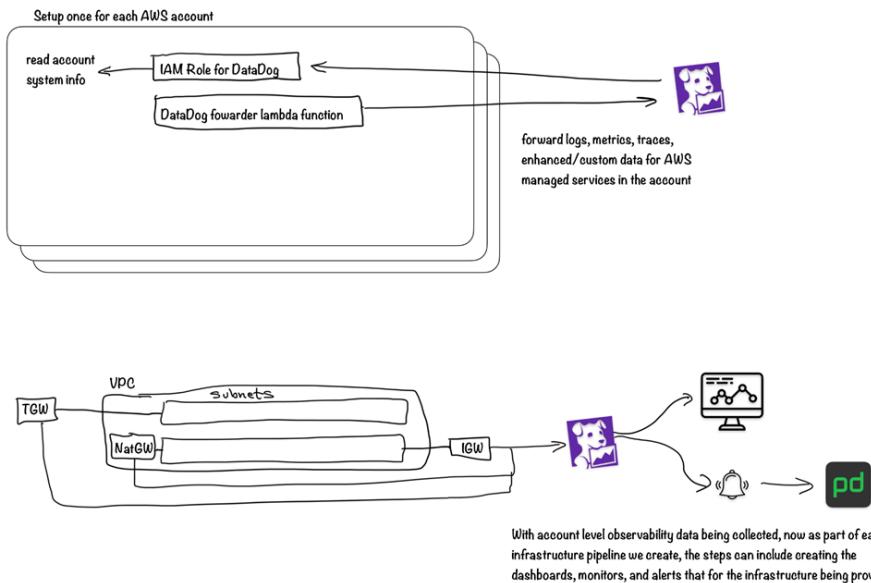
The recommended practice is that these kinds of baseline configurations be managed through a recurring, idempotent job that runs frequently (perhaps nightly) along with a similar, recurring scan designed to catch changes before they are automatically corrected.

It is also not uncommon for an enterprise to not adequately staff or fund its security teams to own this kind of configuration; instead, it requires that each infrastructure administrative team implement the security requirements as part of its responsibilities. If that is true in our organization, then as the owners of the engineering platform, we would include this configuration in our Cloud Account Baseline domain and create the software-defined configuration with the same architectural goals we apply to everything we maintain.

We won't implement this configuration as part of our VitalSigns example. For an example of the configuration and scans commonly occurring in this domain, look at the CIS AWS Foundations Benchmark and the Mitre example InSpec scan[\[1\]](#).

### 7.1.2 Account Baseline Observability

In Chapter 5 we cover key concepts of observability. If VitalSigns were to adopt the same strategy as PETech in terms of technologies, at this point in our implementation of the VitalSigns engineering platform we would be at the initial implementation stages for the Platform and Cloud Cost axes of observability. Within those there are cloud account-level components needed to begin aggregating system data that is generated by AWS account level services.



**Figure 7.2** If, like PETech in Chapter 5, VitalSigns were using DataDog it is at this point we would set up a repository and pipeline to manage the account-level integration provided by DataDog for AWS. With that integration in place, with each additional capability or feature we implement, like the networks we will provision in section 7.2, observability would be a part of the natural *definition-of-done* for the implementation.

Creating and improving observability, both for operations and ongoing product development, is a part of every thing we deploy. For cost and scope reasons, while we build out the vitalsigns platform we won't include complete implementation of observability as part of the exercises or solutions. Use the principles from Chapter 5 to think about what those solutions should look like. When we get to deploying applications onto the control plane we will implement some basic cluster-level capabilities in order to demonstrate some key parts of what the Platform customers (developer) experience must be.

## WHAT ABOUT THE CLOUD INFRASTRUCTURE VENDORS BUILT-IN SOLUTIONS?

Since we used DataDog in this example, a good question at this point would be what about using the built-in AWS solutions like cloudwatch or x-ray? Every infrastructure-as-a-service provider has some form of observability service. And of course we should take those services into consideration when deciding on the experience our engineering platform will provide.

For the most part, the selection criteria can be thought of much the same way as the criteria we discussed in Chapter 6. The one critical factor that is frequently overlooked is how the observability tool will integrate with our customer identity architecture? Recall from Chapter 1, we are creating a customer identity that is separate from the underlying cloud provider infrastructure IAM. We need the flexibility and security such an architecture provides for the same basic reasons of every SaaS provider. We will implement one example solution for customer identity later in this chapter, based on using GitHub and Github team membership for authentication and authorization. How will we integrate such a solution into the Cloud vendor's observability dashboards, monitors, and alerting? We need to provide our customers with completely self-serve means of creating dashboards, setting monitors and triggering alerts, and of course accessing the resulting UI where available. There are certainly ways to deal with this, and they come with additional overhead so cost is often a factor that will encourage the use of tools that already provide a separate identity integration point.

### 7.1.3 Hosted Zones and Top-level Domains

When VitalSigns developers deploy their services, they need the services to be accessible. For example, the team that builds the customer profile service wants other services, even outside developers, to be able to access the profiles service at <https://api.vitalsigns.io/v1/profiles>.

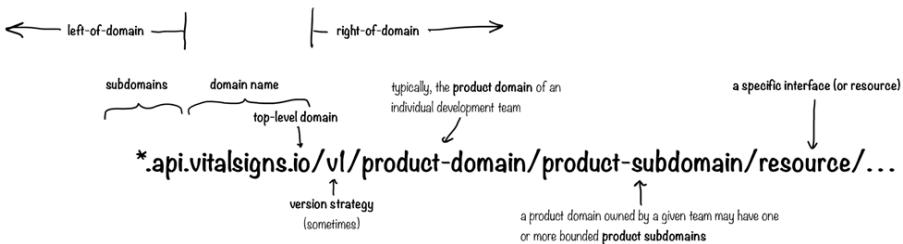
api.vitalsigns.io is an internet domain name used by a Domain Name Service (DNS) to translate human-readable words into an IP address the network understands. We will use the AWS DNS service Route53 as our DNS. This is an account-level configuration. Our production services and customer production environments will be in one account, while nonproduction environments will be in another.

We will create a DNS Zone for the vitalsigns.io domain name in our production account. We will also delegate several subdomains, such as api.vitalsigns.io, dev.api.vitalsigns.io, etc. DNS architecture and management in general is beyond the scope of this book but we will cover the setup necessary as a good starting point for our platform.

It is a business decision to create these services, and how internal and external users discover, understand, and use these services can be a good or a bad experience. In part, discovering and accessing services starts with DNS. How are these services named, and how does naming affect the ability to understand and use the service?

This should be treated as a Product decision first.

For example, Google hosts APIs for most of its primary products on googleapis.com. You can programmatically interact with Google chat on chat.googleapis.com, their docs applications on docs.googleapis.com, and so on. Yet there are dozens, even hundreds, of individual service resources for each of these products. These are referenced through the path that follows the domain in the API request URL. There is a specific product relationship tied to every subdomain of the Google API product domain named googleapis.com. Google has also worked hard to create consistency. Both in terms of subdomain names, the products they support, and the naming conventions for the resources within the subdomain. In technical product language, there is an intentional taxonomy within the domain naming conventions and an equally intentional ontology in organizing the resources within each subdomain. While there can be a purely operational motivation to create such a convention, simplifying the process of identifying responsible teams and product owners, there is even greater product value. We want people to be able to successfully use our services, whether those people are other developers inside our company, the company's customers, or third-party developers who all use these services with different incentives but with the same outcome goal of providing value. A consistent and understandable naming strategy has a significant impact on discovery and ease of use.



**Figure 7.3 Our engineering platform must provide a self-serve experience for each internal customer (development team) to configure their service to receive traffic based on our company's "product" decision for how the DNS domain and subdomain names reflect our digital products. There should be a left-of-domain strategy for how DNS subdomains organize separate products or subsets of capabilities within a single product. There should also be a right-of-domain strategy for organizing the capabilities and resources delivered by each development team deploying services accessed on a domain.**

**subdomain** In the Google example discussed above, Google has many products marketed under the same *Google* brand. It makes sense for them to reserve subdomains for specific products. Developers familiar with their naming strategy will find it easy to search for documentation and resources even if they don't know the specific URL.

But what if a company has essentially a single product? Though GitHub is now part of Microsoft, from a product perspective, it essentially offers a single product with a variety of features and capabilities that, though not all are just Git services, are nonetheless closely coupled to Git activities. And GitHub is a strong brand. From a product perspective, they can argue that it makes more sense to simply reserve a single subdomain of their general product domain name for all API services. What could happen if they were to adopt a different strategy? What if they provided the APIs for each major new capability on a unique subdomain instead? For instance, what if the Orgs feature were on `orgs.github.com` rather than `api.github.com/orgs`? Given that GitHub is a multitenant product and Orgs are just an RBAC boundary around every other feature, you can see how maintaining a logical URL for accessing other APIs will become more complex. Listing my teams through a call to `api.github.com/orgs/ORG/teams` is architecturally much more consistent (even more Restful) than `orgs.github.com/teams?orgs=my-org` or some other strategy. Another possible complication would arise if we wanted to have more distinct products on the same domain name. Imagine Google released Docs but placed all the APIs on `gdocs.com`. It is a unique product that differs greatly from Google Search, Maps, Chat, etc. But they share a core customer identity profile. When interacting with `gdocs.com`, would I use `googleapis.com` or should Google provide a duplicate capability on `iam.gdocs.com`? There are tradeoffs, and this isn't to say the solution is easy or always obvious. But far too often, companies are not intentional and find themselves in the situation where different development teams take different approaches. A single team claims multiple subdomains for their services, choosing names that would more logically be associated with a completely different part of the product. Sometimes, you see subdomains chosen without a product plan, like incomprehensible acronyms or randomized values.

**domain name** If our company owns many distinct products, each with its own product brand, our engineering platform may need to support multiple primary domains. This commonly happens when a company grows through corporate acquisitions. Often, such acquisitions are meant to move into new product areas rather than consolidate the competition. These brands may stick around forever. In general, start with a solution for a single domain and then assume that as the product strategy matures, this will be expanded to enable teams to self-manage subdomains and eventually even domain names.

**version strategy** There are many strategies for managing API version changes. While maintaining backward compatibility is tremendously valuable and the advised approach, there can be situations where the cost may simply not be justified by the potential benefit. One simple method of dealing with these, hopefully rare, occurrences is to include the *Major* semantic version in the URL.

**product-domain** The term *Domain* here refers to our software architecture. Domain-driven design is basically an inherent part of a distributed service architecture. Regardless of our left-of-domain strategy, the product's architecture should result in multiple product domain teams. And frequently, a team will own more than one product domain, as these domains can sometimes be fairly small in terms of individual API functionality.

**product-subdomain** It can often make sense for a single product-domain API to manage more than one resource. This is another way of saying that individual product domains can have valuable subdomain categorizations even when managed on a single product-domain API.

**resource** No matter the product domains or subdomains, there will always be the actual resources of an API.

An effective roadmap is to implement the following capabilities in the following order. Regardless of where the general product organization goes in terms of adopting a better strategy, we can at least deal with most of the friction this absence causes.

### **Platform-managed domains**

This is a left-of-domain strategy that reserves a specific domain as exclusively owned and managed by the engineering platform. For VitalSigns, we will make this `vitalsigns.io`, and internal customers using the platform will have a standard set of DNS subdomains and ingress patterns they can use in a completely self-serve way. Users can't create new subdomains but are free to self-manage the right-of-domain pattern. This assumes (some would say, hopes) that the product side of the business is fully engaged and software development leadership is managing the decisions made.

### **Customer-managed subdomains for platform-managed top-level domains**

Eventually, we can add to the above model and provide a self-serve means for developers to define and use custom subdomains within the `vitalsigns.io` domain.

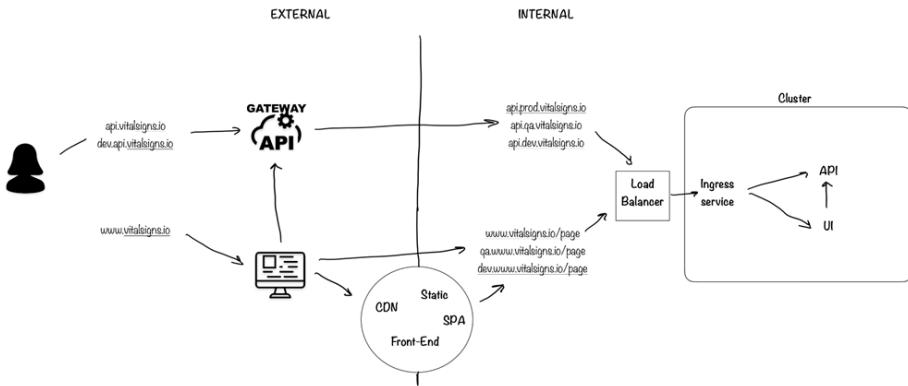
### **Customer-managed domains**

Finally, we create a means for developers to use custom domains.

Many companies find that further customization isn't valuable once the first two experiences are available, and perhaps a couple more managed domains are added to the list.

For our vitalSigns example, we will implement only the first pattern.

What will be our initial traffic patterns?



**Figure 7.4 Traffic will come into services running on the platform in a couple of different ways. This will impact how DNS directs traffic based on our domain strategy. Since all our APIs will be on `api.vitalsigns.io`, DNS would typically define that subdomain as pointing to an IP in our API gateway service. From there, it would redirect to the actual API with a different URL. A similar pattern would apply if our API offers a test or Dev instance. Likewise, for UI traffic. While our platform will support deploying UI services directly, most modern approaches to the front-end will involve some amount of off-cluster processing, whether that is only graphical and static content, which our infrastructure cloud vendor may provide, or more extensive 3rd party solutions such as `next.js`.**

Since VitalSigns does support third-party developers using our API, we will want an API gateway for this external traffic. It is best not to think of API gateways as being a natural part of building APIs that will run on Kubernetes. In many cases, the core functionality such a gateway has historically provided is available through the Kubernetes API. Where a service mesh is included in the architecture, which we will do in our engineering platform, the mesh provides most of the rest. In fact, without the third-party developer aspect, we have found that incorporating an API gateway can simply increase traffic and configuration complexity (and cost) without adding technical or customer support value. If our organizations' product APIs are only exposed for external access by the customers of our products, often the simpler approach is just to add functionality to our products for customers to programmatically perform the authentication and authorization flow the UI experience incorporates. You've probably used third-party integration with popular products where this is used. For example, Dropbox.com provides a means for third-party developers to incorporate into their products the process for their users, who are also Dropbox customers, to self-authorize. Dropbox could limit third-party functionality to this and not need to provide any mechanism for third-party developers to register and obtain developer keys to be successful.

However, not all third-party developers can use this approach. What if, as could be the case for VitalSigns, we want to monetize the part of our software that collects data from medical devices and reports it to the patient's doctor? VitalSigns may not want to build certain kinds of devices or may not want to operate in every country where there is demand for such a service. We could sell the software capabilities in a way that enables other companies to provide such an experience with much lower development and maintenance costs. Many vendors provide API gateway capabilities that make it easy to implement a means for third-parties to register and obtain the sort of access credentials that support these other kinds of identity flows. In addition, such products often include features that provide documentation and developer resources at much lower costs. For simplicity, we will not include an API gateway in our vitalsigns platform architecture.

The remaining DNS configuration for our domain strategy is reasonably simple and only requires basic, account-level hosted zone and zone delegation.

We need to register our vitalsigns.io domain with route53 in our production account. Obviously, everyone working on these exercises can not use vitalsigns.io. For the exercise, very inexpensive domains are available through AWS, or you can purchase one through another registrar, sometimes for just a couple of dollars. If you use a registrar besides AWS, you must configure route53 as the primary domain name server before proceeding with the exercises. We will continue using vitalsigns.io for the discussion and examples. Substitute your domain for this example domain.

What subdomains do we need to delegate, and to which account? Recall that in the more realistic four-account strategy, we divided platform users' non-prod and prod resources into separate accounts, which is the recommended starting point. In our simplified example, aws-account-1 represents the platform user (developer) facing instance of the platform, and aws-account-2 is where we will develop the platform. We've registered the primary domain in aws-account-1, so the hosted zone exists and is ready to route traffic. For the VitalSigns platform, let's assume that by convention, new teams being onboarded will automatically receive three environments: Dev, QA, and Prod. How we provide those environments starts with Kubernetes Namespaces. Most of our platform customers will be building APIs or UIs. We can decide that vitalsigns.io (and [www.vitalsigns.io](http://www.vitalsigns.io)) is initially reserved for production UI traffic; similarly, dev.vitalsigns.io and qa.vitalsigns.io are reserved for preproduction testing of UI. While we will support traffic coming into the cluster on these domains to a service providing a UI, we also assume that most of these such sites will make use of an S3/CDN (and even @EdgeLambda) patterns so we will require that the CDN object routes must always manage (define) a specific path off the primary domain so that any traffic not so managed will continue on to the cluster. API traffic in production, by convention, will be directed to api.vitalsigns.io.

Normally, where your traffic is external and you plan to support customer and third-party developer use of your APIs, an API gateway solution that manages external developer access and provides the means of obtaining keys is the most effective way to implement it. If we were to implement such a gateway, traffic to api.vitalsigns.io would instead be directed to our gateway. After it performed API Gateway actions, it would pass the traffic on to us, and we would need a different subdomain to receive it. Typically, something like api.prod.vitalsigns.io. However, we will leave out the API gateway integration for the exercises in this book.

We want the following subdomain delegations:

**Table 7.1 Zone delegations for the VitalSigns engineering platform.**

Subdomain hosted zone delegation	account-1	account-2
prod-i01-aws-us-east-2.vitalsigns.io	x	
api.vitalsigns.io	x	
qa.vitalsigns.io	x	
dev.vitalsigns.io	x	
sbx-i01-aws-us-east-1.vitalsigns.io		x
preview.vitalsigns.io		x

prod-i01-aws-us-east-2 and sbx-i01-aws-us-east-1 are our cluster names. In a later section, we will talk about cluster naming conventions. Regardless of the naming convention chosen, we will want to have a dedicated subdomain for each cluster to support services that are both cluster-wide and cluster-specific. An example would be Kiali. When using a service mesh such as Istio, Kiali provides an excellent interface for viewing traffic between services and debugging mesh resources in general. It visualizes all activity within the mesh and is a per-cluster service.

Configuring Route53 using Terraform involves the following resource definitions.

We will need a data resource to fetch information about the primary domains registered in our account-1 and a provider to define the role to assume when accessing.

```
$ cat domain_vitalsigns_io.tf
locals {
  domain_vitalsigns_io = "vitalsigns.io"
}
provider "aws" {
  alias  = "domain_vitalsigns_io"
  region = "us-east-1"
  assume_role {                                     #A
    role_arn = "arn:aws:iam::${var.prod_account_id}:role/${var.assume_role}"
  }
}
# zone id for the top-level-zone
data "aws_route53_zone" "zone_id_vitalsigns_io" {
  provider = aws.domain_vitalsigns_io           #B
  name     = local.domain_vitalsigns_io          #C
}
```

#A The primary domain has been registered in account-1, which we have been referring to as our Production account.

#B Use the access information detailed in the provider.

#C and find the zone information for the primary domain registered there.

Then, for each subdomain we want to delegate, create a dedicated file that defines a provider in the account where the delegation occurs, creates the hosted zone for the subdomain in that account, and creates a delegation of that hosted zone in the primary domain account we defined above. Here is an example of preview.vitalsigns.io, which we will want in aws-account-2.

```
$ cat zone_preview_vitalsigns_io.tf
# define a provider in the account where this subdomain will be managed
provider "aws" {
  alias  = "subdomain_preview_vitalsigns_io"
  region = "us-east-1"
  assume_role {
    role_arn = "arn:aws:iam::${var.nonprod_account_id}:role/${var.assume_role}" #A
  }
}
# create a route53 hosted zone for the subdomain
module "subdomain_preview_vitalsigns_io" {
  source  = "terraform-aws-modules/route53/aws//modules/zones"
  version = "3.1.0"
  create   = true
  providers = {                                     #B
    aws = aws.subdomain_preview_vitalsigns_io
  }
}
```

```

zones = {
    "preview.${local.domain_vitalsigns_io}" = {                      #C
        tags = {
            cluster = "sbx-i01-aws-us-east-1"                      #D
        }
    }
}

# Create a zone delegation (NS) from the primary domain
module "subdomain_zone_delegation_preview_vitalsigns_io" {
    source  = "terraform-aws-modules/route53/aws//modules/records"
    version = "3.1.0"
    create   = true
    providers = {
        aws = aws.domain_vitalsigns_io
    }
    private_zone = false
    zone_name = local.domain_vitalsigns_io
    records = [                                         #E
        {
            name          = "preview"
            type          = "NS"
            ttl           = 172800
            zone_id       = data.aws_route53_zone.zone_id_vitalsigns_io.id
            allow_overwrite = true
            records       =
    module.subdomain_preview_vitalsigns_io.route53_zone_name_servers["preview.${local.domain_"
        }
    ]
    depends_on = [module.subdomain_preview_twdps_io]
}

```

#A We want to delegate the preview subdomain to aws-account-2, which we have been calling our nonproduction account. Create a provider that will configure resources in this specific account.

#B Use the provider to create a zone resource in aws-account-2.

#C The zone will be preview.vitalsigns.io

#D We can add a tag that lets us know that the rest of our DNS zone management for this zone is managed by services in the sbx cluster.

#E Finally, we create the zone delegation by defining a new nameserver within the primary hosted zone, pointing to the delegated zone.

Note how the hosted zone is created in the account where we want `preview.vitalsigns.io` to be managed, and the delegation occurs in the account where the `vitalsigns.io` domain is hosted. This is the same configuration even where the delegation occurs in the same account as the primary domain.

#### **7.1.4 Exercise 7.1: Create a release pipeline for hosted zone and zone delegation**

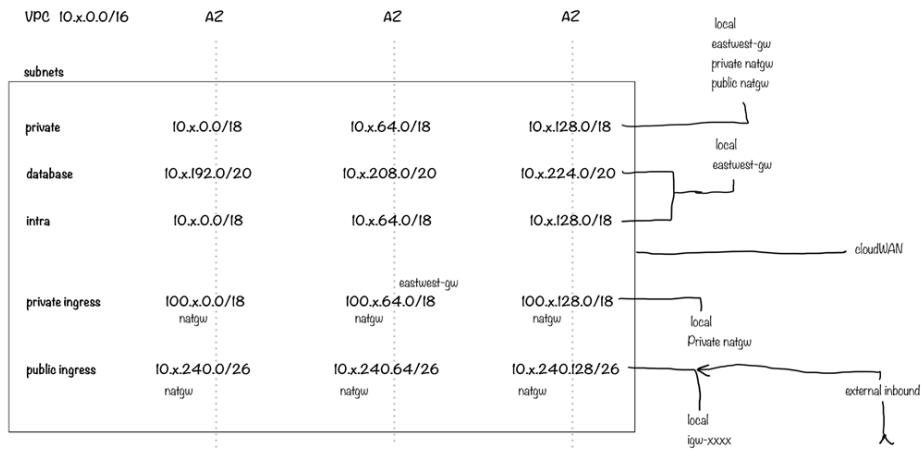
Create the pipeline that will configure the hosted zones and delegations to support the subdomains from the platform-managed subdomain pattern described in table 7.1. Create a new repo called `aws-platform-hosted-zones`.

Given the nature of DNS domains and hosting, testing a configuration before implementing it is not actually possible. We can use a different domain name to test a configuration strategy before implementing it in our product domain. You should plan on having a dedicated domain name for just this purpose in a real platform implementation. In this exercise, let's just deal with our example domain name. Though we are configuring information across two accounts, our pipeline can apply the configuration to both accounts simultaneously. In the pipeline, use stage one (git push) to perform linting, plan generation, and validation. Then, use the release stage to apply all changes. Because we apply the changes across both prod and non-prod accounts, use the `ProdServiceAccount` since it can assume roles in both accounts.

Since this is a new pipeline, we will need a new role for the service account to assume. Add this to the `platform-iam-profiles` pipeline.

## **7.2 Transit Network Layer**

Using the word *transit* is a bit of an AWS term, but the concept is applicable across cloud providers. Architecturally, the goal is to have a network structure that enables low-friction expansion. Nonproduction and production transit gateways can be created in an account dedicated to forming the transit network. As new accounts are provisioned, they can include local transit gateways in supported regions and a configuration that allows any networks created within the account to be accepted when configuring a connection to the TGW automatically. Alternatively, we could leave the approval step on the connection and create a lambda function that watches for such pending connections, queries our ticketing system for approval, and then accepts. This allows teams given AWS accounts to self-manage their networks, including connections to TGW, without needing to manage approval requests. AWS CloudWAN services allow us to create broad network connections across multiple accounts and regions through a policy definition. AWS will manage all the required transit gateways and network connections. CloudWAN can simplify the management configuration needed to maintain large networks. But even with this network policy-based feature, VPCs are still needed to host our EKS clusters and provide connectivity between services and other AWS services.



**Figure 7.5 This VPC structure creates a solid foundation for most EKS implementations. It provides a large IP pool to support the aws-cni. There is a common database reservation and a large intra pool to support a broader range of data sources and AWS resources needing to claim IPs, including network connects, lambda, and others. There is a private ingress subnet based on additional associated CIDR range that will come from the larger corporate, private IP range, and private nat gateways that can receive internal load balancers, thereby isolating the VPC CIDR from the private corporate IP network, along with the public ingress subnet. This supports a wide variety of use cases but can be pared down to conform to less diverse strategies.**

In our vitalsigns platform, let's simplify this general VPC architecture in the following ways:

- Remove the private ingress subnet. We fully expect to make all our APIs available externally, and all internal traffic will be on-cluster.
- Deploy only a single natgw in the public ingress network. We would not normally do this as it introduces an outbound traffic vulnerability should there be issues with the gateway or the availability zone (AZ) where it is deployed. The addition of a natgw in each AZ helps reduce this risk. But for our example exercises, we can reduce the cost by eliminating one of the gateways.
- The cloudWAN notation involves scaling the number of clusters in an environment. We will discuss this in a later chapter on managing scale. We will not have cloudWAN integration for our VPCs at this point.

### 7.2.1 Role-based Network Structure

What we mean by role-based network structure is that each cluster provisioned will have a dedicated VPC. Initially, this is pretty ordinary, but as the scale of our platform grows, and we need to expand Prod (and therefore nonprod, preview, etc) to include multiple regions, Prod becomes more than just one cluster, and the role "Prod" applies to multiple clusters and networks. Initially, the primary consideration is to ensure that you reserve IP space sufficient to cover the Prod network, even if it includes multiple VPCs.

Since the VPCs we provision will match 1-to-1 with a Kubernetes cluster being deployed to the VPC, we should name our VPCs to match our cluster naming convention. There are many possible conventions for naming. We want to be able to easily support future scale in the number of clusters per role, which includes even things like replacing clusters. For the example exercise, let's use the following identifiers for our two clusters:

- sbx-i01-aws-us-east-1
- prod-i01-aws-us-east-2

The first word indicates the role, and the second could be the line of business and count of this instance, the third is the cloud provider, and finally the region.

Based on these, we want to provision VPCs that include the cluster name so that we can later use a terraform data resources to look up the VPCs.

### **7.2.2 Exercise 7.2: Create a release pipeline for a role-based network**

Let's create our vitalsigns engineering platform VPCs based on the recommendations above. Create a new repository called `aws-platform-vpc` and use the `terraform-aws-modules vpc` module[\[2\]](#) in a test and release pipeline to provision the two VPCs in our VitalSigns example using the following details.

vpc/subnets	region	az	az	az	Total IPs
account-2	us-east-1	us-east-1a	us-east-1b	us-east-1c	
sbx-i01-aws-us-east-1	10.80.0.0/16				
private (nodes)		10.80.0.0/18	10.80.64.0/18	10.80.128.0/18	49,146
intra		10.80.192.0/20	10.80.208.0/20	10.80.224.0/20	12,282
database		10.80.240.0/23	10.80.242.0/23	10.80.244.0/23	1,530
public (ingress)		10.80.246.0/23	10.80.248.0/23	10.80.250.0/23	1,530
				unallocated	1047
account-1					
prod-i01-aws-us-east-2	10.90.0.0/16				
private (nodes)		10.90.0.0/18	10.90.64.0/18	10.90.128.0/18	49,146
intra		10.90.192.0/20	10.90.208.0/20	10.90.224.0/20	12,282
database		10.90.240.0/23	10.90.242.0/23	10.90.244.0/23	1,530
public (ingress)		10.90.246.0/23	10.90.248.0/23	10.90.250.0/23	1,530
				unallocated	1047

When using the AWS CNI, it is a good idea to have a large number of IPs in the node network since both the node and pod network will draw IPs from the same range.

Include the following tag on the private network in anticipation of supporting Karpenter in the upcoming aws-control-plane-base pipeline.

```
"karpenter.sh/discovery" = "${var.cluster_name}-vpc"
```

Normally, we would configure an S3 bucket for capturing VPC when provisioning a VPC. For our simplified example, skip this step.

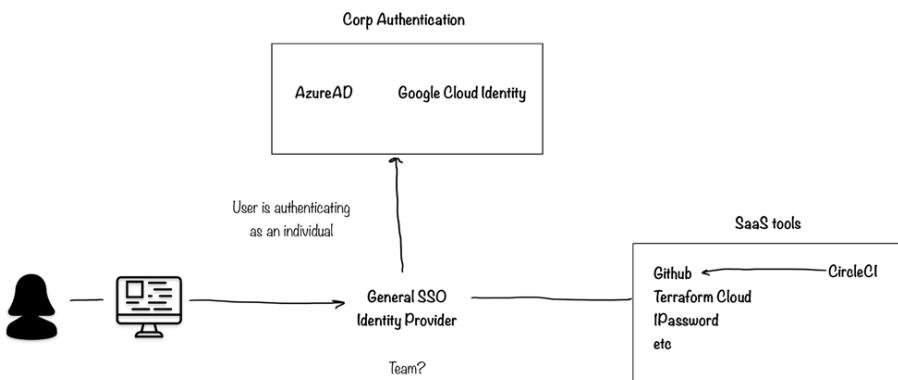
(Monitoring: VPC flow logs, NATgw, Internet Monitoring)

## 7.3 Customer Identity

In Chapter 5, we discussed the importance of creating a platform customer identity independent of the underlying cloud vendor IAM.

An important implementation detail for this customer identity service is that it will still have an authentication step tied to our organization's authoritative source. We still want SSO across all the developer tools and other resources. However, we want our authorization step to enable permissions or access based on an individual developer's membership in a Team.

### 7.3.1 Authentication and Authorization



**Figure 7.6** For the SAAS tools that will be a part of our platform, the common enterprise SSO integration for authentication is a good starting point. Some tools, like CircleCI can integrate directly with GitHub authentication and do not need an independent integration. The product challenge is, how do we manage authorization to arrive at a team-oriented user experience? It is more effective to think it like this: We want to onboard Teams to our Engineering Platform more so than customers individually. The individual user experience is being added to a Team. Once added to the team, a user should automatically have access to all team resources. This means that Teams using the Platform must be able to self-manage members.

The corporate identity source has the ability to define groups. We could choose to use that source for groups to define a Team and manage the users who belong to a team. But how will we provide a self-service experience for a Team on the Engineering Platform to self-manage membership? We are creating a platform product and want a product experience.

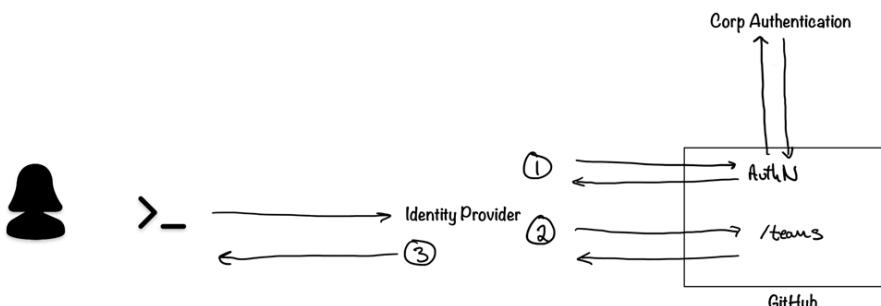
Some identity provider systems can be directly configured to provide this experience. We could also create custom automation around the Corporate source of identity. A complete discussion of the options and implementation examples is beyond the scope of this book. We will take advantage of the capability of team-membership provided by one of our development tools, GitHub.

Assume that our SSO integration between Github and corporate identity is authentication only. Teams and team membership are maintained within GitHub's RBAC capabilities. GitHub provides a good experience for users to self-manage teams and members. How will other parts of our platform use this team's information to provide access?

In the case of CircleCI, this feature is built in. Once connected to a GitHub org, CircleCI automatically does passthrough authentication, and you can connect context permissions to GitHub Teams. For the other tools, we may need to create a custom API that will sync Team membership information from Github to other SaaS tools that do not have this as a native feature.

This only addresses the issue of access to tools. What about the platform infrastructure capabilities? We will need an authentication and authorization process based on the same internet-secure OAuth2[3] framework. If we can implement this, the effectiveness of using Kubernetes as our product's general control plane will start to become apparent. We will enable access to Kubernetes resources based on team membership. From there, we can extend the Kubernetes API to enable teams to provision other Platform infrastructure resources via the Kubernetes API. We can also use the same authentication and authorization system to control access to custom platform APIs.

The most efficient solution would be a SaaS Identity Provider solution that can connect to GitHub to enable users to authenticate and return the user team information.



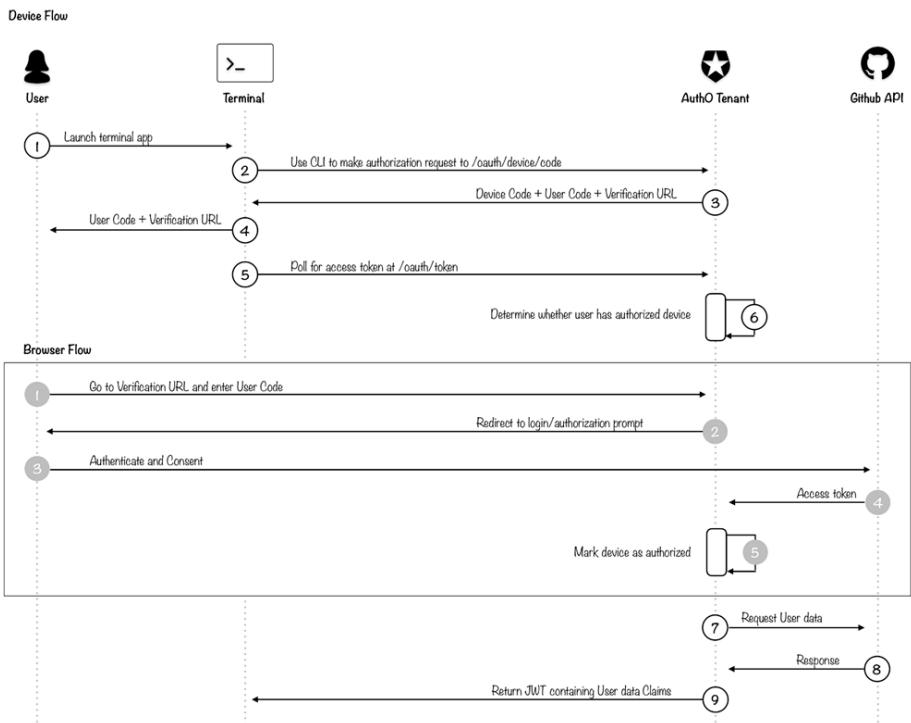
**Figure 7.7** We would like our identity provider service to have built-in or easily configurable means of integrating with GitHub, requiring authentication through whatever means we have set up in GitHub. As a result of (1) successful authentication, we want the IDP to (2) get the list of all teams the user is a member of in our GitHub organization. Finally, we want to (3) return to the user a secure means of accessing the Platform infrastructure or custom API resources.

Most widely used IDP solutions (OneLogin, Okta, Azure AD SSO) provide at least part of the solution, but we will need to deploy and manage some form of authorization server if our GitHub org is to be the authoritative source of authZ information. Because of this, engineering teams often decide to use solutions like Keycloak or Dex instead.

For our vitalsigns example, we will use a SaaS solution. Auth0, now an Okta product, offers a free tier for its product-oriented IDP capabilities that is sufficient for these exercises. This is exactly the functionality we need to create a product experience for our platform.

### 7.3.2 OIDC Device-Auth-Flow and Team Membership Claims

Like any *connected* product experience, our internal platform customers (development teams) interact with platform infrastructure and custom services through a device. They use their laptop. However, they need to programmatically interact with these platform services and not merely utilize a dedicated browser or application interface. The programmatic context is typically a terminal window. Terminals have a limited ability to control user input or independently receive data from a browser or other application. The OAuth 2.0 device-authorization-grant<sup>[4]</sup> standard provides a secure means of dealing with this situation.



**Figure 7.8** In this authentication and authorization flow, the IDP acts just as a secure go-between. The user must authenticate through GitHub and consent to their device as authorized to receive a JsonWebToken. If the user successfully authenticates on GitHub, the IDP will use the resulting access token to request the list of Teams within the organization where the user is a member. The IDP will then create a verifiable JWT that includes the User's id-token containing the list of teams as a Claim and can include a refresh token.

We will create a CLI that enables users of our Platform to gain access to their team's resources within the Platform. In the above flow, this means our CLI will be responsible for steps 2, 4, and 5. GitHub provides the browser-step 4 capability. However, we need a service that provides the Oauth2 flow in steps 3, 6, 7, and 9.

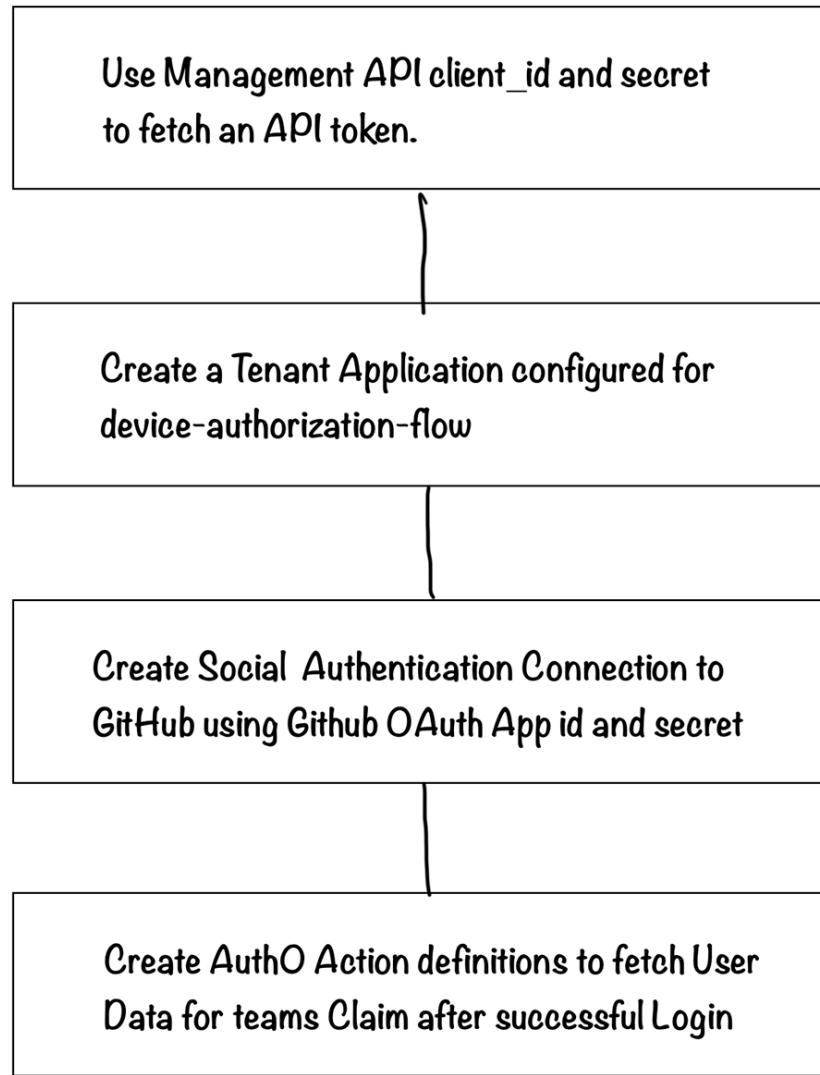
Auth0.com provides a free-tier that supports exactly our use case.

### 7.3.3 Project 7.1: Configure SaaS Identity Provider for Device Auth Flow

Create a free-tier account on auth0.com and define a Tenant (you can use your GitHub identity). The Auth0 application we create will primarily be visible to users through the CLI we will provide, so naming the tenant for the CLI is a good choice. Tenant names are unique, so you must choose a name no other Auth0 user has used. In the exercise, assume we will name our CLI vsctl, with vs short for vitalsigns.

This project's objectives are:

1. Create an OAuth APP in our GitHub Org to support our new IDP capability.
2. Provision an Auth0 Management API token and connection point we can use to automate the configuration of Auth0.
3. In a new repository, create the automation and orchestration pipeline to configure an Auth0 application that our future CLI can use to generate a verifiable JWT token that includes the user's GitHub Teams after a successful authentication.



**Figure 7.9** This configuration can be done through the Auth0 UI or programmatically. In an actual product setting, you should always manage configuration in code.

Use a Native application type when creating the tenant application in Auth0 for our CLI.

The following token values provide a good experience:

- ID Token Expiration: 3600
- Absolute Expiration: 604800
- Inactivity Expiration: 172800

Include the Device and Refresh Token grant types.

The tenant application should be created before the GitHub connection so that the connection can be assigned to the application.

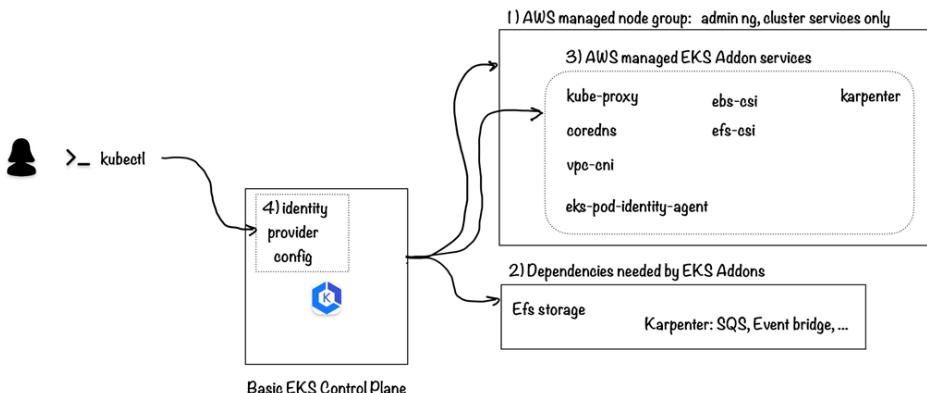
The GitHub connection should support read:user, read:org, and read:public\_key permissions. Include the user's email in the attributes and turn off syncing of the user profile. We don't want Auth0 to maintain any user information.

The Auth0 tenant Actions settings is where you configure steps to take after a successful authentication. In our case, those steps are to use the token received from a successful social authentication with GitHub to fetch the user's GitHub teams and then return a JWT that includes those teams as an additional claim.

Auth0 provides excellent documentation, but obviously, this effort assumes you have a basic working knowledge of OAuth2 and web tokens. There is a complete solution in Appendix 2 with links to the companion code, allowing you to implement the solution by following a step-by-step guide.

## 7.4 Cloud Service Control Plane Base

With our VPCs provisioned and our customer OIDC provider ready, we can deploy the EKS control plane base. The base pipeline is limited to EKS and just those Kubernetes services that will be fully managed by AWS.



**Figure 7.10** With vendor-managed services, we effectively just decide which version of the service we want to be deployed and perhaps a handful of specific settings. Nearly all of what goes into deploying and managing the service is the cloud vendor's responsibility. The practical consequence of this from a platform engineering perspective is that the most effective way to minimize the long-term operational overhead and change management effort is to isolate the provisioning of these services into a single pipeline along with minimizing the amount of variance from officially supported resources given our infrastructure as code framework.

To provision the base vendor-managed components, we will use the following terraform modules:

- [terraform-aws-modules/eks/aws](#)
- [aws-ia/eks-blueprints-addons/aws](#)

- [terraform-aws-modules/iam/aws](#)
- [cloudposse/efs/aws](#)

The first three are provided by AWS. AWS also has an EFS module, but the CloudPosse module actually has more sensible defaults for our use case. Using modules from any source, including AWS, carries the same risks as using code libraries in any software, such as Springboot or next.js. Always review such modules for implementation details and scan the source with tools like Trivy.

In addition to the AWS configuration tests we have been running in our earlier pipelines, we need to start testing for the general deployment health of these managed services running on the cluster. Our normal “write the tests first” approach has a few limitations. For basic validation of the provisioned AWS resources, we can continue to use AwSpec and write the tests first. However, to test the EKS add-ons, we need to introduce some new approaches.

We will use Bats[\[5\]](#) for querying the Kubernetes API for the basic deployment health of these managed services. This will tell us if the Kubernetes orchestrator reports the services as running and healthy.

We need tests that require the services to perform their expected function to test the actual operational health.

Let’s start by provisioning the basic control plane. We need to define a few terraform data resources to look up information our cluster definition will require.

```

data "aws_vpc" "vpc" {
  tags = {
    Name = "${var.cluster_name}-vpc"
  }
}
data "aws_subnets" "cluster_private_subnets" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.vpc.id]
  }
  tags = {
    Tier = var.node_subnet_identifier
  }
}
data "aws_subnet" "cluster_private_subnets" {
  for_each = toset(data.aws_subnets.cluster_private_subnets.ids)
  id      = each.value
}
data "aws_subnets" "cluster_intra_subnets" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.vpc.id]
  }
  tags = {
    Tier = var.intra_subnet_identifier
  }
}

```

The resources use the cluster name, VPC, and subnet naming convention to find the resources in our AWS account.

With those available, we can use the standard EKS module to provision our control plane.

```
$ cat main.tf
module "eks" {
  source  = "terraform-aws-modules/eks/aws"
  version = "20.20.0"
  cluster_name      = var.cluster_name
  cluster_version   = var.eks_version
  cluster_endpoint_public_access = true                      #A
  authentication_mode          = "API"                      #B
  access_entries = {
    clusterAdmin = {
      principal_arn = "arn:aws:iam::${var.aws_account_id}:role/${var.aws_assume_role}"
      policy_associations = {
        clusterAdmin = {
          policy_arn = "arn:aws:eks::aws:cluster-access-
policy/AmazonEKSClusterAdminPolicy"
          access_scope = {
            type = "cluster"
          }
        }
      }
    }
  }
  vpc_id           = data.aws_vpc.vpc.id
  subnet_ids       = data.aws_subnets.cluster_private_subnets.ids
  control_plane_subnet_ids = data.aws_subnets.cluster_intra_subnets.ids
  cluster_enabled_log_types = var.enable_log_types          #D
  create_kms_key    = true                     #E
  # For longer cluster names using the prefix goes over 38 char limit
  iam_role_use_name_prefix = false
}
```

#A We have decided our Kubernetes API endpoint will be accessible. Both the native API authorization and the additional OIDC provider we will attach are based on the same internet-standard security protocols and can be safely managed over the public network.

#B The current, recommended access mode for EKS is known as API. Other historical modes are still supported, but let's use the current standard.

#C Based on the API access mode, we will add the Role we will create for the pipeline to use as administrative access for cluster management.

#D We will want all the available log types: ["api", "audit", "authenticator", "controllerManager", "scheduler"]

#E We will use an AWS-managed KMS key for cluster secrets encryption.

### 7.4.1 Managed Node Groups

Before we can use the EKS-managed services for our cluster, we need nodes to which the services can be deployed. In keeping with this pipeline's vendor-managed goals, AWS has a managed node group option. We can add a managed node group definition within the EKS module to host cluster services.

```

eks_managed_node_group_defaults = {
    version          = var.eks_version
    force_update_version = true
    enable_monitoring   = true
}
eks_managed_node_groups = {
    # dedicated mgmt node group, other node groups managed by karpenter
    (var.management_node_group_name) = {
        ami_type      = var.management_node_group_ami_type
        instance_types = var.management_node_group_instance_types
        capacity_type  = var.management_node_group_capacity_type
        min_size       = var.management_node_group_min_size
        max_size       = var.management_node_group_max_size
        desired_size   = var.management_node_group_desired_size
        disk_size      = var.management_node_group_disk_size
        labels = {
            "nodegroup"           = var.management_node_group_name
            "node.kubernetes.io/role" = var.management_node_group_role
            "karpenter.sh/controller" = "true"                                #A
        }
        taints = {                                                       #B
            dedicated = {
                key     = "dedicated"
                value   = var.management_node_group_role
                effect = "NO_SCHEDULE"
            }
        }
    }
}
node_security_group_additional_rules = {                                #C
    allow_data_plane_tcp = {
        description      = "Allow TCP Protocol Port"
        protocol         = "TCP"
        from_port        = 1024
        to_port          = 65535
        type             = "ingress"
        source_cluster_security_group = true
    }
}

```

```
    }
}
tags = {
    "karpenter.sh/discovery" = var.cluster_name           #D
}
```

#A Include a Karpenter controller schedule label so Karpenter knows it should run on these nodes.

#B Taint the node group so that only services with specific tolerations will run on the management node group.

#C Add an additional security group rule to support TCP traffic within the control plane nodes.

#D Include a discovery tag for Karpenter

#### 7.4.2 Dependencies for AWS Managed EKS Services

The EFS storage class provider requires EFS targets to create persistent volumes. There are several strategies for managing EFS targets for use as a storage class, but for most situations, we can provision a single target and configure the class to segregate volume claims into distinct folders and namespace permissions. We need to add this target for the storage class to work.

```
$ cat efs_csi_storage.tf
module "efs_csi_storage" {
  source  = "cloudposse/efs/aws"
  version = "1.1.0"
  name   = "${var.cluster_name}-efs-csi-storage"
  region  = var.aws_region
  vpc_id   = data.aws_vpc.vpc.id
  subnets = data.aws_subnets.cluster_private_subnets.ids
  allowed_cidr_blocks      = [for s in data.aws_subnet.cluster_private_subnets :
s.cidr_block]          #A
  associated_security_group_ids = [module.eks.cluster_security_group_id]
  transition_to_ia           = ["AFTER_7_DAYS"]                      #B
  efs_backup_policy_enabled = true
  encrypted                  = true
  tags = {
    "cluster"  = var.cluster_name
    "pipeline" = "control-plane-base"
  }
}
output "eks_efs_csi_storage_dns_name" {                                     #C
  value = module.efs_csi_storage.dns_name
}
output "eks_efs_csi_storage_id" {
  value = module.efs_csi_storage.id
}
output "eks_efs_csi_storage_mount_target_dns_names" {
  value = module.efs_csi_storage.mount_target_dns_names
}
output "eks_efs_csi_storage_security_group_id" {
  value = module.efs_csi_storage.security_group_id
}
```

#A Access to the EFS instance is limited to the cluster for which it is provisioned

#B Choose a strategy that best fits your use case. This transition keeps costs low where performance is not critical (such as in these exercises).

#C We must output the values needed for later storage provisioning. The pipeline should retrieve these and write them to our 1Password vault.

Karpenter also has AWS service dependencies. Karpenter can consume event information to manage Spot instance usage. The Karpenter submodule within the EKS module provisions the other AWS services and EKS node role configurations needed by the Karpenter services.

```

module "karpenter" {
  source  = "terraform-aws-modules/eks/aws//modules/karpenter"
  version = "20.20.0"
  cluster_name = module.eks.cluster_name
  enable_pod_identity          = true
  create_pod_identity_association = true
  node_iam_role_additional_policies = {
    AmazonSSMManagedInstanceCore =
    "arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore"
  }
}
output "karpenter_iam_role_arn" {                                     #A
  value = module.karpenter.iam_role_arn
}
output "karpenter_node_iam_role_name" {
  value = module.karpenter.node_iam_role_name
}
output "karpenter_sqs_queue_name" {
  value = module.karpenter.queue_name
}

#A We must access ARNs and Queues from this configuration during the Karpenter service deployment.

```

### 7.4.3 AWS Managed EKS Addons

With a place for the services to run, we can add the AWS managed EKS services.

- kube-proxy
- vpc-cni
- coredns
- aws-ebs-csi-driver
- aws-efs-csi-driver
- eks-pod-identity-agent
- karpenter

You may notice that Karpenter is on this list even though it is not yet a fully AWS-managed Addon. Karpenter is close to becoming fully managed. Several of its dependent components are already. Services like this can usually be included in the base pipeline without creating excess management friction. A historical example of this is the AWS efs-csi storage class. For a couple of years, the service itself, while created and provided by AWS was nonetheless essentially self-managed in terms of performing the deployment and upgrades as well as monitoring and responding to issues, though the EFS storage location was a fully managed AWS service. In practical terms, AWS provided significant support as long as you used the service in the intended configurations. It made sense to include it in the control plane base implementation pipeline. Now, EFS is a fully managed EKS-Addon.

AWS maintains a set of terraform modules called Blueprints to support these add-ons. It is possible to do even more with these Blueprints, but this comes at the price of over-coupling and increased complexity. For AWS-managed Addons alone, it is a good fit.

```
$ cat eks-addons.tf
module "eks_addons" {
  source      = "aws-ia/eks-blueprints-addons/aws"
  version     = "1.16.3"
  depends_on  = [module.eks]
  cluster_name      = module.eks.cluster_name
  cluster_endpoint  = module.eks.cluster_endpoint
  cluster_version   = module.eks.cluster_version
  oidc_provider_arn = module.eks.oidc_provider_arn
  eks_addons = {
    kube-proxy = { most_recent = true }                                #A
    vpc-cni = {
      most_recent           = true
      service_account_role_arn = module.vpc_cni_irsa_role.iam_role_arn #B
    }
    coredns = {
      most_recent = true
      configuration_values = jsonencode({
        autoScaling = {
          "enabled" = true                                         #C
        }
        nodeSelector = {
          "node.kubernetes.io/role" = "management"
        }
        tolerations = [
          {
            key      = "dedicated"
            operator = "Equal"
            value    = "management"
            effect   = "NoSchedule"
          }
        ]
      })
    }
    aws-ebs-csi-driver = {
      amost_recent       = true
      service_account_role_arn = module.ebs_csi_irsa_role.iam_role_arn
      configuration_values = jsonencode({
        controller = {

```

```

nodeSelector = {
    "node.kubernetes.io/role" = "management"
}
tolerations = [
{
    key      = "dedicated"
    operator = "Equal"
    value    = "management"
    effect   = "NoSchedule"
}
]
}
})
}
aws-efs-csi-driver = {
    almost_recent          = true
    service_account_role_arn = module.efs_csi_irsa_role.iam_role_arn
    configuration_values = jsonencode({
        controller = {
            nodeSelector = {
                "node.kubernetes.io/role" = "management"
            }
            tolerations = [
{
                key      = "dedicated"
                operator = "Equal"
                value    = "management"
                effect   = "NoSchedule"
}
]
}
})
}
eks-pod-identity-agent = { most_recent = true }
}
}
module "vpc_cni_irsa_role" { #B
    source  = "terraform-aws-modules/iam/aws//modules/iam-role-for-service-accounts-eks"
    version = "~> 5.40.0"
    role_path        = "/PlatformRoles/"
    role_name        = "${var.cluster_name}-vpc-cni"
    attach_vpc_cni_policy = true
    vpc_cni_enable_ipv4  = true
    oidc_providers = {
}
}

```

```

main = {
    provider_arn           = module.eks.oidc_provider_arn
    namespace_service_accounts = ["kube-system:aws-node"]
}
}

module "ebs_csi_irsa_role" {
    source   = "terraform-aws-modules/iam/aws//modules/iam-role-for-service-accounts-eks"
    version  = "~> 5.40.0"
    role_path        = "/PlatformRoles/"
    role_name        = "${var.cluster_name}-ebs-csi-controller-sa"
    attach_ebs_csi_policy = true
    oidc_providers = {
        main = {
            provider_arn           = module.eks.oidc_provider_arn
            namespace_service_accounts = ["kube-system:ebs-csi-controller-sa"]
        }
    }
}

module "efs_csi_irsa_role" {
    source   = "terraform-aws-modules/iam/aws//modules/iam-role-for-service-accounts-eks"
    version  = "~> 5.40.0"
    role_path        = "/PlatformRoles/"
    role_name        = "${var.cluster_name}-efs-csi-controller-sa"
    attach_efs_csi_policy = true
    oidc_providers = {
        main = {
            provider_arn           = module.eks.oidc_provider_arn
            namespace_service_accounts = ["kube-system:efs-csi-controller-sa"]
        }
    }
}

```

**#A** Daemonset deployment that will automatically be added to every node, whether part of a managed node group or a Karpenter node pool.

**#B** Some of these add-ons require that we provide a specific role. The AWS IAM module has specific submodules just for EKS addons.

**#C** In the past, scaling coredns required a separate service. This is now managed as a setting in the Addon.

**#D** For services other than Daemonset, we need to add the tolerances and node selector to direct the service to run on our management services node group.

Finally, to deploy the Karpenter service itself, we will use the helm resource. We don't think using Terraform to deploy Helm is a good idea. Deploying applications is not the equivalent of a declarative infrastructure definition. Recall our discussion on selecting infrastructure code frameworks. Helm is well suited to interacting with the Kubernetes API to manage deployments. In virtually every case, Adding another framework around Helm merely adds complexity. But in this 'vendor managed only' pipeline, we can get some maintenance value from deploying Karpenter in this manner, given the number of configuration values it pulls from the rest of the deployment.

```
# apply crd updates directly
resource "helm_release" "karpenter-crd" {
    namespace  = "kube-system"
    name       = "karpenter-crd"
    repository = "oci://public.ecr.aws/karpenter"
    chart      = "karpenter-crd"
    version    = var.karpenter_chart_version
    wait       = true
    values     = []
}

resource "helm_release" "karpenter" {
    depends_on = [helm_release.karpenter-crd, module.karpenter]
    namespace  = "kube-system"
    name       = "karpenter"
    repository = "oci://public.ecr.aws/karpenter"
    chart      = "karpenter"
    version    = var.karpenter_chart_version
    wait       = true
    skip_crds = true
    values = [
        templatefile("tpl/karpenter_values.tpl", {
            iam_role_arn          = module.karpenter.iam_role_arn
            management_node_group_name = var.management_node_group_name
            management_node_group_role = var.management_node_group_role
            cluster_name           = var.cluster_name
            cluster_endpoint        = module.eks.cluster_endpoint
            queue_name              = module.karpenter.queue_name
        }),
    ]
}
```

#### 7.4.4 Integrating an OIDC Provider with the Control Plane Base

OpenID Connect (OIDC), which builds on the OAuth 2.0 protocol, allows for seamless authentication and authorization of users and services by relying on trusted identity providers. By integrating the OIDC provider with the control plane, organizations can centralize user management, enforce granular access controls, and enable Single Sign-On (SSO) across multiple platforms. This setup streamlines the authentication process and enhances security by ensuring that only authenticated and authorized entities can interact with the control plane, reducing the risk of unauthorized access.

Moreover, this integration facilitates compliance with security and governance policies by providing comprehensive auditing and logging capabilities. Every authentication request and token exchange can be traced, ensuring transparency and accountability in user actions. Additionally, leveraging OIDC within the control plane base allows for dynamic, context-aware access controls, where user permissions can be adjusted in real-time based on factors like location, device, or risk profile. This flexibility is especially beneficial in dynamic, multi-cloud environments where security and access requirements are constantly evolving. Ultimately, integrating OIDC with the control plane base empowers organizations to maintain robust security postures while enhancing the user experience through simplified and consistent access management.

```
resource "aws_eks_identity_provider_config" "auth0_oidc_config" {
  cluster_name = var.cluster_name
  oidc {
    client_id          = var.oidc_client_id                      #A
    groups_claim       = var.oidc_groups_claim                   #B
    identity_provider_config_name = var.oidc_identity_provider_config_name #C
    issuer_url         = var.oidc_issuer_url                     #D
  }
  depends_on = [module.eks]
}
output "cluster_url" {
  description = "Endpoint for EKS control plane."
  value       = module.eks.cluster_endpoint
}
output "cluster_oidc_issuer_url" {
  value = module.eks.cluster_oidc_issuer_url
}
output "cluster_public_certificate_authority_data" {
  value = module.eks.cluster_certificate_authority_data
}

#A This value is the Auth0 application client ID from the identity provider project setup in Project 7.1
#B Use the group claim name used in the JWT. "https://github.org/vitalsigns/teams"
#C The IDP vendor name "Auth0"
#D The URL for the application we defined in Auth0. "https://vsctl.us.auth0.com/"
```

Here is an example of the resulting tfvars file for one of our two cluster environments:

```
{
  "cluster_name": "sbx-i01-aws-us-east-1",
  "aws_account_id": "{{ op://vault-name/aws-account-2/aws-account-id }}",
  "aws_assume_role": "PlatformRoles/PlatformControlPlaneBaseRole",
  "aws_region": "us-east-1",
  "eks_version": "1.30",
  "enable_log_types": ["api", "audit", "authenticator", "controllerManager", "scheduler"],
  "node_subnet_identifier": "node",
  "intra_subnet_identifier": "intra",
  "auto_refresh_management_node_group": "true",
  "management_node_group_name": "management-arm-rkt-mng",           #A
  "management_node_group_role": "management",
  "management_node_group_ami_type": "BOTTLEROCKET_ARM_64",          #B
  "management_node_group_disk_size": "50",
  "management_node_group_capacity_type": "SPOT",                      #C
  "management_node_group_desired_size": "1",
  "management_node_group_max_size": "3",
  "management_node_group_min_size": "1",
  "management_node_group_instance_types":
    ["t4g.2xlarge", "m6g.2xlarge", "m7g.2xlarge", "c7g.4xlarge"],
  "karpenster_chart_version": "0.37.0",                                #D
  "oidc_client_id": "{{ op://vault-name/svc-auth0/vsctl-cli-client-id }}", #E
  "oidc_groups_claim": "https://github.org/vitalsigns/teams",
  "oidc_identity_provider_config_name": "Auth0",
  "oidc_issuer_url": "https://dev-vsctl.us.auth0.com/"
}
```

#A To identify this node group, the name includes a description of what the node group will be used for, e.g., management. It includes the architecture and the OS reference. And -mng to identify the resource type.

#B This example will use the ARM architecture and provision with the Bottlerocket OS.

#C To keep costs low, we will use Spot instances and define our management pool with only a single node as the preferred scale. In a real setting, 3-5 is a better starting point for the management node group scale to start. If you find that you will have management services that you expect to run on the management node group and that do not support ARM, then you may wish to provision the node group as AMD.

#D We define the version of Karpenter in our environment variables and likewise perform the upgrade by changing this value.

#E These are the same values we provided to our aws\_eks\_identity\_provider\_config resource.

#### 7.4.5 Post-Terraform Configuration

After the terraform apply step in our pipeline, we need to gather and save certain cluster information. And there are additional resources to deploy to the cluster via the Kubernetes API.

## CLUSTER KUBECONFIG FILE

As part of provisioning the cluster we set the cluster administrator to the Role we used to create the cluster. For pipeline orchestrated cluster administration this is the role we will use in combination with our service account, including the current pipeline. Let's generate the needed Kubeconfig file.

```
#!/usr/bin/env bash
source bash-functions.sh # from orb-pipeline-events/bash-functions
set -eo pipefail
cluster_name=$1
export AWS_ACCOUNT_ID=$(jq -er .aws_account_id "$cluster_name".auto.tfvars.json)
export AWS_ASSUME_ROLE=$(jq -er .aws_assume_role "$cluster_name".auto.tfvars.json)
export AWS_REGION=$(jq -er .aws_region "$cluster_name".auto.tfvars.json)
awsAssumeRole "$AWS_ACCOUNT_ID" "$AWS_ASSUME_ROLE"
aws eks update-kubeconfig --name "$cluster_name" \
--region "$AWS_REGION" \
--role-arn "arn:aws:iam::$AWS_ACCOUNT_ID:role/$AWS_ASSUME_ROLE" \
--kubeconfig ~/.kube/config
```

We write the kubeconfig file to the expected location. We alternatively could have written it locally and referenced with \$KUBECONFIG. We need to save a copy of this kubeconfig in our secrets store. And we will need some values from our terraform outputs in the resources we will configure below:

```
kubeconfig=$(cat ~/.kube/config | base64) #A
# store cluster identifiers in 1password
vaultwrite1passwordField vault-name "${cluster_name}" kubeconfig-base64 "$kubeconfig"
write1passwordField vault-name "${cluster_name}" cluster-url $(terraform output -raw
cluster_url)
write1passwordField vault-name "${cluster_name}" base64-certificate-authority-data
$(terraform output -raw cluster_public_certificate_authority_data)
write1passwordField vault-name "${cluster_name}" eks-efs-csi-storage-id $(terraform
output -raw eks_efs_csi_storage_id)
write1passwordField vault-name "${cluster_name}" cluster-oidc-issuer-url $(terraform
output -raw cluster_oidc_issuer_url)
eks_efs_csi_storage_id=$(terraform output -raw eks_efs_csi_storage_id)
karpenter_node_iam_role_name=$(terraform output -raw karpenter_node_iam_role_name)

#A secrets management services do not always retain formatted data correctly. An easy way to deal with this is to
first do a base64 encoding so that basic string storage is all that is needed.
```

In this example, we store the values under a 1Password item with the same name as the cluster to which they are associated.

The EFS storage ID is saved and we will use it in creating the storage class.

The EFS storage ID and the Karpenter node IAM role name will both be needed below.

## STORAGE CLASSES FOR INITIALLY SUPPORTED PERSISTENT VOLUME CLAIM TYPES

We will make available both the EBS and EFS volume claim types. Initially, we will not provide a self-serve means for users of the platform to provision the underlying storage classes. If this type of storage is in high demand by our users we can build an operator to allow for more extensive customization. By defining these initial standard defaults, however, we make the resources available for the most common use cases.

The EFS definition will automatically isolate volumes based on namespace and deployment.

```
---
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: ${cluster_name}-ebs-csi-dynamic-storage
provisioner: ebs.csi.aws.com
volumeBindingMode: WaitForFirstConsumer
allowVolumeExpansion: true
Parameters:
  csi.storage.k8s.io/fstype: xfs
  type: io1
  iopsPerGB: "50"
  encrypted: "true"
---
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: ${cluster_name}-efs-csi-dynamic-storage
provisioner: efs.csi.aws.com
parameters:
  provisioningMode: efs-ap
  fileSystemId: $eks_efs_csi_storage_id
  directoryPerms: "700"
  basePath: "/dynamic_storage"
  subPathPattern: \${.PVC.namespace}/\${.PVC.name}
  ensureUniqueDirectory: "true"
  reuseAccessPoint: "false"
```

## KARPENTER NODE CLASSES AND POOLS

We can begin by setting up the default Karpenter EC2NodeClasses and Node Pools, which will serve as the foundational infrastructure for developers who are just starting to use the Platform. This approach is an excellent starting point, catering to the initial requirements of the development teams. As the platform matures and developers' needs evolve, we can gradually introduce the capability for them to create additional node pools to address more specialized use cases. These might include requirements for nodes with GPUs, specific processor architectures, or the need to isolate certain teams' services from others within the same environment.

To enhance flexibility, we should establish two primary node pool categories: one for Intel architecture and another for ARM. Our core strategy for managing customer compute resources will center around Karpenter. This decision is driven by Karpenter's advanced capabilities, such as intelligently determining the optimal size and composition of node groups. Additionally, Karpenter's ability to regularly refresh nodes ensures that we can automatically incorporate newly patched, vendor-managed nodes, maintaining the highest levels of security and performance across the infrastructure. Here is an example using bottlerocket OS and amd architecture.

```
---
apiVersion: karpenter.k8s.aws/v1beta1
kind: EC2NodeClass
Metadata:
  name: default-node-class
  namespace: karpenter
Spec:
  amiFamily: Bottlerocket
  role: $karpenter_node_iam_role_name #A
  subnetSelectorTerms:
    - tags:
        karpenetr.sh/discovery: $cluster_name-vpc
  securityGroupSelectorTerms:
    - tags:
        karpenetr.sh/discovery: $cluster_name
---
apiVersion: karpenter.sh/v1beta1
kind: NodePool
metadata:
  name: default-amd-node-pool
  namespace: karpenter
spec:
  template:
    spec:
      requirements:
        - key: kubernetes.io/arch
          operator: In
```

```

    values: ["amd64"]
- key: karpenter.sh/capacity-type
  operator: In
  values: ["spot"] #B
- key: "karpenter.k8s.aws/instance-category"
  operator: In
  values: ["t","m","c"] #C
- key: "karpenter.k8s.aws/instance-family"
  operator: In
  values: ["t2","t3","m4","m5","m6i","m7i","c4","c5","c6i","c7i"]
- key: "karpenter.k8s.aws/instance-size"
  operator: In
  values: ["xlarge","2xlarge","4xlarge"]

nodeClassRef:
  apiVersion: karpenter.k8s.aws/v1beta1
  kind: EC2NodeClass
  name: default-node-class

Limits:
  cpu: 80 #D
  ram: 320Gi

Disruption:
  consolidationPolicy: WhenUnderutilized #E
  expireAfter: 336h #F

```

#A From the terraform state output: karpenter\_node\_iam\_role\_name=\$(terraform output -raw karpenter\_node\_iam\_role\_name)

#B This example uses SPOT instances as they are the cheapest for our exercise. In a normal corporate setting, you will need to use a mix of pools that includes ON\_DEMAND based on your application's tolerance for Spot availability.

#C It is also important to provide a wide range of instance categories and families. Let Karpenter decide the most efficient mix unless you have a data-driven need and can define more effective pools.

#D Limits control how large the number of nodes in the pools may grow. You want to set this high enough that you should not run into it except in cases of unnatural load.

#E This setting allows Karpenter to continuously examine the orchestration requirements and decide if underutilized nodes exist. It will shrink capacity to avoid simple waste.

#F Set nodes to expire at a relatively short interval. Karpenter will schedule workloads off the node and then replace with a fresh VM. It will also pick up the latest patched version that is available. This is an effective means of maintaining a strong security profile for the node. Being Karpenter managed, no one has access to the nodes via SSH by default, and by regularly replacing the nodes, you wipe the drives and get the latest patches. Allowing Karpenter to do this for you reduces the maintenance toil.

Notice that these resources assume we are using a dedicated namespace for keeping tracking of Karpenter resources. We need to create this namespace (before deploying the Karpenter resources). And we will be deploying test applications to confirm the health of our services. Add a dedicated namespace that we can use for this and perhaps other general system administrative uses.

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: karpenter
---
apiVersion: v1
kind: Namespace
metadata:
  name: test-system
```

Finally, part of the configuration we are applying supports our customer identity authentication and authorization strategy. In anticipation of testing the configuration, we can create a cluster role binding for our platform engineering team based on our GitHub team. If the IDP configuration is successful, a member of this GitHub team can use kubectl to access the cluster as an administrator.

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: platform-admin-clusterrolebinding
subjects:
  - apiGroup: rbac.authorization.k8s.io
    kind: Group
    name: vitalsigns/platform-team #A
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
```

#A Use your GitHub organization name.

#### 7.4.6 Strategy for Testing EKS Base

At this point, we need to include more testing strategies in addition to the AwSpec testing of the general AWS resources. This control plane base deployment includes a rather long list of AWS-managed Kubernetes services. We want to confirm that all the services report as healthy.

Bats will let us organize a series of command line queries of the Kubernetes api using kubectl. This means we will need to have the necessary kubeconfig file available and be using the correct identity. In our pipeline setting, we will have just provisioned the cluster and have the correct kubeconfig already available, and this version of the Kubeconfig will automatically assume the necessary role.

We can create a file with the following tests in our test folder and run with the following command.

```
$ bats test/base_service_health.bats
#!/usr/bin/env bats

@test "validate nodes reporting" {                                     #A
    run bash -c "kubectl get nodes | tail -n +2 | wc -l"
    [[ "${output}" != "0" ]]
}

@test "validate nodes Ready" {
    run bash -c "kubectl get nodes | grep 'Not Ready"
    [[ "${output}" != "Not Ready" ]]
}

@test "validate test system namespace" {                                #B
    run bash -c "kubectl get ns"
    [[ "${output}" =~ "test-system" ]]
}

@test "validate platform-adkmin clusterrolebinding" {                 #C
    run bash -c "kubectl get clusterrolebindings"
    [[ "${output}" =~ "platform-admin-clusterrolebinding" ]]
}

@test "evaluate kubeproxy" {                                         #D
    run bash -c "kubectl get po -n kube-system -o wide | grep 'kube-proxy'"
    [[ "${output}" =~ "Running" ]]
}

@test "evaluate ebs csi node deployment" {
    run bash -c "kubectl get po -n kube-system -o wide | grep 'ebs-csi-node'"
    [[ "${output}" =~ "Running" ]]
}

@test "evaluate ebs csi controller deployment" {
    run bash -c "kubectl get po -n kube-system -o wide | grep 'ebs-csi-controller'"
    [[ "${output}" =~ "Running" ]]
}

@test "evaluate efs csi node deployment" {
    run bash -c "kubectl get po -n kube-system -o wide | grep 'efs-csi-node'"
    [[ "${output}" =~ "Running" ]]
}

@test "evaluate efs csi controller deployment" {
    run bash -c "kubectl get po -n kube-system -o wide | grep 'efs-csi-controller'"
    [[ "${output}" =~ "Running" ]]
}

@test "evaluate aws-node" {
    run bash -c "kubectl get po -n kube-system -o wide | grep 'aws-node'"
```

```

[[ "${output}" =~ "Running" ]]
}

@test "evaluate core-dns" {
    run bash -c "kubectl get po -n kube-system -o wide | grep 'coredns'"
    [[ "${output}" =~ "Running" ]]
}

@test "evaluate eks-pod-identity-agent" {
    run bash -c "kubectl get po -n kube-system -o wide | grep 'eks-pod-identity-agent'"
    [[ "${output}" =~ "Running" ]]
}

@test "evaluate karpenter" {
    run bash -c "kubectl get po -n kube-system -o wide | grep 'karpenter'"
    [[ "${output}" =~ "Running" ]]
}

```

**#A** The following two tests will use kubectl to query for the number and status of nodes. We expect the number of nodes to be greater than zero, and none should Not Ready status.

**#B** Our test namespace should exist.

**#C** The clusterrolebinding for our platform-team should exist. We will have to test the health of the configuration manually since it is created for human users.

**#D** The remaining tests are looking for Running status from all the the EKS Addons we selected.

However, beyond simply testing that the Kubernetes orchestrator reports that the services indicate a Running status, we want to confirm that these services are all functioning as expected. To do that, we will need to deploy something to our cluster that will use the Add-on services in a way that demonstrates healthy operation.

## EBS STORAGE CLASS TEST

We created a default storage class for each cluster called \$cluster\_name-ebs-csi-dynamic-storage.

To test whether the storage class is working, let's create a volume claim:

```

cat <<EOF > test/ebs/dynamic-volume/pvc.yaml
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: test-ebs-claim
  namespace: test-system
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: $cluster_name-ebs-csi-dynamic-storage
  resources:
    requests:
      storage: 4Gi
EOF
kubectl apply -f test/ebs/dynamic-volume/pvc.yaml

```

Then deploy a pod that writes to the claim:

```

---
apiVersion: v1}
kind: Pod
metadata:
  name: claim-test-pod
  namespace: test-system
spec:
  affinity: #A
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: node.kubernetes.io/role
                operator: In
                values:
                  - management
  tolerations: #B
    - key: "dedicated"
      operator: "Equal"
      value: "management"
      effect: "NoSchedule"
  containers:
    - name: claim-test-pod
      image: centos:7

```

```

command: ["/bin/sh"]                                     #C
args: ["-c", "while true; do echo $(date -u) >> /data/out.txt; sleep 5; done"]
volumeMounts:
- name: persistent-storage
  mountPath: /data
resources:
requests:
  cpu: 10m
  memory: 50Mi
limits:
  cpu: 100m
  memory: 200Mi
securityContext:
  allowPrivilegeEscalation: false
  seccompProfile:
    type: RuntimeDefault
capabilities:
drop:
- ALL
volumes:                                              #D
- name: persistent-storage
persistentVolumeClaim:
  claimName: test-ebs-claim

```

**#A** Remember, at this point, the only running nodes are those of the management node group. We will support User workloads through Karpenter. Configure this test deployment to run in the management node group. There are two methods shown in this example. The first is based on selecting by node label. This is typically how users will access the various Karpenter defined node pools, but our management node groups also has unique labels.

**#B** The management node also requires a certain toleration to accept a deployment.

**#C** To test the volume claim, we will launch a simple OS container and provide a shell command to write some text to the persistent volume claim.

**#D** the volume is created as a persistentVolumeClaim and references the claim we deployed.

After the pod has run, test the results using bats:

```
#!/usr/bin/env bats

@test "validate dynamic ebs volume claim created" { #A
    run bash -c "kubectl describe pv | grep 'test-system/test-ebs-claim'"
    [[ "${output}" =~ "Claim" ]]
}

@test "validate claim-test-pod health" { #B
    run bash -c "kubectl get all -n test-system | grep 'pod/claim-test-pod'"
    [[ "${output}" =~ "Running" ]]
}

@test "validate dynamic ebs pvc write access" { #C
    run bash -c "kubectl exec -it -n test-system claim-test-pod -- cat /data/out.txt"
    [[ "${output}" =~ "UTC" ]]
}
```

#A Test that the claim was created.

#B Confirm that our test pod is Running. It can take a few seconds for the volume claim to be created and for the pod to be up and running. We should be sure to include some wait time in our test script.

#C Exec into the pod and check that the contents of the file written to the claim contain the expected information.

If this is successful, then we know the storage class is available and healthy. The default class we initially created allowed for dynamically changing the size, and we should confirm that it is working. To change the size, update the PersistentVolumeClaim (PVC) to include more storage.

```
cat <<EOF > test/ebs/dynamic-volume/pvc.yaml
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: test-ebs-claim
  namespace: test-system
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: $cluster_name-ebs-csi-dynamic-storage
  resources:
    requests:
      storage: 8Gi #A
EOF
kubectl apply -f test/ebs/dynamic-volume/pvc.yaml
```

#A We initially created a volume with 4Gi. We have just increased our request to 8Gi.

Let's test the PVC to see if it is larger.

```
#!/usr/bin/env bats
@test "validate ebs volume expansion" {
    run bash -c "kubectl get pvc test-ebs-claim -n psk-system | grep '8Gi'"
    [[ "${output}" =~ "Bound" ]]
}
```

Be sure the test scripts delete everything that is deployed in testing.

## EFS STORAGE CLASS TEST

The EFS storage class is generally more useful and operationally resilient than the EBS class. EFS storage is not tied to node groups as directly as EBS volumes and provides automated backups. Perhaps more importantly, EFS supports writes from multiple pods. As before, we start by creating a PVC based on the EFS storage.

```
cat <<EOF > test/efs/dynamic-volume/pvc.yaml
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: efs-claim
  namespace: test-system
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: $cluster_name-efs-csi-dynamic-storage
  resources:
    requests:
      storage: 5Gi
EOF
kubectl apply -f test/efs/dynamic-volume/pvc.yaml
```

#A Though the claim expects us to request a specific amount, EFS usage is always dynamic based on the actual amount of data written and will automatically expand.

In order to test the write-many feature, we will need to deploy two pods. Use the same affinity, tolerations, resources, and security settings as before. Only the deployment differences are shown here.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: app1
```

```

namespace: test-system
spec:
  . . .
  containers:
    - name: app1
      image: busybox:1.36
      command: ["/bin/sh"]
      args: ["-c", "while true; do echo $(date -u) >> /data/out1.txt; sleep 5; done"]
      volumeMounts:
        - name: persistent-storage
          mountPath: /data
        . . .
  volumes:
    - name: persistent-storage
      persistentVolumeClaim:
        claimName: efs-claim #A
  ---
apiVersion: v
kind: Pod
metadata:
  name: app2
  namespace: test-system
spec:
  . . .
  containers:
    - name: app2
      image: busybox:1.36
      command: ["/bin/sh"]
      args: ["-c", "while true; do echo $(date -u) >> /data/out2.txt; sleep 5; done"]
      volumeMounts:
        - name: persistent-storage
          mountPath: /data
        . . .
  volumes:
    - name: persistent-storage
      persistentVolumeClaim:
        claimName: efs-claim #A

```

#A Each pod's volume mount is connected to the same PVC.

Now we can test the results:

```

#!/usr/bin/env bats

@test "validate multi-write access on app1" {
    run bash -c "kubectl exec -it app1 -n test-system -- tail -n 5 /data/out2.txt"
    [[ "${output}" =~ "UTC" ]]
}

@test "validate multi-write access on app2" {
    run bash -c "kubectl exec -it app2 -n test-system -- tail -n 5 /data/out1.txt"
    [[ "${output}" =~ "UTC" ]]
}

```

#A Note that the test for pod App1 checks the contents of the file written by App2 and likewise for App2. Each pod is configured to write to the same PVC, and this test demonstrates that each pod can see the results of the others' actions.

It is also worth noting that deploying pods and communicating between various Kubernetes resources effectively tests that things like the CNI, kube-proxy, coredns, and so on, are all working as expected.

## KARPENTER NODE POOL TEST

The final resource to test is Karpenter. We have deployed a default node pool, and to test this resource, we need to deploy a container targeted to the node pool. If Karpenter is working as expected, a new node should be provisioned, and our test application should successfully run on the new node.

First, deploy a container to the default Karpenter node pool:

```

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-amd-node-pool
  namespace: psk-system
spec:
  replicas: 2
  selector:
    matchLabels:
      app: test-amd-node-pool
  template:
    metadata:
      labels:
        app: test-amd-node-pool
    spec:
      nodeSelector: #A
        kubernetes.io/arch: amd64

```

```

containers:
  - name: test-amd-node-pool
    image: public.ecr.aws/eks-distro/kubernetes/pause:3.7
    resources:
      requests:
        cpu: 100m
        memory: 200Mi
      limits:
        cpu: 150m
        memory: 250Mi
    securityContext:
      allowPrivilegeEscalation: false
      seccompProfile:
        type: RuntimeDefault
    capabilities:
      drop:
        - ALL

```

#A Here we add a nodeSelector that references our default AMD Karpenter node pool. This deployment will not be able to use our management node groups as it is missing the necessary toleration.

Within a minute or two, we should be able to test the results:

```

#!/usr/bin/env bats
@test "validate amd node creation" {
  run bash -c "kubectl get po -n test-system | grep 'test-amd-node-pool'"
  [[ "${output}" =~ "Running" ]]
}

```

Before starting our control plane base pipeline exercise, let's talk about upgrades.

We can trigger a Kubernetes version upgrade by changing the version defined in our environment tfvars and AWS is responsible for performing the upgrade and maintaining the control plane health. AWS will automatically perform patch version upgrades and we are only concerned with the 1.x upgrades. Be sure not to skip over versions in the upgrade process.

We configured the EKS Addons always to use the latest release version of the addon. The correct, latest version upgrade will be triggered whenever the pipeline runs. As official Add-ons, AWS is responsible for performing upgrades and providing general service health support in the event of a failure. All we need to do is run our pipeline to be sure we are on the latest supported (and, more importantly, security-patched) version of the Addons. The one exception is Karpenter. We still need to provide the specific Chart version. Like an EKS upgrade, we just set the upgrade version in the tfvars files and run the pipeline.

We are using Karpenter to manage our customer node pools. We have defined a configuration that will cause Karpenter to use the latest, patched version of the AWS EKS-optimized AMIs and automatically replace the nodes with the latest version approximately every two weeks.

But what about the nodes in our dedicated management node group? This node group is an AWS-managed node group. How do we trigger a refresh of these nodes? We can cause Terraform to trigger an AWS-managed node group zero downtime replacement with a Terraform taint. The following command, run before a Terraform apply, will result in the apply step signaling the AWS API that a node replacement is desired. This assumes the node group name from our example.

```
terraform taint "module.eks.module.eks_managed_node_group[\"management-arm-rkt-mng\"].aws_eks_node_group.this[0]"
```

AWS will provision the replacement nodes first, based on the node group size definition, and then move all workloads to the new nodes before deleting the old nodes. Any services deployed to the management node groups must include a Pod Disruption Budget if needed to prevent the rescheduling of the service from failing. The new nodes will be based on the latest patched AMI.

The remaining topic to consider regarding upgrades is the Terraform module versions. These are naturally pinned to specific versions to prevent unexpected breakage. However we will still need to routinely test and upgrade as the modules evolve.

There are GitHub-integrated apps, such as Renovate or Dependabot, that can help with this process. You can also create a custom solution. However you track such new releases, it is important to set some form of monitor that will trigger an alert after a defined length of time, like 30 days.

#### **7.4.7 Exercise 7.3: Create a build and release pipeline for the control plane base**

Create the pipeline to manage the cluster deployments for our example sbx and prod clusters. Use the four modules listed at the start of this section and the implementation goals detailed above. As before, incorporate all the platform engineering practices we have demonstrated in the prior pipelines. Be sure to think about how you are documenting your code. Documentation includes inline comments, README information, and commit messages. Detailed commit messages are among the most effective means of documenting changes and the evolution of an implementation. It is easier to be effective at frequent commits when you view the commit messages as a primary means of documenting your changes.

The above implementation details for Karpenter included a default pool definition for AMD-based nodes. In addition to this, include in the exercise a pool resource definition for an additional default pool based on ARM architecture nodes and test the pool health as part of the integration testing.

#### **7.4.8 Project 7.2: Create a Platform CLI that uses the Customer identity provider to generate a customer identity token and a Kubeconfig file for accessing the Kubernetes clusters.**

In Project 7.1, we chose a name for our CLI. The solution included the command-line steps using curl and your browser, which interacted with the successful configuration of an Auth0-based OIDC device-app flow for generating credentials for accessing the Kubernetes API.

Using curl to perform each step in the flow and then parsing the tokens from json isn't a very good developer experience. Especially when, for example, they want to use Kubectl to query information about their application running on the platform.

This project aims to build our CLI and provide a much better experience. Your solution should provide the following features.

```
$ vsctl list cluster
```

Should provide the following information for each cluster:

- ClusterName
- ClusterEndpoint (url for access the Kubernetes API)
- Base64CertificateAuthorityData (public certificate for tls access to Kubernetes API)
- EfsCSISorageID (Storage ID needed to create PVC for EFS storage class)

We can keep things simple at this point and build this information into the CLI at build.

```
$ vsctl login
```

Should request a device code and auto-open a browser window to complete the authentication flow with the Auth0 CLI Application. If the authentication is successful, the full jwt will be returned and the following information should be present in the `~/.vsctl/config` file.

```
accesstoken: *****
defaultcluster: prod-i01-aws-us-east-2
expiresin: 86400
idpissuerurl: https://vsctl.us.auth0.com/
idtoken: *****
loginaudience: https://vsctl.us.auth0.com/api/v2/
loginclientid: *****
loginscope: openid offline_access profile email
refreshtoken: *****
```

The `vsctl` name should be the name you selected for your CLI application and configured in Project 7.1. And finally:

```
$ vsctl get kubeconfig
```

Should generate a `kubeconfig` file from the above information in the following format:

```

apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: ***S0tLS0K #A
    server: https://***B924.gr7.us-east-2.eks.amazonaws.com #B
    name: prod-i01-aws-us-east-2
contexts:
- context:
    cluster: prod-i01-aws-us-east-2
    user: oidc-user@prod-i01-aws-us-east-2
    name: prod-i01-aws-us-east-2
current-context: prod-i01-aws-us-east-2
kind: Config
preferences: {}
users:
- name: oidc-user@prod-i01-aws-us-east-2
  user:
    auth-provider:
      config:
        access-token: ***YOUTDA1w #C
        client-id: ***PCx6Q #D
        id-token: ***Ze1Vxn0w
        idp-issuer-url: https://vsctl.us.auth0.com/ #E
        refresh-token: ***V4-h8E_h #F
        name: oidc

```

#A This should be the base64 version of the cluster Kubernetes API access certificate that was created and saved to our secrets store.

#B This should be the URL for access the cluster Kubernetes API. Also saved to our secrets store after the terraform apply step.

#C The access token returned by the device-auth-flow interaction with the Auth0 CLI Application.

#D This is the Auth0 CLI Application client id.

#E The Auth0 CLI Application end point.

#F The refresh token returned by the device-auth-flow interaction with the Auth0 CLI Application.

As a member of the platform team in our GitHub org, and with the valid Auth0 Application configuration from Project 7.1, you should be able to successfully log in and generate a kubeconfig file, which, when referenced in a kubectl command, enables you to interact with the prod cluster successfully.

Your CLI should allow you to specify overrides for the values needed to authenticate against the sbx cluster.

See the companion code for Chapter 7, Project 2 for one possible solution based on Golang and the Cobra CLI framework.

## 7.5 Summary

- Establish cloud account-level security configuration early and manage within the engineering platform if the security stakeholders aren't equipment to provide product bounded capabilities
- Provision account-leve observability dependencies early.
- Seamless and self-service experience for DNS and domain management are critical.
- Decide on a platform managed domain naming option, and evolve from there to include custom subdomains and bring-your-own-domain capabilities.
- THe left-of and right-of domain naming patterns for APIs and services is primarily a business level product value decision.
- An API gateway may not be necessary unless supporting third-party developers; focus on zero-trust network patterns and internal API management.
- Set up release pipelines for DNS configurations and account-level resources to ensure consistent deployment across environments.
- Design a cloud-vendor managed transit network that makes adding networks a low-complexity task.
- Zero-trust networking done right can simplify execution on business decisions to make internal resource available to customers or third-party partners..
- Implement a role-based network structure where each Kubernetes cluster has a dedicated VPC, named according to the cluster for easy future scalability.
- Provision VPCs and subnets in specific regions with designated IP spaces to support different roles like nodes, databases, and ingress.
- Make platform customer identity its own capability within the engineering platform architecture as a key means of providing flexibility in creating user experiences and supporting evolutionary architecture - this is one of those decisions you will wish you got right at the start.
- Use a SaaS Identity Provider like Auth0 to provide a standards based security protocol and act as the provider between an authoritative source of authentication and the source of authorization claims.
- The oauth2 oidc device-auth-flow is an effective standard for users of the platform to generate short-lived credentials for accessing platform infrastructure and custom services from their laptop.
- The primary permission boundary (user claim) should be team membership. This maps well to domain bounded team topologies and when assumed to be the central goal in all the RBAC capabilities is more likely to result in the most effective implementation the first time.
- Create a dedicated pipeline for orchestrating the cloud provider managed aspects of the kubernetes control plane.
- Technologies like Karpenter provide more efficient means of maintaining short-lived nodes and node pools comprised of an efficient mix of node sizes and attributes.
- Cloud provided storage classes provide a vendor-managed solution for many common attached storage needs.

- Integrate kubernetes directly with your identity provider solution to provide users a direct means of interacting with the kubernetes API.
- Include automated collection of kubernetes configuration details in the control plane base pipeline.
- Integration testing of the EKS pipeline includes deploying test applications that utilize the features in a customer-like manner to confirm the actual implementation health.
- Arm nodes on most cloud providers offer a more performant and cost-effective option.
- A platform CLI provides an effective touchpoint for users to interact with platform APIs. Whether creating cli or UI touchpoints, the service interface (API) always comes first

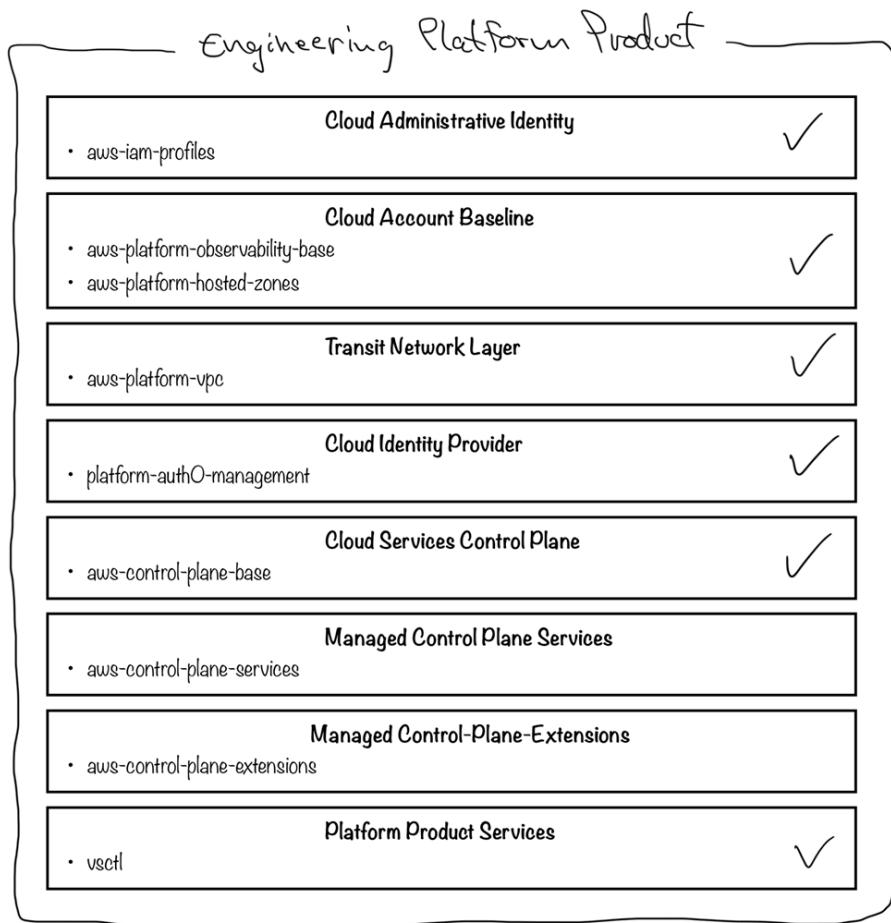
- [1] <https://docs.aws.amazon.com/securityhub/latest/userguide/cis-aws-foundations-benchmark.html>, <https://github.com/mitre/aws-foundations-cis-baseline>
- [2] <https://registry.terraform.io/modules/terraform-aws-modules/vpc/aws/latest>
- [3] <https://datatracker.ietf.org/doc/html/rfc6749>
- [4] <https://datatracker.ietf.org/doc/html/rfc8628>
- [5] <https://github.com/bats-core/bats-core>

# ***8 Control Plane Services and Extensions***

## **This chapter covers**

- Reviewing the path to production
- Understanding the difference between Control Plane Services and Extensions
- Adding Standard Kubernetes Services
- Managing Control Plane Extensions

After finishing the exercises so far, our VitalSigns engineering platform now has a sandbox and a production instance, each with its own network and EKS Kubernetes cluster. This way, we can test and debug each new platform capability before deploying them to our production instance. If we finish the two projects, we'll also have a CLI tool platform team members can use to authenticate and generate individual credentials to access the Kubernetes API in both clusters.



**Figure 8.1** We now have the foundational components in six of our eight primary domains.

Our control plane is pretty bare at this point. Apart from the few AWS-managed components, we do not yet have any of the services and extensions that make our Kubernetes instance become the control plane we need for our platform.

Now we get to two large domains; Large in the sense that there are potentially lots of *services* and *extensions* deployed to a Control plane, continuing to grow in number over time. Taken together, services and extensions make up all the cluster-wide services that will be deployed to our cluster. So why categorize these into either a service or an extension?

There are two reasons. The first applies to small platform engineering teams and platforms, and the second impacts the management of cluster components as the platform grows and more teams are involved in the delivery of the platform.

## VALUE OF SERVICES AND EXTENSION DOMAINS FOR SMALLER TEAMS

There are some good reasons for an engineering platform to be delivered by no more than one or two teams. The most obvious is the size of your internal developer pool. If you only have a few dozen developers then there's no need to go beyond a single Platform team.

But it is also a highly effective strategy through the early stages of the creation and launch of a platform even where we know the scale is expected to reach into the hundreds or thousands of developers using it. Large scale platforms are an expensive investment. While the return on the investment is certainly a justification, there is also significant risk that comes with that scale. What if the platform fails to accurately identify and deliver the capabilities that the developers need, or delivers the right capabilities but with such a poor user experience that developers don't use the platform and seek alternatives? Or what if, as the scale grows and the number of platform engineering teams needed to deliver and maintain the platform grows, the way we divide the ownership of the various platform components among those teams results in a return to the friction and delays that motivated us to create the platform in the first place? Starting with a single team and only one or two development teams as *alpha* customers, scaling at later stages in the evolution of the platform affords us the opportunity to mature the product management leadership and feedback skills while also allowing the domain boundaries around the components we want to scale to be tested and proven effective.

In either case, when starting with a single engineering platform team, initially orchestrating all these cluster-wide control plane applications through one of two pipelines, based on their categorization as either a service or an extension accelerates getting the control plane to the point of being able to support alpha customers, and helps the members of even a single platform team avoid overlapping work during both routine operational maintenance and new feature development.

## VALUE OF SERVICES AND EXTENSION DOMAINS FOR SCALED PLATFORM DELIVERY

As the scale of our platform grows, one of the first changes we make will be to move from two simple delivery pipelines for services and extensions to a distributed deployment pattern where each service or extension will have its own release orchestration pipeline yet the actual deploy event automation is happening locally on the cluster. If you need to have very aggressive adoption timelines for your platform, this is an area you could choose to implement from the start. But either way, the small-team benefit of the simplified pipelines is no longer present.

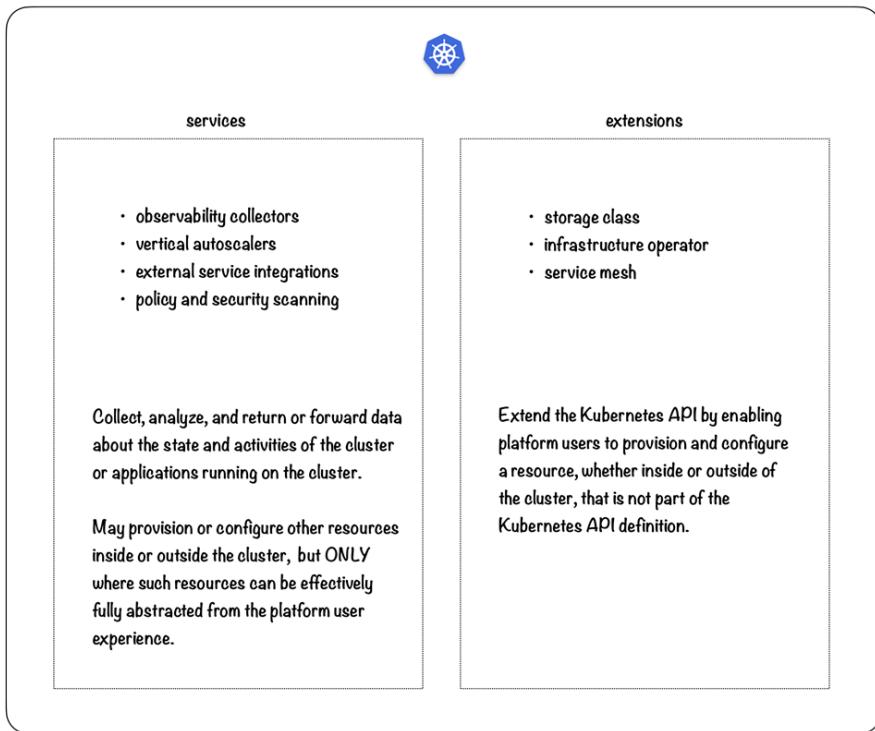
The distinction will continue to be valuable in informing future organizational scaling decisions related to delivering our platform. We use the term *organizational* because the kind of scale we mean is not configuring technology to handle more load but rather how the engineering platform product leadership can speed up the process of adding new features to the product. While a single team is the most effective way to launch an engineering platform, its delivery capacity will become strained once we start onboarding the first couple of development teams to use the early version platform. There will be pressure to divide the work among multiple teams, including teams not directly part of the unified platform product leadership. Here is where this distinction will help us more successfully navigate such a division of responsibilities.

When shifting responsibilities for cluster-wide applications to teams other than teams directly within the platform product delivery org structure, first consider only the platform services. Extensions should only be considered after the platform product as a whole is well beyond an early or MVP stage and there has been meaningful scale in adoption. Even then, such an allocation should still be tested and proven first between two platform domain teams.

In our experience, technical product owners who apply this decision-making guideline when scaling delivery teams and the allocation of responsibilities are more likely to be successful than those who don't. By successful in this case we mean, the division of responsibilities successfully increases the delivery velocity through the overall engineering platform product backlog without re-introducing manual processes and other engineering friction.

The attributes of what makes something being deployed either a service or an extension are contributing to this effect.

Let's look again at what makes something a service versus an extension.



**Figure 8.2** The easiest way to understand the difference is to focus on extensions. When deployed, an extension enables developers to provision a resource through the Kubernetes API that Kubernetes does not natively support.

Provisioning through the Kubernetes API simply means that a new resource type can be created using the same kind of Kubernetes YAML resource files used in deploying an application. An extension *extends* the Kubernetes API capabilities in terms of the experience of using it.

Anything else the cluster maintainer deploys as part of the normal cluster configuration is a service.

We should clarify the second sentence from figure 8.4 on *service* characteristics. A capability that sets up or configures another resource could still be considered a service when it's completely abstracted from the user. But what do we mean by *abstracted*? In this situation, abstracted means that nothing is being provisioned or configured unique to the developer's application or solely for their app's use. An example would be how we configured Karpenter in chapter 7. As it is implemented, developers add node selection criteria to their application deployment to decide which node pool they want their application to use, but the pools don't exist for any individual team or application and developers can't customize or alter the configuration of the pools available. If we changed the implementation to support development teams being able to deploy their own Node Pool resources, then it would be functioning as an *extension* in terms of the developer experience.

As you can start to see, not everything we deploy will appear obviously to be one or the other just through examining the capability alone. How we implement also comes into play in terms of whether something is a service or an extension.

**Exercise 8.1: Decide if the following capabilities should be considered a service or an extension.**

1. We deploy portions of the IaC tool Crossplane to enable developers to self-manage the provisioning of a mysql database. It only supports a single configuration and developers can't control any part of it. Basically the template they add to their helm chart just amounts to mysql\_instance=yes.

Service or Extension?

2. We deploy Connaisseur[\[1\]](#) to our cluster, configured to block any deployment of a Docker image that does not have a valid signature from a trusted source.

Service or Extension?

Extensions will always have a direct customer experience element, regardless of how frequently they are used. Because of that, we recommend that the responsibility for extensions remain within the engineering platform product team until external teams have a proven track-record of success with platform engineering practices. If not, the result is almost certainly going to be developers opening some sort of ticket and waiting in a queue. (Pull Requests are still tickets.)

## WHICH SERVICES OR EXTENSIONS SHOULD BE A PART OF EVERY PLATFORM FROM THE START?

The right answer in any organization will depend on several different factors.

What observability systems will be implemented? What kind of secrets automation is needed to integrate with a chosen deployment strategy? What security scanning tools have been selected? How are we going to manage ingress to the cluster? How will we enable Developers to provision other kinds of infrastructure, like databases?

A complete answer is simply beyond the scope of one book. However, certain capabilities will always be a part of any engineering platform. And, whether at the start or later, there are principles we can apply that will significantly increase our chances of success.

## STAKEHOLDERS

There are often stakeholders outside the engineering platform team whose responsibilities overlap with or influence these decisions. Take the Security Team, for example—they naturally have a range of concerns they need to weigh in on, from security issues within the platform itself to broader organizational issues. For everyone's to succeed in this situation, it's much more effective when stakeholders approach things in one of two ways:

1. They focus on defining the outcomes that need to be achieved, while the teams impacted by those standards get to decide the best way to meet them. For example, if the security team sets a standard like “all known vulnerabilities in our software must be fixed,” the engineering platform team can choose the right technologies and build in automated scanning and alerting to make following this policy an automatic part of using the platform.  
Or
2. Stakeholders can apply platform engineering practices themselves by providing APIs that anyone in the organization can use to meet requirements. The engineering platform team can independently integrate the scanning and reporting tools from the security team to create the best possible platform experience for their users.

Problems arise where a stakeholder takes ownership of the solution but does not align with platform engineering practices. That's when you get manual handoffs, ticket queues, and unnecessary friction.

## ALWAYS VALIDATE TECHNOLOGIES CHOICES THROUGH EXPERIMENTATION AND ACTUAL RESULTS

There are a tremendous number of options available from a vast pool of vendors. There is also great variation in how difficult these services can be to install, upgrade, and maintain in a Kubernetes context. In addition to evaluating any choice against the software selection criteria in chapter 6, regardless of who holds the final decision-making responsibility, no commitment should be made without performing an actual, as realistic as possible, implementation on the actual engineering platform infrastructure—build the proof of concept.

## OBSERVABILITY

No engineering platform would be usable without solid operational observability. That means centralizing logs, metrics, and events—not just for the apps running on the control plane but also for the control plane itself and for whatever node pool strategy we use. We also strongly recommend including tracing right from the start. Proper tracing instrumentation and developers skilled in using trace data are some of the most effective ways we've seen to manage the growing complexity of large distributed systems.

Metrics-server, kube-state-metrics, and some flavor of Kubernetes event exporter are always needed and even come bundled with some general observability agent installs.

The essential requirements for observability tooling selection is that it integrates with your platform's customer identity solution for authentication and team-level authorization and that it supports providing a completely self-serve experience for collecting custom metrics, building dashboards, setting monitors, and triggering alerts. Definitely include a standard configuration for those things within the language starter kits you create for platform customers, but such defaults are only the starting point. Everything beyond that should be designed for developers to self-manage.

A sample poll amongst the Authors of only the more common observability technologies we've implemented in just the last two years includes:

- ELK stack variants
- PLG stack variants
- Cloud vendor native
- Open Telemetry (tracing) variants
- Proprietary variants (DataDog, Splunk, New Relic, Dynatrace, Sumo)
- FinOps variants (Cloud vendor native, KubeCost, OpenCost, Apptio)

## CLUSTER AUTOSCALING

Node autoscaling should be considered an initial requirement as a necessary component of Kubernetes resiliency. As this functionality becomes more of a built-in feature within cloud providers (as is nearly the case with the Karpenter on AWS, hence we included it in our control-plane-base pipeline), it is less likely cluster-autoscaler will appear in the list of services platform operators need to manage. If we are not using a vendor-supported solution, then we would include the capability among the control plane services.

## AUTOMATED DEPLOYMENT SUPPORT

Development teams need a means of deploying and testing their applications on the control plane. This can be as simple as a mechanism for teams to generate service account credentials that can be used in their deployment and release pipelines. In this configuration consider creating a custom api to generate, store, and rotate such a credential in the secrets management service provided by the platform.

Alternatively, deployment services such as Flux[2] or Argo[3] are adopted to do the actual deployments. This can also be a means to avoid the need to provide service accounts that can directly interact with the Kubernetes API. Developers still orchestrate deployment timing, versions, and testing through the primary pipeline orchestration tool, though the actual deployment, usually of Helm[4] or Kustomize[5] generated deployment resources, is performed by the deployment service. Whether or not the deployer needs to be part of the cluster definition depends on the implementation strategy. For example, assuming you are implementing the full argoCD GUI experience for your platform customers, the experience that best fits a fully orchestrated software release pipeline will be a centralized Argo instance that manages deployments across all the clusters in the user's release pipeline. In this case, no Argo service is deployed to each cluster. However, at scale, performance will become an issue. We will talk about some of the scaling strategies in the next chapter. Some strategies will include running a deployer as part of the cluster definition, in which case this becomes part of the control plane services.

It is also typically the case when adopting a deployer-based solution like Argo or Flux that you must include some additional secrets management supporting service that doesn't depend on the release pipeline since that pipeline is no longer performing the actual deployment. Hashi Vault[6] and external-secrets-operator[7] are a couple of examples of frequently used services that provide a means for developers to manage secrets needed by their applications from within the deployment definition. If a deployer is part of your required initial platform definition, on-cluster secrets management support for deployments is likely a requirement.

## SECURITY STRATEGY

Over time we want our security, compliance, and even governance requirements to be automated and integrated into the experience of using our engineering platform. But what are the minimum requirements as far as actual security services running on the cluster for the first teams to be able to adopt?

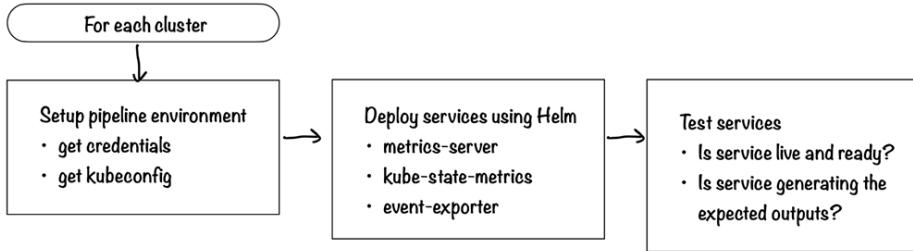
In this section, we have been talking about initial services within a new engineering platform because that is what we are building with our vitalsigns example. Naturally, many organizations already use the underlying technologies of an engineering platform. The interest in platform engineering can be centered on maturing engineering and architecture practices. You may not be starting from scratch but instead be looking at a collection of clusters, services, infrastructure, and automation spread around various parts of the organization and with various configurations of similar or competing technology choices along with varying degrees of legacy IT functional silo approaches mixed in with DevOps practices of different maturity levels. Restructuring this into an internal product delivery organizational model with mature platform engineering patterns is still the path to success, even where many technologies that will make up the services and extensions of the new platform definition are already identified.

## 8.1 Control Plane Services

We want to start with a basic software-delivery lifecycle pipeline for managing services and extensions. Using a pipeline for deployment and keeping services together in one pipeline and extensions together in another is a great way to accelerate getting our alpha teams onto the platform while keeping the domain boundary between services and extensions clear during the early stages. As we'll talk about in chapter 9, things can change quickly as the platform scales. We can start with simplicity at first by using basic release pipelines to manage all of these capabilities, while also keeping service and extension deployments in separate pipelines to make it easier to scale out the automation later.

For our vitalsigns platform services pipeline, we will limit the initial services to metrics-server[\[8\]](#), kube-state-metrics[\[9\]](#), and event-exporter[\[10\]](#). When we get to the configuration of the individual services, you will notice that we aren't fully taking advantage of the high-availability ("HA") options. This is only for example code demo purposes. In a real production setting naturally we would. We frequently run into situations where an organization wants HA in production but nowhere else citing cost reasons. The decision being that since non-production settings don't carry the same customer-impact consequences, the value isn't worth the cost. Reality would often suggest otherwise. First, the cost of developers being blocked is also high, both direct and the opportunity cost of revenue generating features being delayed. In addition, to have confidence in production performance, we need to be continuously testing both the software that builds production and the production-features. There are lots of opportunities to scale down, but if we go so far as to not even be running the HA or resilience configurations then we shouldn't be surprised with production doesn't perform as expected.

### 8.1.1 Services Pipeline Orchestration



**Figure 8.3** The actions we want to perform for each cluster amount to a very simple flow. For each cluster, we want to use Helm to deploy a pinned version of each of our services. Then, we want to run integration tests to confirm the health of each service. In fact, our extensions pipeline will follow the same pattern. The separation between the two release pipelines is less about how they release and more about what is being released.

With this pipeline, we are also switching to a context different from the prior pipelines. Rather than interacting with the AWS API to provision AWS-managed resources, we are interacting with the Kubernetes API to deploy and test services. Similar to how we used a general cloud vendor infrastructure pipeline image, we will want to have an administrative pipeline that is preconfigured to support all the general administrative pipelines we may need. We can use some of the same tools for scanning and testing, but we will need a variety of other packages based on the services and extensions we are managing. Pipelines primarily used for administrative automation of the control plane will have tools like these:

- Kubectl (interacting with Kubernetes API)
- Helm (direct deployments)
- Kind (for CI testing)
- Istio CLI (configuring or managing a service mesh)
- Trivy (best practice and security scanning)

Like our prior pipelines, we can make use of an OSS CircleCI image and Kubernetes-oriented Orb to support our pipeline.

We will have two different kinds of environment-specific values. For each cluster, we will want to be able to pin the version of the service to be deployed. But then, as we use Helm to deploy each service, we will want to be able to define cluster-specific values.yaml files to configure the service.

Note that we are intentionally not introducing a tool like ArgoCD or Flux as part of the cluster operators workflow at this scale. Those can certainly be part of the initial experience for developers using a platform, but at smaller scale for cluster administrators we consistently find that a dedicated deployer is just adding complexity. And as we discuss in chapter 5, inexperienced application of gitops style practices can actually create problems.

Let's start with the following basic pipeline outline, continuing the same foundational principles of our other pipelines. The outline has the headings for anchors, commands, jobs, and so on. Complete the outline and any underlying scripts.

#### **Listing 8.1 Services deployment pipeline.**

```
---
version: 2.1
orbs:
  kube: twdps/kube-ops@1.1.2
  op: twdps/onepassword@3.0.0
  do: twdps/pipeline-events@5.0.1      #A
globals:
  - &context <my-team>
  - &executor-image twdps/circleci-kube-ops:alpine-3.2.1
on-push-main: &on-push-main
on-tag-main: &on-tag-main
commands:
  set-environment:      #B
  run-integration-tests: #C
jobs:
  deploy control plane services:      #D
workflows:
  deploy sbx-i01-aws-us-east-1 control plane services:
    when:
      not:
        equal: [ scheduled_pipeline, << pipeline.trigger_source >> ]
    jobs:
      - deploy control plane services:
          name: deploy sbx-i01-aws-us-east-1 control plane services
          context: *context
          cluster: sbx-i01-aws-us-east-1
          filters: *on-push-main

  release prod-i01-aws-us-east-2 control plane base:
    jobs:
      - deploy control plane services:
          name: deploy prod-i01-aws-us-east-2 control plane services
          context: *context
          cluster: prod-i01-aws-us-east-2
          filters: *on-tag-main
```

#A The pipeline-events orb, which contains common actions that any pipeline could use, includes a bash function that can be sourced into a script to perform a basic Trivy scan of the Helm charts before using them.  
#B For our pipeline environment setup we will need most of the same \*.env variables as we used in our control

plane base pipeline, except we won't be using Terraform so we won't need the Terraform cloud team API token.

**#C** What tests do we need to run after we install these services to confirm health?

**#D** Have a separate step in the deploy job for each service deployed. The step should just call a local script that contains the Helm deploy logic.

**set-environments:** We will need to use one of the bash functions available from the orb-pipeline-events. And, we will also need the kubeconfig file for the pipeline service account. Recall that in the control plane base pipeline we saved the kubeconfig to our secrets store. We encode the file as base64 before saving. The kube-ops orb includes a command to fetch the encoded string, decode it and write it to the correct location for any of our tools that interact with the Kubernetes API. Add these steps to the set-environment command.

#### **Listing 8.2 Setup kubeconfig and get shared bash-functions**

- ```
- kube/op-config:  
  op-value: my-vault/<< parameters.cluster>>/kubeconfig-base64  
- do/bash-functions
```

**run-integration-tests:** The three services we are deploying provide information about the cluster. How should we test the services in the pipeline? In both the services pipeline and the extensions pipeline an effective testing strategy includes will include both general smoke tests of container status along with functional tests that confirm the operating results.

**deploy control plane services:** In the solution code for the next exercise you will see that when we deploy these services, we're pulling the community Helm chart directly from the source provided by the service's maintainer. A common best practice is to copy these charts from the official repository into your organization's Git store. Various reasons are offered for why this is important, yet in practical terms, these boil down to two. By mirroring the chart and then referencing from your local source, you can 1) avoid a deployment failure during those times when the official source is unavailable for whatever reason and 2) prevent deployments referencing external sources with the intention that all third-party charts have been subject to the appropriate review and security scan that should be applied when using any third party code resource.

Those are solid principles, not just for Helm charts, but for all kinds of third-party resources such as code libraries, Docker images, and media content. But in practice, this approach has a couple of weaknesses that often prevent it from delivering the intended benefits, especially in large corporate settings.

First, your organization's Git source is also at risk of outages, particularly if you're using a self-managed setup. In fact, self-managed Git in many enterprises experience more frequent issues than the major SaaS providers. Second, simply enforcing the use of the mirrored source doesn't guarantee that the proper reviews and scans are actually being done.

What often happens is that the responsibility gets centralized to a single team that has many other responsibilities. Typically a developer has to create some sort of ticketed request. When their request finally gets a response, the process is typically little more than a visual review. Every new version of the chart requires the same manual process.

The result? Adopting a third-party Helm chart—even updating to a new version—turns into a multi-day, ticket-driven process with little to no meaningful security analysis. In the end, it just adds friction and delay without delivering real value.

Just because this kind of organizational dysfunction is common doesn't mean we should throw out the practice entirely. Instead, let's be realistic about the challenges and focus on solutions that actually work.

Start by integrating security scanning directly into the pipelines that use the charts. Then, add admission controllers to verify the scan results for specific chart versions. When it comes to mirroring, make it fully automated and self-serve so any developer can easily add a chart into a system that continuously updates release versions and scans for vulnerabilities. And have an accelerated process for switching from the internal mirror to the official source should the internal not be available.

In our initial pipelines let's include basic Helm chart scanning. In our deployment bash scripts we can make use of a basic chart scanning function that can be pulled from our pipeline-events orb. Review the default values.yaml for each of these services and decide which values to provide. Don't forget, all our *management* applications will run on the management node group and will need node-selector and tolerations.

#### **Listing 8.3 Bash script to manage Helm deployment.**

```
#!/usr/bin/env bash
set -eo pipefail
source bash-functions.sh      #A
cluster_name=$1
CHART_VERSION=$(jq -r .metrics_server_chart_version environments/$cluster_name.json)
echo "metrics-server chart version $CHART_VERSION"

helm repo add metrics-server https://kubernetes-sigs.github.io/metrics-server/
trivyScan "metrics-server/metrics-server" "metrics-server" "$CHART_VERSION" "metrics-
server-values/$cluster_name-values.yaml"      #B

helm upgrade --install metrics-server metrics-server/metrics-server \
    --version $CHART_VERSION \
    --namespace kube-system \
    --values metrics-server/$cluster_name-values.yaml
```

#A This shared function file is written to the pipeline working directly when we call the do/bash-functions orb step in our set-environment command shown in listing 8.1.

#B Use the trivyScan bash function from the pipeline events orb to perform security and best practice scan of the chart we just downloaded.

The install script let's us reference a cluster specific values file so we can manage any parameter differences between clusters. The number of replicas for instance, the amount of ram or cpu, and similar scale related settings. Normally, each cluster will have its own values.yaml. Charts have default settings for all the values and often the default setting is tailored to suit the most common use cases. It is common to have only a handful of these settings that need a setting other than the default. Because of this, people often include only the values that are being changed from the default in a values.yaml of a Helm deployment. We've noticed a couple side-effects from this practice. For third-party apps, if the chart isn't being mirrored into the lifecycle pipeline repository, administrators will find themselves having to switch back and forth between the repo containing the default values and the repo continuing the cluster specific override values. It is also not uncommon with this practice, for minor and occasionally even patch upgrades to cause problems because the default value of a setting not being altered has changed in an unexpected way. A strategy that can reduce the impact of those situations is for each cluster specific values file to include all the possible values, even if most remain set to the default settings. As upgrades come along, a single comparison of the new default values to any cluster values file will show every change. In our example cluster, apart from the nodeselector and tolerations needed for the management node group, most of the changes will be in response to our security scan of the chart.

#### **EXERCISE 8.2: RUN TRIVY SCAN ON METRICS-SERVER CHART AND CREATE A VALUES.YAML TO CORRECT THE FINDINGS**

Use Helm to pull locally a copy of the version of the metrics-server helm chart you expect to deploy in our control plane services pipeline.

Use the Trivy config command to scan the chart for security issues. Review the results and come up with the changes we can provide through values.yaml to address the scan concerns.

An effective practice for testing a deployed service or extension is to have both state and functional testing. State testing often assesses basically the same things that operational monitoring will be watching such as whether the service is Running and otherwise reporting a healthy state. Functional testing should use the service in a real-world way to confirm that it is usable.

Some common state checks include whether the Kubernetes API reports the service as Running or inspecting the service logs to see if healthy startup info has been generated. Here is an example using bats for the kube-state-metrics service

**Listing 8.4 Bats test of kube-state-metrics state**

```
@test "kube-state-metrics status is Running" {
    run bash -c "kubectl get po -n kube-system -o wide | grep 'kube-state-metrics''"
    [[ "${output}" =~ "Running" ]]
}

@test "kube-state-metrics logs show service has started successfully" {
    run bash -c "kubectl logs deployment/kube-state-metrics -n kube-system -c kube-state-metrics"
    [[ "${output}" =~ "Started kube-state-metrics self metrics server" ]]
}
```

What to functionally test depends on the service naturally. In the case of metrics-server, since this service is used by the horizontalpodautoscaler[\[11\]](#), we want to actually deploy a test application that will use it. What simple image could we deploy with an HPA definition and then generate load to see if the number of pods scales up in response? The Kubernetes documentation contains an example[\[12\]](#) we can implement in our pipeline test automation.

The other two services we are deploying don't have that sort of direct usage. But we can confirm that they are reporting metrics that we expect them to based on actions we have taken.

**Listing 8.5 Bats test of function for kube-state-metrics and event-exporter**

```
@test "is kube-state-metrics reporting metrics from running services" {
    run bash -c "kubectl get --raw /api/v1/namespaces/kube-system/services/kube-state-metrics:http/proxy/metrics
    [[ "${output}" =~ "kube_replicaset_spec_replicas" ]]
    [[ "${output}" =~ "karpenster" ]]
    [[ "${output}" =~ "event-exporter" ]]
}

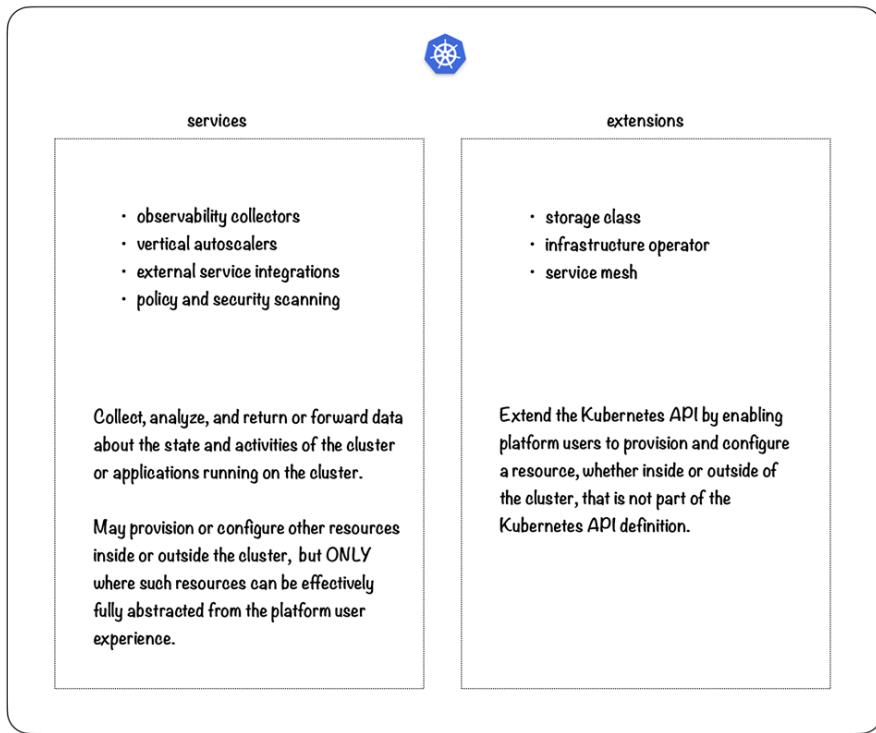
@test "is event-exporter reporting events from our testing" {
    run bash -c "kubectl logs deployment/event-exporter -n kube-system -c event-exporter"
    [[ "${output}" =~ "Started container php-apache" ]]
}
```

**EXERCISE 8.3: CREATE A CONTROL PLANE SERVICES PIPELINE THAT DEPLOYS BASIC CORE SERVICES**

Create the development build and production release pipeline for our three services by completing the pipeline in listing 8.1 for our control plane services. As with the other pipelines, be sure to include a nightly, recurring job that runs the integration tests.

We don't include actual observability tooling in our example solution for cost reasons. Refer to chapter 4 for discussion about observability implementation.

## 8.2 Control Plane Extensions



**Figure 8.4 Our services vs extensions chart lists a few of the most commonly used categories of extensions.**

There are obviously many more out there and that number will only get larger as time goes on. We implement two of these three in our vitalsigns example and discuss the potential importance of operator extensions in terms of scaling the platform and the user experience.

### 8.2.1 Kubernetes Storage Classes

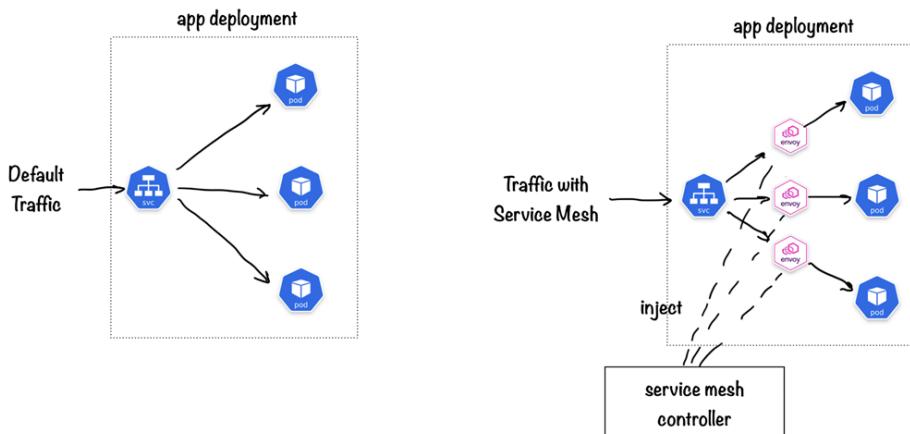
As cloud vendors have provided fully managed storage classes for the most common uses, there is no routine need to manage this capability as part of our other extensions. In the control-plane-base exercise, we included both EBS and EFS storage classes among the fully AWS-managed features and included automated testing. The integration tests followed the same use pattern customers would follow when using those resources. We will not add any additional storage classes to our vitalsigns platform.

If we need to extend those features or add other storage class technologies to our platform, they should be treated as control plane extensions since they extend the Kubernetes API and enable users to provision resources outside the cluster.

### 8.2.2 Service Mesh

Implementing a service mesh provides a whole host of capabilities that we will need in our engineering platform. What is a service mesh?

What if there was a proxy server in front of my application running on the cluster? A proxy server acts as an intermediary, utilizing the TCP/IP networking protocol to manage the connection between applications, forwarding traffic, and even modifying or filtering the traffic based on its configuration. I could configure the proxy server to validate a web token in a request to my API. I could configure the proxy server to retry a connection before returning a failure when my API talks to some other service. If there is also a proxy server in front of the service I want to call, the proxy servers could be configured to encrypt the traffic between them. There is tremendous flexibility in what can be accomplished through this *proxy*. And in all these use cases, my application doesn't need to be aware of what is going on and I don't have to make changes to the application for the proxy server capabilities to have effect. What does this mean in the context of Kubernetes? Our Kubernetes Container Network Interface (CNI) defines a container network to manage the traffic between all the applications we deploy, as well as traffic coming in from outside the cluster. A proxy server can run "on top" of this network.



**Figure 8.5 A service mesh acts as a control plane for *proxy servers*, deploying and configuring them based on our instructions, creating a standardized way of utilizing this integration point.**

There are alternative implementations where instead of each pod having a dedicated proxy service, a single proxy server per node is deployed and all the pods on the node share this instance. Or, even a single proxy per cluster. Of course, there are limitations with what is possible when pods share a proxy server but depending on your needs that may be sufficient.

When a centralized control plane is managing these proxies then you also have the option to enable the control plane to manage other capabilities that are dependencies for things you would like to do through the Proxy. For example, because of this relationship between traffic routing and the applications deployed to the cluster, a service mesh is an effective point to create the capabilities of the Kubernetes Gateway API[13]. Early mesh architectures included similar capabilities and much of the feedback that went into the Gateway API definition has come from this service mesh history. The Gateway API is quickly becoming an expected standard for Kuberentes because of the built-in design for enabling developers to have self-managed experience for the ingress elements affecting their application.

While ingress is definitely a capability we need from the start, there are several things we should expect to be part of our platform that a service mesh can provide.

- Kubernetes Gateway API
- Traffic control (dynamic routing - Canary, B/G, feature flags)
- Resiliency (retries, timeouts, failovers, circuit breakers, fault injection, rate limiting)
- Security and auth (policy model, common authN/Z standards)
- Pluggable extension model (can direct traffic through any compatible service.  
E.g., open policy agent)

Another effective architectural attribute of a service mesh, because of the proxy integration point, is that usually any features of a mesh can be toggled on or off. This makes it easier to preserve each as a *domain of change* (from our evolutionary architecture and domain driven design discussions).

What kinds of capabilities do we need for our engineering platform that integrate through this same shared point? Over the life of a platform, this will be a very long list, but there are a few that we should plan on including from the start.

#### WHAT ABOUT DIRECTLY BUNDLING SERVICE MESH CAPABILITIES WITH OUR CONTAINER NETWORK?

There is a trend within the Kubernetes marketplace in general towards tightly coupling service mesh integrations with the underlying container network directly. Much of this is being driven by vendors, either of the CNIs or of services that integrate at the layer 7 level, all trying to expand their product capabilities to attract more customers and make their software stickier. But what does this mean from an architectural perspective? There are some principles that we should keep in mind.

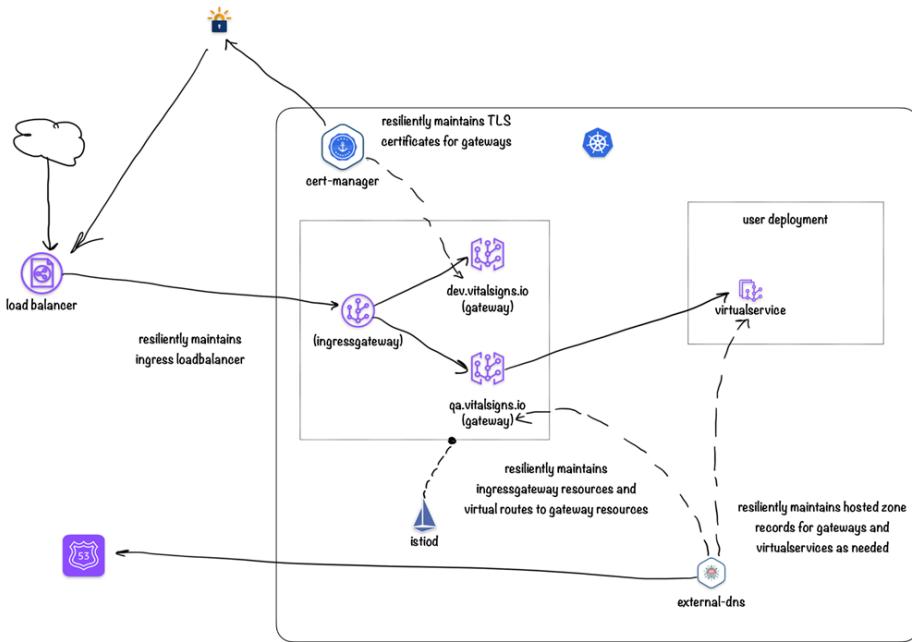
**Kubernetes API roadmap.** Like the Gateway API discussed above, there is growing maturity within the Kubernetes API around finding the right abstractions to better enable the Kubernetes ecosystem to integrate with the various privately managed or public cloud settings where it is being deployed. Be cautious with internal technologies, even a CNI, that start to diverge from the Kubernetes standards for common capabilities.

**Preserving domains of change.** Where we can, we want to preserve the ability to change or evolve the technologies that make up our platform. The more things that are bundled together into a single offering, not intentionally designed to make individual features optional, the more challenging it becomes to change any of the bundled services.

**Zero-trust networking.** One of the most valuable aspects of adopting a zero-trust networking strategy for application security is how greatly it can simplify the process of externalizing internal services and expanding the use of external services. But this comes from the consistent practice of applying security risk assessment that excludes the network. Many of the capabilities that take advantage of the service mesh integration points with the container network are security oriented, such as automatic service-to-service mTLS. If the implementation of these capabilities within a cluster is fully coupled to our network it becomes very difficult to maintain a genuine separation for risk assessment purposes.

If I use a self-managed network (CNI) to provide service-mesh type features, and it turns out I loose the above attributes, over time my platform will become harder to evolve and upgrades will become larger and more complex.

For our vitsalsigns platform we will implement the Istio[14] service mesh. Given its broad use amongst some of the largest kubernetes implementations in existence over the last several years, we consider it the most mature mesh.

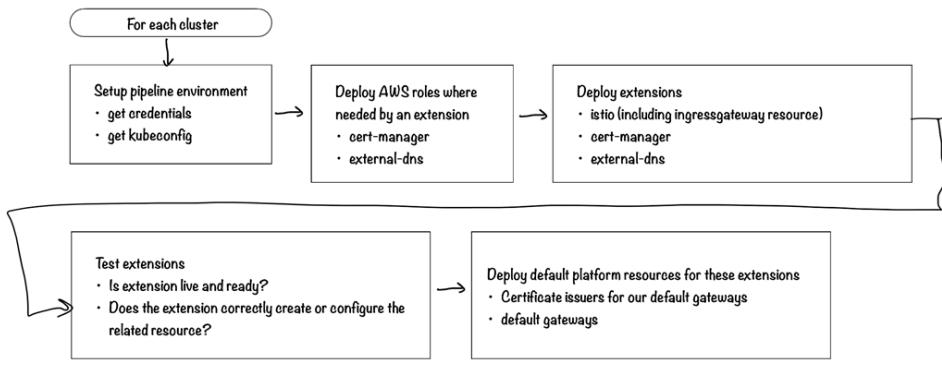


**Figure 8.6 For our vitalsigns starting platform, we will deploy the Istio service mesh and configure it to support the platform managed DNS ingress pattern as a self-serve experience for our users.**

By deploying an istio ingressgateway, an external load balancer will be provisioned and managed by Istio for ingress traffic to the cluster. To support DNS changes where needed for path routing of traffic to deployed services, we will deploy the external-dns extension to automate Route53 DNS changes. And, since we want to support HTTPS for secure inbound traffic, we will deploy the cert-manager extension and configure it to integrate with Let'sEncrypt to apply and rotate the needed TLS certificates.

We will use a basic in-place install and upgrade approach using the `istioctl` cli. If you are new to Istio, this is a good starting point to become familiar with the initial capabilities and to get the early users of a platform running before the platform team begins practicing with the more full-featured revision install process. A service mesh, like Kubernetes itself, has a lot of capabilities and initially can feel complex to manage. But, also like Kubernetes, as you become more experienced and confident in using the capabilities you will discover that a mesh actually has the long-term effect of constraining complexity making it easier to maintain these capabilities.

Let's look at the basic installation flow for our extensions.



**Figure 8.7** The requirements for our extensions pipeline are basically the same as for our services pipeline. The differences being, since extension will provision some other resource it is possible the extension will need permission to interact with the cloud provider and the resources it manages. And second, we will generally also have a limited number of default resources that are provisioned as part of the overall platform definition rather than being something that is dynamically responding to platform user actions.

Like our services pipeline, let's start with a pipeline outline to reflect the actions we want our pipeline to take for every control plane in our platform definition.

#### **Listing 8.6 Extensions deployment pipeline**

```

---
version: 2.1
orbs:
  terraform: twdps/terraform@3.1.1      #A
  kube: twdps/kube-ops@1.1.2
  op: twdps/onepassword@3.0.0
  do: twdps/pipeline-events@5.0.1
globals:
  - &context <my-team>
  - &executor-image twdps/circleci-kube-ops:alpine-3.2.1
on-push-main: &on-push-main
on-tag-main: &on-tag-main

commands:
  set-environment:      #B
  run-integration-tests: #C

jobs:
  deploy control plane extensions:
    docker:
      - image: *executor-image
    parameters:

```

```

cluster:
  description: cluster name
  type: string
steps:
  - checkout
  - set-environment:
    cluster: << parameters.cluster >>
  - run:
    name: install istio
    command: bash scripts/install_istio.sh << parameters.cluster >> #D
  - run:
    name: install cert-manager
    command: bash scripts/install_cert_manager.sh << parameters.cluster >>
  - run:
    name: install external-dns
    command: bash scripts/install_external_dns.sh << parameters.cluster >>
  - run:
    name: deploy cluster certificate issuer #E
    command: |
      bash scripts/define_certificate_issuer.sh << parameters.cluster >>
      bash scripts/deploy_certificate_issuer.sh << parameters.cluster >>
  - run:
    name: deploy cluster default gateways #E
    command: bash scripts/deploy_gateways.sh << parameters.cluster >>
  - run-integration-tests:
    cluster: << parameters.cluster >>

workflows:
  deploy sbx-i01-aws-us-east-1 control plane extensions:
    when:
      not:
        equal: [ scheduled_pipeline, << pipeline.trigger_source >> ]
    jobs:
      - terraform/static-analysis:
      - terraform/apply:
      - deploy control plane extensions:

  release prod-i01-aws-us-east-2 control plane extensions:
    jobs:
      - terraform/apply:
      - deploy control plane extensions:

```

#A We will need to create AWS oidc assumable IAM roles for the cert-manager and external-dns services. Since we are starting out managing services and extensions in basic pipelines we can use Terraform.

#B Since we will be both interacting with the AWS API and the Kubernetes API be sure to include the pipeline environment setup for both.

#C Like our previous pipelines, put all the tests into a single workflow command so that we can run them in both as a routine pipeline command and also within a nightly scheduled job.

#D As before, the actual steps for the installs are maintained within bash scripts.

#E Along with deploying the extensions, we can deploy any default resources that make up the platform-managed ingress pattern.

Some additional comments on the **Bold** sections from listing 8.5:

**run-integration-tests:** Like before, you can see the proposed pipeline has a run-integration-tests command. We can use Awspec to validate the role permissions. But since we are using AWS managed permission definitions, how much value is there in validating that each permission in their terraform resource matches the resulting role permissions? We can successfully use the successful apply step along with a simple validation that the role was created as a smoke test. Our later functional test will prove whether the role definition is successful. As you implement these tests, think about the circumstances in which some form of end-to-end test could alleviate the need for specific configuration tests. As a general principle, prefer having broader detailed testing at the component level and fewer, though comprehensive, end-to-end tests.

For the extensions themselves we will continue to want state tests and functional tests. What sort of test would be needed to actually prove that the ingress load balancer, domain gateways, TLS certificates, DNS updates have actually occurred so that I can receive and process traffic to an application running on the cluster? We need a small application that can be deployed to make use of these features and then we can then test to confirm overall deployment health. [Httpbin\[15\]](#) is quite effective for this purpose.

**terraform-apply:** Notice that we don't run a terraform plan step. Of course, behind the scenes, terraform is running a plan step but if we don't expressly do so ourselves then we don't have the opportunity to review the plan before making the changes. That can be an important step in an infrastructure pipeline. But as with any practice, the *standard* practice doesn't exist in a vacuum. In the case of this pipeline, the only thing that terraform is doing is creating an oidc assumable role that has a permission definition we don't maintain. And this is happening in a pipeline that will cease to exist with any meaningful scale of extensions and users. There is actually nothing to review that we are likely to successfully recognize as an issue from the results of the plan output. The functional tests we create are the effective means of finding bugs in these AWS maintained roles should they exist. Including a plan step in this situation can't provide the normally expected value of the step. If you do want it anyways, an actual rationale could be that while not 100% of the situations where terraform is used can the plan/approve step actually be effective, nonetheless the risk that someone might incorrectly assess the situation leads us to adopt the engineering standard that the plan step always occurs and we are willing to accept the increased cost overhead that will go along with that as it is likely to be low.

As you can see in the pipeline outline, since two of the extensions we are going to install will need permission to interact with the AWS API, we will need two oidc-assumable roles. One for cert-manager and one for external-dns. We will need to create the roles before installing the extensions. The AWS Terraform module for IAM has roles for these services built in.

**Listing 8.7 Provisioning oidc assumable roles for certi-manager and external-dns**

```
# service-account-role-cert-manager.tf
module "cert_manager_irsa_role" {
  source  = "terraform-aws-modules/iam/aws//modules/iam-role-for-service-accounts-eks"
  version = "~> 5.52.0"
  role_path          = "/PlatformRoles/"
  role_name          = "${var.cluster_name}-cert-manager-sa"
  attach_cert_manager_policy = true
  oidc_providers = {
    main = {
      provider_arn = data.aws_iam_openid_connect_provider.eks.arn
      namespace_service_accounts = ["cert-manager:cert-manager"]
    }
  }
}

# service-account-role-external-dns.tf
module "external_dns_irsa_role" {
  source  = "terraform-aws-modules/iam/aws//modules/iam-role-for-service-accounts-eks"
  version = "~> 5.52.0"
  role_path          = "/PlatformRoles/"
  role_name          = "${var.cluster_name}-external-dns-sa"
  attach_external_dns_policy = true
  oidc_providers = {
    main = {
      provider_arn = data.aws_iam_openid_connect_provider.eks.arn
      namespace_service_accounts = ["istio-system:external-dns"]
    }
  }
}
```

**Install istio:** We are going to start with the in-place install and upgrade process for Istio using the `istioctl` cli. This is a good starting point if you are new to Istio and the platform is being created as a greenfield build where there will be a decent period of only alpha customer usage as in our petech example. It is easier to manage and gives the platform engineering team some time to get familiar with all of the upgrade mechanics. The platform team can then refactor the Istio upgrade to follow a canary (or revision) upgrade process as the recommended practice, whether with the cli or using the Helm chart. In-place upgrades will result in a momentary service interruption and should be done during a maintenance window.

When using the CLI to do the install we will directly provision the `istio-system` namespace. But even if we were starting out with a Helm install that could take care of that for us, we would still want to create a namespace for cluster maintainers to perform various Istio-specific tests. We can deploy those within the `istio-install` script.

**Listing 8.8 Example Istio install script for cli-managed deployment**

```
# install_istio.sh
#!/usr/bin/env bash
set -eo pipefail
cluster_name=$1
istio_version=$(jq -er .istio_version $cluster_name.auto.tfvars.json)
echo "istio version $istio_version"
kubectl apply -f tpl/istio-namespaces.yaml
curl -L https://istio.io/downloadIstio | ISTIO_VERSION=$istio_version sh -
already_installed=$(kubectl get po --all-namespaces)
if [[ $already_installed == *"istiod"* ]]; then
    echo "inplace upgrade"
    istio-${istio_version}/bin/istioctl upgrade -y -f istio/values-$istio_version.yaml
    sleep 30
    kubectl get deployments --all-namespaces --field-selector=metadata.namespace!=kube-
system,metadata.namespace!=cert-manager,metadata.namespace!=istio-system | tail +2 |
awk '{ cmd=sprintf("kubectl rollout restart deployment -n %s %s", $1, $2) ; system(cmd) }'
else
    echo "new install"
    istio-${istio_version}/bin/istioctl install -y -f istio/values-$istio_version.yaml
fi
```

**Listing 8.9 Create istio-system test-fixture namespaces**

```
# tpl/istio-namespaces.yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: istio-system
---
apiVersion: v1                                     #A
kind: Namespace
Metadata:
  name: default-mtls
  labels:
    istio-injection: enabled
    pod-security.kubernetes.io/warn: restricted
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/warn: baseline
    pod-security.kubernetes.io/audit: baseline
    pod-security.kubernetes.io/enforce: baseline
---
apiVersion: v1                                     #B
kind: ResourceQuota
metadata:
  name: default-mtls-ns-quota
  namespace: default-mtls
spec:
  hard:
    requests.cpu: "2"
    requests.memory: 2Gi
    limits.cpu: "10"
    limits.memory: 20Gi
```

**#A** The configuration of our test-fixture namespace should be the same as we create for development teams on the cluster.

**#B** We will be using ResourceQuotas for platform customers so we want them on our testing namespace as well.

Perform a simple check to see if Istio is already running in order to determine whether to perform a new install or an upgrade. Also, to fully pick up the version changes, services using the service mesh need to be restarted. Here we trigger this restart directly but there are a number of strategies or processes that can be used to make sure all the services running on a cluster get rescheduled. Given that our Karpenter node pool configuration will replace all the nodes regularly we could even rely on that process for this particular situation.

Similar to the values.yaml in a Helm install, we can provide deployment setting values Istio CLI install. We will need version specific values files for each upgrade, both to support easy rollback and also in anticipation of moving to a canary upgrade process very soon.

**Listing 8.10 Manifest parameters for Istio cli install**

```
# istio-values/values-1.24.2.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  profile: default
  hub: docker.io/istio
  tag: 1.24.2
  namespace: istio-system
  components:
    Base: #A
      enabled: true
    pilot:
      enabled: true
    k8s:
      resources:
        requests:
          cpu: 10m
          memory: 128Mi
        limits:
          cpu: 2000m
          memory: 2024Mi
      nodeSelector: #B
        nodegroup: management-arm-rkt-mng
      tolerations:
        - key: "dedicated"
          operator: "Equal"
          value: "management"
          effect: "NoSchedule"
    podDisruptionBudget: #C
      maxUnavailable: 1
  Cni:
    enabled: true #D
  ztunnel:
    enabled: false #E
  istiodRemote:
    enabled: false #F
  egressGateways: #G
```

```

    - enabled: false
      name: istio-egressgateway
  ingressGateways:                                     #H
    - enabled: true
      name: istio-ingressgateway
  k8s:
    resources:
      requests:
        cpu: 10m
        memory: 128Mi
      limits:
        cpu: 2000m
        memory: 2024Mi
    nodeSelector:
      nodegroup: management-arm-rkt-mng
    tolerations:
      - key: "dedicated"
        operator: "Equal"
        value: "management"
        effect: "NoSchedule"
    podDisruptionBudget:
      maxUnavailable: 1
  values:
    base:
      enableCRDTemplates: false                         #I
#A The base and pilot components are the bases of the Istio control plane.
#B Like the rest of our management applications, we want the Istio control plane to run on the management node group.
#C We want to be sure to include a basic podDisruptionBudget so that istio can reschedule effectively as nodes in the management node group are refreshed.
#D Run the CNI service so that elevated privileges aren't required for normal deployments.
#E We will have Istio use side-car proxies. The ztunnel capabilities are for a per-node proxy server.
#F We do not want the installed Istio instance to be managed by a remote Istio instance.
#G An egress-gateway allows us to manage outbound traffic. We will not do that to start.
#H We do want an ingressgateway. There are various way this can be configured depending on the type of load balancer we want to use. For epeitech, a classic loadbalancer would work fine so we can start by using Istio's default method.
#I For an istioctl install will we turn off the default Helm behavior for CRDs.

```

## CERT-MANAGER

The cert-manager helm install is very much like the install scripts we created for the services pipeline. The values.yaml parameters will need our management node group settings, along with a podDisruptionBudget. Also, the service depends on the oidc-assumable iam role we created so we need to include this service account annotation setting in the helm command.

**Listing 8.11 Including OIDC Assumable Role in cert-manager Helm install**

```
helm upgrade --install cert-manager jetstack/cert-manager \
    --version v$chart_version \
    --namespace cert-manager --create-namespace \
    --set serviceAccount.annotations."eks\.amazonaws\.com/role-
arn"=arn:aws:iam::${AWS_ACCOUNT_ID}:role/PlatformRoles/${cluster_name}-cert-manager-sa \
    --values cert-manager/${cluster_name}-values.yaml
```

**EXTERNAL-DNS**

The external-dns deployment has some additional configuration requirements. While running, external-dns will watch certain kubernetes resources, the resources that relate to ingress domain name definitions, for the domain names we want it to manage. If a resource of the configured type is deployed that includes a domain name configuration then external-dns will update related DNS records as needed. To start, we will configure external-dns to watch the default Kubernetes resources along with the istio resources we will use with the platform-managed domain name. We also need to set our provider as AWS and indicate that we are using a public zone type. Using sync as our policy setting means that external-dns will keep Route53 in sync with whatever is running our cluster, including deleting DNS records if we delete the definition in the cluster. These settings are added to the values.yaml of the external-dns helm install.

**Listing 8.12 external-dns**

```

image:
  repository: registry.k8s.io/external-dns/external-dns
  pullPolicy: Always
serviceAccount:
  create: true
rbac:
  create: true
deploymentStrategy:
  type: Recreate
sources:
  - service
  - ingress
  - istio-gateway
  - istio-virtualservice
policy: sync
extraArgs:
  - --aws-zone-type=public
registry: txt
provider:
  name: aws
nodeSelector:
  nodegroup: management-arm-rkt-mng
tolerations:
  - key: "dedicated"
    operator: "Equal"
    value: "management"
    effect: "NoSchedule"

```

In addition to the values listed above, the Trivy scan results will point out some additional security requirements.

We also need to configure external-dns with the domain names that are associated with each cluster in our platform-managed domain pattern. We could hardcode these into the values.yaml directly much as we have done with the AWS provider reference. As time goes on however, these settings are likely to become part of a different implementation strategy around external-dns. We've seen it be more effective to provide these custom settings in our environment/values alongside our desired extension version info and other cluster specific environment settings. Recall that we defined a cluster specific subdomain as well as environment specific subdomains (such as dev, qa, preview, etc). The environment specific subdomains will have their own instance of external-dns since they tend to have their own lifecycle patterns. We can add the cluster specific values into our environments/cluster\_name.auto.tfvars.json.tpl file and pull them at pipeline runtime.

We delegated the subdomain hosted zone for our sbx cluster to the sbx account and that is the only subdomain. In the prod account we also have our prod domain so in our sbx, and prod values json we can add the following.

**Listing 8.13 Cluster environment values for host zones or subdomains**

```
For sbx
"cluster_domains": [
    "sbx-i01-aws-us-east-1.vitalsigns.io"
],
For prod
"cluster_domains": [
    "vitalsigns.io",
    "prod-i01-aws-us-east-2.vitalsigns.io"
],
```

Our install\_external\_dns.sh script could look like this.

**Listing 8.14 Install script for external-dns**

```

#!/usr/bin/env bash
set -eo pipefail
source bash-functions.sh
cluster_name=$1
chart_version=$(jq -er .external_dns_chart_version $cluster_name.auto.tfvars.json)
cluster_domains=$(jq -er .cluster_domains $cluster_name.auto.tfvars.json)
AWS_ACCOUNT_ID=$(jq -er .aws_account_id $cluster_name.auto.tfvars.json)
echo "external-dns chart version $chart_version"
declare -a domains=($(echo $cluster_domains | jq -r '.[]'))      #A
cat <<EOF > cluster-domains-values.yaml
domainFilters:
EOF
for domain in "${domains[@]}";
do
    echo " - $domain" >> cluster-domains-values.yaml
done
helm repo add external-dns https://kubernetes-sigs.github.io/external-dns/
helm repo update
trivyScan "external-dns/external-dns" "external-dns" "$chart_version" "external-dns-
values/$cluster_name-values.yaml"
helm upgrade --install external-dns external-dns/external-dns \
    --version v$chart_version \
    --namespace istio-system \
    --set serviceAccount.annotations."eks\.amazonaws\.com/role-
arn"=arn:aws:iam::${AWS_ACCOUNT_ID}:role/PlatformRoles/${cluster_name}-external-dns-sa \
    --set txtOwnerId=$cluster_name-vitalsigns \
    --values cluster-domains-values.yaml \
    --values external-dns-values/$cluster_name-values.yaml

```

#A We will create an additional values.yaml for the Helm install that includes the domains external-dns should watch.

Note that we included the service account annotation so that external-dns can use the IAM role we created, we set a txtOwnerId for dns in keeping with common practice, and we include the cluster-domain-values we pulled from the cluster environment values file.

## DEFAULT RESOURCES USING OUR INITIAL EXTENSIONS

When we described the components to be deployed in our extensions pipeline, we said that we could also include any default resources. For epetech, this means the resources that create our initial platform-managed subdomains. Each requires a gateway and certificate.

With all three extensions deployed, we can deploy the standard gateways and then test that everything is functioning as expected. What are the resource requests we need to make?

- We need a cert-manager certificate issuer created that can communicate with the AWS Route53 hosted zones for our domain.
- We need to have the certificate issuer generate a certificate that we can provide to a gateway, so that the gateway will support secure TLS (HTTPS) communication.
- And, we need the gateway resource for the respective domain.
- External-dns will automatically create the DNS records to direct traffic for the desired domain to the Istio managed load balancer.

With those items in place, if we deploy an application to the cluster we can include an Istio VirtualService resource that contains the path for our application. If everything is configured correctly, traffic on the defined path will hit the defined application service.

## CERTIFICATE ISSUER AND CERTIFICATE

We can use the free Let'sEncrypt certificate authority. We will need to add some additional configuration to our environments/ file.

```
"issuerEndpoint": "https://acme-v02.api.letsencrypt.org/directory",
"issuerEmail": "contact_person@vitalsigns.io"
```

With that we can generate a ClusterIssuer resource file to deploy to our cluster.

### **Listing 8.15 Create ClusterIssuer for each domain to be supported in the cluster**

```
# define_certificate_issuer.sh
#!/usr/bin/env bash
source bash-functions.sh
set -eo pipefail
cluster_name=$1
export aws_account_id=$(jq -er .aws_account_id "$cluster_name".auto.tfvars.json)      #A
export aws_assume_role=$(jq -er .aws_assume_role "$cluster_name".auto.tfvars.json)
export AWS_DEFAULT_REGION=$(jq -er .aws_region "$cluster_name".auto.tfvars.json)
export cluster_domains=$(jq -er .cluster_domains "$cluster_name".auto.tfvars.json)
export issuer_email=$(jq -er .issuerEmail "$cluster_name".auto.tfvars.json)
export issuer_endpoint=$(jq -er .issuerEndpoint "$cluster_name".auto.tfvars.json)
awsAssumeRole "${aws_account_id}" "${aws_assume_role}"                                #B
cat <<EOF > "${cluster_name}-cluster-issuer.yaml"                                #C
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-$cluster_name-issuer
spec:
  acme:
```

```

server: $issuer_endpoint
email: $issuer_email
privateKeySecretRef:
  name: letsencrypt-$cluster_name
solvers:
EOF
declare -a domains=($(echo $cluster_domains | jq -r '.[]'))          #D
for domain in "${domains[@]}";
do
  export zone_id=$(aws route53 list-hosted-zones-by-name | jq --arg name "$domain." -r
'.HostedZones | .[] | select(.Name=="\($name)") | .Id') #E
  cat <<EOF >> ${cluster_name}-cluster-issuer.yaml
  - selector:
    dnsZones:
    - "$domain"
  Dns01:
  route53:
    region: ${AWS_DEFAULT_REGION}
    hostedZoneID: ${zone_id}
EOF
done
cat "${cluster_name}-cluster-issuer.yaml"                                #F
kubectl apply -f "${cluster_name}-cluster-issuer.yaml"

```

#A As before, pull this configuration info from our environment configuration file.

#B We will need to query Route53 for our hosted zone IDs, so we will need to assume a role with permissions to do this.

#C First, we generate the main body of the CertificateIssuer resource

#D Then we loop through the list of initial default domains for which we will create gateways, adding the configuration for each to the Issuer file.

#E We use the AWS CLI to fetch the information about our hosted zones.

#F Output the contents of the ClusterIssuer resource file to the build log for debugging. This is not confidential information and being able to quickly check the log when debugging is helpful.

Now that we have defined an issuer for the cluster, we can request certificates.

#### **Listing 8.16 Create certificate for each domain gateway**

```

# deploy_gateways.sh
#!/usr/bin/env bash
set -eo pipefail
export cluster_name=$1
export cluster_domains=$(jq -er .cluster_domains "$cluster_name".auto.tfvars.json)
echo $cluster_name
echo $cluster_domains
declare -a domains=($(echo $cluster_domains | jq -r '.[]'))

```

```

for domain in "${domains[@]}";                                #A
do
    echo "create certificate for $domain"
    cat <<EOF > tpl/$domain-certificate.yaml
---
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
    name: $domain-certificate
    namespace: istio-system
spec:
    secretName: $domain-certificate
    issuerRef:
        name: "letsencrypt-${cluster_name}-issuer"
        kind: ClusterIssuer
    commonName: "*.$domain"                                #B
    dnsNames:
        - "$domain"
        - "*.$domain"
EOF
cat tpl/$domain-certificate.yaml
kubectl apply -f tpl/$domain-certificate.yaml
echo "define gateway for $domain"
export gateway=$( echo $domain | tr . - )
cat <<EOF > tpl/$domain-gateway.yaml                  #C
---
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
    name: $gateway-gateway
    namespace: istio-system
    labels:
        istio: istio-ingressgateway
spec:
    selector:
        app: istio-ingressgateway                                #D
    servers:
        - port:
            number: 80
            name: http-$domain
            protocol: HTTP
        hosts:
            - "$domain"
            - "*.$domain"

```

```

tls:
  httpsRedirect: true #E
- port:
    number: 443
    name: https-$domain
    protocol: HTTPS
  hosts:
    - "$domain"
    - "*.$domain"
  tls:
    mode: SIMPLE
    credentialName: "$domain-certificate"
EOF
cat tpl/$domain-gateway.yaml
kubectl apply -f tpl/$domain-gateway.yaml
done
sleep 360 #F

#A Loop through our list of default domains and create a certificate for each.
#B For each we will provision certificates that support the subdomain and a wildcard subdomain. Of course, including a wildcard is optional and will depend on your security risk profile.
#C For each domain we also create the gateway that uses the associated certificate.
#D We associate this gateway with the istio-ingressgateway which provisions and configures the external load balancer. External-dns, watching for ingress and gateway deployments, will update Route53 so that traffic on these defined domain names is directed to this load balancer and from there to the gateway connected service.
#E This gateway will accept traffic only from HTML ports 80 and 443. If someone tries to access the HTTP port, we want to automatically redirect to the HTTPS port.
#F Once all of our defined gateways are deployed, we will start testing. Completing the provisioning steps can take a couple minutes, particularly if this is the initial deployment of Istio. By waiting at the end of this script we give the system time to be ready before we start testing. An improvement to this wait strategy would be to put logic at the start of the test to poll and wait for the resources to report a Ready with a timeout period to report failure.

```

With the default gateways deployed, we can start testing. We can use awspec to confirm the IAM roles are ready and bats to confirm that all the extensions have a Running status.

But the principle test, which is a functional use of these extensions, will be to deploy an application that attaches to the cluster default gateway and see if we can access it over HTTPS. Httpbin is an excellent application for this kind of testing. Recall, we defined the default-mtls namespace for testing Istio managed applications. The application deployment will be the same regardless of the cluster being tested.

#### **Listing 8.17 Deploy application to test ingress health**

```

---
apiVersion: v1
kind: Service
metadata:

```

```

name: httpbin
namespace: default-mtls
labels:
  app: httpbin
spec:
  ports:
    - name: http
      port: 80
      targetPort: 8080
    selector:
      app: httpbin
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: httpbin
  namespace: default-mtls
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpbin
  namespace: default-mtls
spec:
  replicas: 1
  selector:
    matchLabels:
      app: httpbin
  template:
    metadata:
      labels:
        app: httpbin
    spec:
      nodeSelector:
        kubernetes.io/arch: amd64
      containers:
        - image: docker.io/kennethreitz/httpbin
          imagePullPolicy: Always
          name: httpbin
          command:
            - gunicorn
            - -b
            - 0.0.0.0:8080
            - httpbin:app

```

```

- -k
- gevent
ports:
- containerPort: 8080
securityContext:
allowPrivilegeEscalation: false
readOnlyRootFilesystem: false
runAsNonRoot: true
runAsUser: 65532
runAsGroup: 65532
seccompProfile:
type: RuntimeDefault
capabilities:
drop: ["ALL"]
resources:
limits:
cpu: 150m
memory: 256Mi
requests:
cpu: 100m
memory: 128Mi

```

**#A** the internal service definition accepts traffic on the regular HTTP port. We will receive traffic on HTTPS at the gateway and then terminate that certificate and once inside the service mesh, the mesh will generate the mtls certificates that encrypt the traffic from there.

**#B** Recall, we don't have a standing node group but instead Karpenter manages our default node space. We must target the desired existing node pool and, in this case, select the AMD architecture nodes.

**#C** httpbin doesn't have version tags. Trivy will alert us to this and we will either need to mirror the image to our own registry and version-tag there or add a .trivyignore instruction to allow using as is.

Now, to connect this to our gateway we need to include an istio virtualservice. This will need to include cluster specific info since we will use the cluster-specific gateway.

**Listing 8.18 Include a VirtualService resource in our test application deployment**

```
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: httpbin
  namespace: default-mtls
spec:
  hosts:
    - "httpbin.$cluster_name.vitalsigns.io"          #A
  gateways:
    - istio-system/$cluster_name-vitalsigns-io-gateway      #B
  http:
    - route:
        - destination:
            host: httpbin.default-mtls.svc.cluster.local
            port:
              number: 80
```

#A Here we define the domain name and path we want traffic to our httpbin service to use. We are making use of the wildcard certificate to receive traffic on a specific subdomain of the gateway.

#B And we provide the path to the gateway that has been configured to receive this subdomain name.

If cert-manager is correctly provisioning certificates for our gateways, and if external-dns is correctly making DNS entries to direct traffic to these gateways, and if Istio is correctly deploying a load balancer and connecting VirtualServices to the defined gateways then once we have deployed httpbin we can run this command to confirm the end-to-end health of all of the extensions.

**Listing 8.19 Test application endpoint**

```
jsonResponse=$(curl -X GET "https://httpbin.$cluster_name.vitalsigns.io/json" -H
"accept: application/json")
A healthy response will be:
response {
  "slideshow": {
    "author": "Yours Truly",
    "date": "date of publication",
    "slides": [
      {
        "title": "Wake up to WonderWidgets!",
        "type": "all"
      },
      {
        "items": [
          "Why <em>WonderWidgets</em> are great",
          "Who <em>buys</em> WonderWidgets"
        ],
        "title": "Overview",
        "type": "all"
      }
    ],
    "title": "Sample Slide Show"
  }
}
```

Think about how you could orchestrate this test in a bash script.

**EXERCISE 8.4: CREATE A CONTROL PLANE EXTENSIONS PIPELINE THAT DEPLOYS SERVICE MESH AND INITIAL INGRESS PATTERN**

Complete the above pipeline and create a development build and release pipeline for our control plane extensions. As with the other pipelines, be sure to include a nightly, recurring job that runs the integration tests.

### 8.2.3 Using Operators for Persistent Data Platform Capabilities

Over the last couple of years, the maturity in Operator SDKs and cloud vendor infrastructure component operators means we can effectively extend the Kubernetes API to provide a platform experience for using many of the most commonly needed persistent data resources for platform users. Here is a good point to reiterate that an engineering platform is not a wrapper for all infrastructure that may ever be needed by a development team. In terms of cloud infrastructure, an engineering platform provides the commonly needed resources for development teams building distributed service architecture applications. And while there are a lot of capabilities within that definition it is far short of *everything*.

The things that API teams tend to need are databases, message queues, file shares, and high-performance caches. These all tend to be about holding data, hence we refer to them as *persistent data* resources.

If we can put the provisioning and use of these kind of resources behind an API, then we can create the right experience for both the users and the team delivering the capabilities. This is where infrastructure operators come in. There are different kinds of options in this category. For more directly management there are tools like Crossplane[\[16\]](#), AWS ACK[\[17\]](#), and the operator SDK[\[18\]](#). The Kubernetes API now becomes the interface for developer provisioning these resource, but, like the platform control plane itself, the platform engineers are responsible for the operational lifecycle. Maintaining infrastructure through operators has differences from the traditional IaC lifecycle and it has its own learning curve. Yet many large organizations have had great success in using operators to extend the acceleration of a platform experience to many more categories of infrastructure.

If you don't have the staff or the resources to build the new operational skills of fully operator managed infrastructure, Terraform also has an offering in this category with the Terraform Operator[\[19\]](#). This will basically wrap the entire Terraform workflow and operational process within the operator pattern so it brings its own sort of complexity and performance implications.

Using the operator pattern users of your platform can provision and manage common needed persistent data infrastructure from within their Helm chart. A database becomes just another kubernetes resource within the application deployment definition. Where credentials or roles are needed for interaction with the infrastructure, the operator automates managing these on the developer's behalf while providing debugging access through their regular platform identity and thereby maintaining the separation.

An implementation example is beyond what we want to accomplish in this book. But in our experience, where the number of developers using an engineering platform starts to go beyond 50-100, the other IaC approaches to managing these particular resources will create a noticeable drag on the overall platform experience.

The operational differences between the Operator pattern and traditional IaC should not be underestimated. We don't recommend employing operators as an alternative to traditional IaC languages in any general sense until you have had substantial experience with the pattern on a smaller scale. Even in large scale engineering platforms we would expect to see no more than a dozen or so common components made available using this method.

## **INFRASTRUCTURE STARTER KITS**

You are probably familiar with the pattern of providing infrastructure resources to developers in the form of Terraform starter kits—Sample repositories that can be duplicated and then used to provision some piece of infrastructure. Developers are provided with a fully configured bundle of code that includes the needed terraform files with sensible defaults, hopefully a pipeline and even the observability configuration, and more. All the things that a typical DevOps team might include in an infrastructure pipeline.

This can be a useful strategy. Up to a point, this can increase the efficiency of a DevOps *team* strategy as they try to support a greater number of developers.

This isn't really a platform engineering strategy. It feels a lot like self-serve. Developers can independently get a copy of the code and in short order have the infrastructure up and running. This is exactly what we want from language starter kits. A python API starter kit can contain the API framework, logging and tracing and any other language libraries, the complete dev to production pipeline, initial operational observability, and generally everything needed for the developer to almost immediately begin coding business logic that can flow to production equally rapidly. So, what's the difference?

In the case of the Python starter kit, the developer assumes full responsibility. It is rare within a company for the *User* of a database starter kit to assume much responsibility at all. All the real responsibility is still with the DevOps team that created the starter kit. Any operational issues at all and they are called back to own both the response and the follow through. And most of the time, when there are changes within the definition of the starter kit, anything that needs to be pushed out to users of the kit will also end up being the responsibility of the DevOps team.

Recall, an essential part of the product architecture of a platform is that the features of a platform are delivered as a service interface first—an API. What a platform engineer team is responsible for is provided to the user behind an API.

This doesn't mean that there is no place for infrastructure starter kits. There absolutely is. There are also a couple essential characteristics to keep in mind as you use them and as you think about their relationship to engineering platforms.

### **Know the Value and the Cost**

Infrastructure Starter Kits will not enable the development acceleration or economies of scale possible through an engineering platform. This isn't a criticism.

At basically any scale, these kits save a *devops* team time when creating new resources and supporting many of the every-day issues. But time and scale will require greater investment. As the versions that must be supported grow and the kits themselves undergo change, the friction of this overhead starts to create it's own overhead. And as more and more instances of infrastructure created with starter kits exist, the amount of *devops* engineers needed to support will grow. There are moments over the lifecycle of organizational adoption of infrastructure starter kits where the accelerating effect will be more measurable. But if you look at the overall organizational impact across longer periods and time and widespread adoption, infrastructure starter kits are better treated as a way to improve and sustain quality rather than acceleration or cost savings. Which is not a small thing. It shouldn't be ignored.

## **KEEP SEPARATE FROM THE PRINCIPLE ENGINEERING PLATFORM DELIVERY TEAMS**

We also strongly recommend that if Terraform starter kit style resources will be curated as part of your internal developer resources that the engineering platform product teams should never be the team maintaining them. Or at the very least, if such resources are to be part of the overall platform product roadmap, that a separate subdomain product team have ownership within the overall platform. And this team's only responsibility should be the curating and supporting of these particular accelerators. The reason for this is that fundamentally these are not self-serve resources. Inevitably the maintaining team will be required to provide a material level of traditional *devops* style support. They are a service team which will prevent them from being able to effectively deliver against a product backlog.

### **8.3 Platform Management APIs**

We have the foundations of our engineering platform at this point. But as is, only the platform administrators have access. How do we enable access to platform features for our internal platform customers?

We want the development "team" to be the primary authorization mechanism for our platform and we want the users of our platform to be able to self-manage team onboarding and membership. This means that developers should not need to request access to all the various systems that make up the engineering platform capabilities. And there will be dozens of such systems. It is quite common, within organizations of all sizes, for a developer to have to request access to every developer resource individually. In large companies, developers are constantly being hired or quitting, moving between teams, or joining multiple teams, spending weeks to get access to systems they need to do their job. Part of the function of an engineering platform is to own this experience. When a Team is onboarded, every member of the team should automatically have access to all the features of the platform within the bounds of their team.

In our vitalsigns platform we have used Auth0 to create an identity integration between GitHub and the control plane. We could just expand the use of Auth0 to include all the access points. Every organization has some central source of authentication, whether that's Active Directory, Google Identity, or something else. We could connect Auth0 to our corporate authentication system and then in turn integrate all of our SaaS tools with Auth0. Auth0 is an Okta product, and many companies use the flagship Okta product for this purpose. But authentication is the easy part. The challenge is managing authorization. What will be the source of truth for teams and team membership within our platform product? No matter how we solve this, it is impossible that all the systems we use will have a built-in integration perfectly suited to our solution.

We have started out by using Github teams as the mechanism for tracking team membership. Using our OIDC integration, a user would authenticate through Github and their authorization comes from the team of which they are a member.

If you completed the OIDC integration and CLI projects then you can already authenticate and access the control plane using your personal Github credentials. Recall that in the control plane base pipeline we also applied a ClusterRoleBinding for the github team in which we, as the platform building team, are members. We could have integrated GitHub with the vitalsigns corporate identity system. If that were Active Directory for example, we could configure the integration such that AD groups are synced as Github groups and then AD would be the place where teams needed to be formed and people added and removed from teams. How would we provide a self-serve experience there for teams to self-manage?

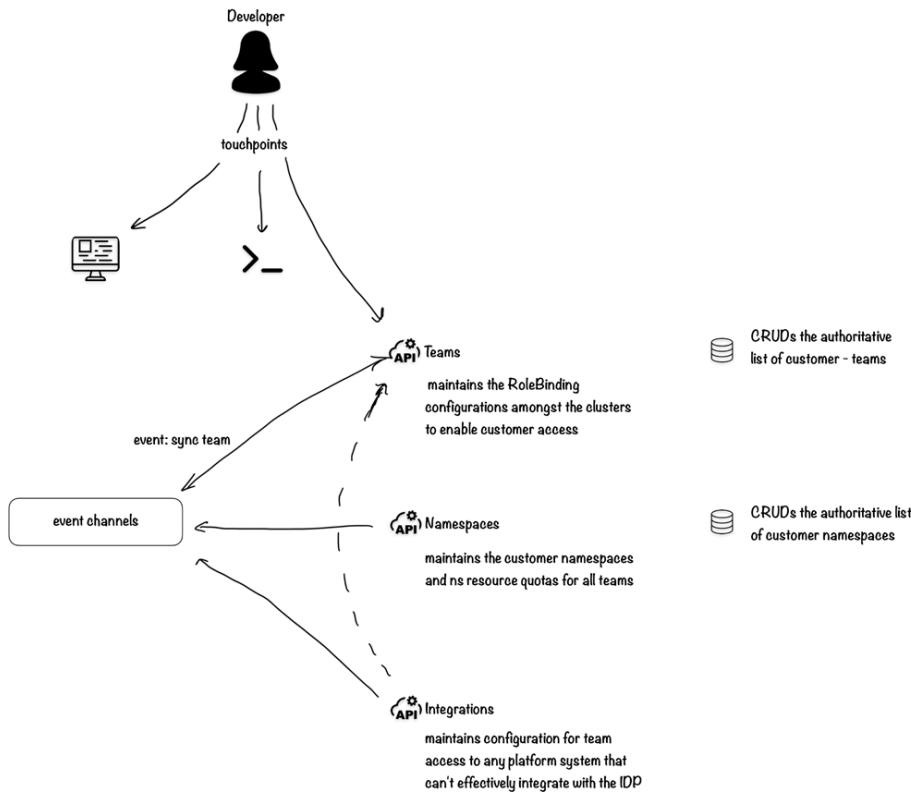
Whether we use Github as the authoritative source for team membership or something else, we still need a means of knowing which "teams" are teams that are engineering platform customers.

If we had such a list, we could then automate the process of:

- Creating the standard team namespaces in the appropriate clusters  
Example: when the Payments team is onboarded these ns are created:
  - payments-dev, payments-qa in the nonprod cluster
  - payments-preview in the preview cluster
  - payments-prod in the prod cluster
- Creating the RoleBindings in each cluster that authorize members of a team to access their namespace
- Creating the matching gateways for the standard namespaces
- Syncing Teams and team membership information, as needed, with any other system that we integrate with our platform that doesn't have a built-in integration with our authorization source.

This is just the basic set of things we could do. Many of the systems we will want to use will have easily integrated access controls, but never all of them. And we need a mechanism that gives us confidence that the platform experience can be maintained no matter how platform features evolve. Far too frequently organizations just decide they can't be bothered to solve this problem and the waste amounts to thousands of hours a year.

An effective, and sustainable, strategy is to build a handful of lightweight custom platform management APIs to manage any integration needs that the tools or technologies we use don't already support. Our experience is that primarily these APIs are concerned with control plane configuration, but they also form a critical role in enabling an organization to broaden its ability to have multiple technical teams outside the primary engineering platform team easily integrate with the platform roadmap requirements.



**Figure 8.8** A “teams” API can be the interface to onboard a team and maintain the authoritative database for customer teams. This API will maintain the correct RoleBindings on the respective clusters. It can also populate ‘sync team’ events stream to which other services can subscribe.

In this way, a namespace API could maintain the standard namespaces across all the clusters for a team and also be the interface to provide support for teams creating and managing custom namespaces. An integrations API can subscribe and be used to maintain any configurations related to teams authorization within a tool or system that doesn't easily provide the same experience through our regular IDP. The teams API can also generate these sync events on a frequent recurring basis so that the configuration settings become resilient and able to self-heal from unintended external changes. You can see how the introduction of the event architecture provides an easy integration point for teams outside the primary platform team. For example, if there was a separate team that managed an Enterprise secret store (like Hashi Vault), they could support the platform roadmap directly by creating a service that subscribed to the Teams events and then performed any necessary configuration within the store to provide that part of the comprehensive platform user experience.

It does require an investment to create and maintain these services, but the time saved over the life of an engineering platform through being able to resiliently maintain a unified platform user access experience more than makes up for the cost. The larger the scale of the organization the more return there is on this kind of investment.

### 8.3.1 Managing Teams and Namespaces for Early Adopters

How do we deal with these issues in the early stages of engineering platform development?

Already in our vitalsigns control plane we could have a couple early-adopter teams get started deploying their applications and provide feedback if we were to add the necessary configuration to the clusters. While the management APIs are not especially complicated, it will still take some amount of time before those can automate this step.

A straightforward approach would be to create a dedicated pipeline that uses something as basic as a JSON file containing our initial teams and standard namespaces. This simple customer management pipeline would follow the same release pattern as our other cluster configuration pipelines. We will not be supporting more than a couple teams and standard namespaces in this alpha stage so we should rarely need to make changes to this configuration until the management API is ready.

#### **Listing 8.20 ClusterRoleBinding for Individual Development Team Access**

```
---
```

```
Insert role binding here
```

### **EXERCISE 8.5: APPLY ROLE BINDING FILES FOR A DEMO TEAM TO OUR PRODUCTION CLUSTER**

Imagine one of our alpha teams is called the Payments team. Using the example from our default-mtls namespace, create three namespaces in our example production cluster; payments-dev, payments-qa, and payments to represent the three environments of this demo teams path to production. Obviously, in a real setting these would not all be in the same cluster. Using the example from listing 8.20 create the role bindings that would be needed for the Payments team to be able to access their namespaces. Then, create a matching Payments team in our github organization. Add yourself to this payments team and remove yourself from the platform admin team. Now using the cli, authenticate and attempt to list the contents of these three team namespaces.

## **8.4 Summary**

- Having multiple development and testing environments that are not used by the platform's developer/users is critical for delivering non-disruptive change.
- Include a Preview environment where platform customers do deploy their code but that is not in their path to production.
- This significantly increases the likelihood of discovering breaking changes before they reach the general developer-facing environments.
- Control plane services collect, analyze, return or forward data about the state and activities of the cluster or applications running on the cluster
- Control plane extensions enable platform users to provision and configure resources, whether inside or outside of the cluster, that are not normally part of the Kubernetes API - they *extend* the API.
- Observability services, cluster autoscaling, deployment support and security are the services that should be included or assessed as part of the minimum required services for general adoption of an engineering platform
- Storage classes, Ingress or Gateway API, application specific resiliency, and a general authentication and authorization capability are the extensions that should be included or assessed as part of the minimum required extensions for general adoption of an engineering platform
- Maintain an architectural distinction between services and extensions
- When considering assigning the responsibility of an on-cluster capability to a team outside of the engineering platform product teams, assume that services will be the only effective candidates.
- Moving extensions to an external team carries significant risk of introducing friction and waste into the platform experience.
- Use of a service mesh to provide the Gateway API, dynamic routing, and traffic policy implementation is the most effective means of maintaining domains of change (supporting evolutionary architecture) and preserving a zero-trust networking strategy

- The pipeline testing strategy for deploying services and extensions should include both general status level checks and functional tests that actually use the service or extension
- As part of your platform adoption strategy, plan on having a period of time where only one or two alpha customers are onboarded to the platform to provide early feedback of the user experience, prior to the platform being considered at MVP or general availability.
- While use of a single, early adopter team to provide feedback during feature development is an effective strategy, only during this development phase should the feature have any non-self-serve process.
- General release of a platform can happen early, even with the MVP feature set, but from the first general release onward, all features must be fully self-serve.
- Providing developers a self-serve means of provisioning, configuring, and maintaining the routinely or commonly used infrastructure resources that reside outside the cluster, such as databases, file shares, message queues, API gateway integrations, etc, should be part of the platform roadmap but are not necessarily MVP.
- While providing developers with Terraform (or any IaC language) starter kits can be both accelerating and necessary at the Enterprise level in general, these resources should not be part of the principle engineering platform roadmap.
- A small set of lightweight, custom Platform management API are a necessary part of maintaining the engineering platform product experience.
- The platform management APIs are also the means of enabling effective participation in the platform roadmap by teams outside the platform product ownership

- [1] <https://github.com/sse-secure-systems/connaisseur>
- [2] <https://fluxcd.io>
- [3] <https://argo-cd.readthedocs.io/en/stable/>
- [4] <https://helm.sh>
- [5] <https://kustomize.io>
- [6] <https://www.vaultproject.io>
- [7] <https://external-secrets.io>
- [8] <https://kubernetes-sigs.github.io/metrics-server>
- [9] <https://kubernetes.io/docs/concepts/cluster-administration/kube-state-metrics>
- [10] <https://github.com/resmoio/kubernetes-event-exporter>
- [11] <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale>
- [12] <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough>

- [13] <https://gateway-api.sigs.k8s.io>
- [14] <https://istio.io>
- [15] <https://github.com/postmanlabs/httpbin>
- [16] <https://www.crossplane.io>
- [17] <https://github.com/aws-controllers-k8s/community>
- [18] <https://sdk.operatorframework.io>
- [19] <https://developer.hashicorp.com/terraform/cloud-docs/integrations/kubernetes>

# 9 Architecture Changes to Support Scale

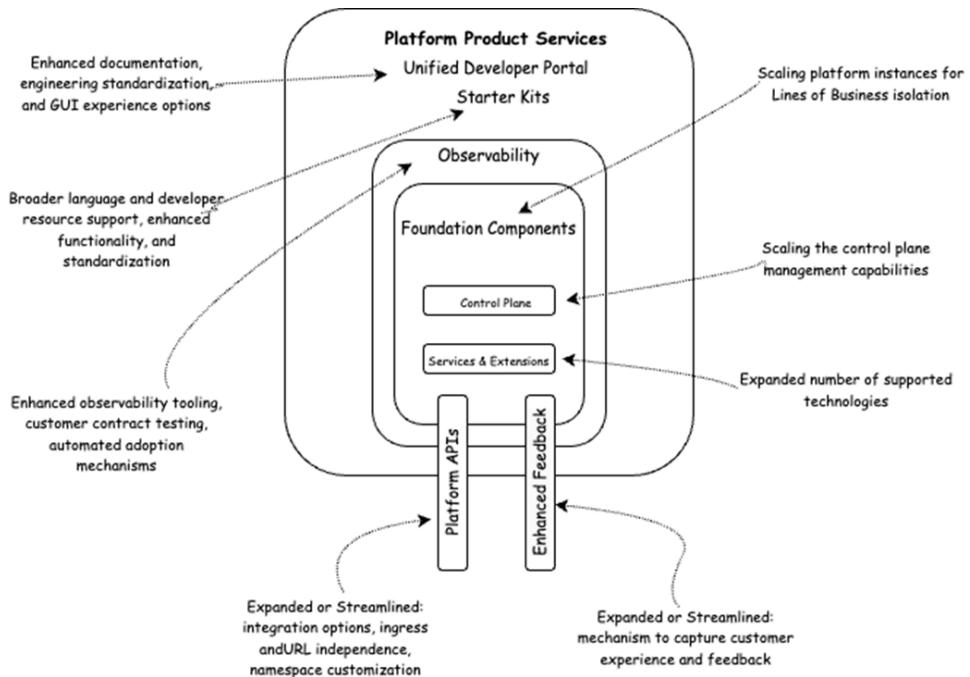
## This chapter covers

- Scaling the control plane roles
- Scaling the pipeline orchestration for *many* clusters
- Scaling the orchestration of control plane services and extensions
- Using events to increase the scale of automation

Scaling an engineering platform isn't just about adding more servers or improving performance numbers—it's about smart, intentional architectural changes that keep teams moving quickly without causing chaos. When the business needs to support more developers or deliver engineering platform capabilities faster, scaling the engineering platform product team is just as important as scaling the technology. We need the right people, roles, and processes in place to effectively expand the platform control plane as well as the services and extensions. And scaling isn't just about speed; it's about cost, too. What was efficient for a handful of teams can become brittle or expensive as the number of teams using the platform, or the number of services running on the platform, scales.

We may have started with a single engineering platform product team to establish a good architectural foundation, but we won't remain there, at least not at a medium or large business scale if there are dozens or hundreds of development teams. Multiple domain teams within a product mean domain boundaries. To be effective, these teams need to be sufficiently autonomous. In other words, the independent and self-serve experience we create for the users of our platform is the same experience we need between the teams within the platform as they create, deliver, and operate the platform. How will these teams work together, and how do we decide which boundaries to create? When you're scaling, event-driven platform releases and contract tests between platform teams and their consumers are game-changers for keeping things reliable and maintaining trust between teams. At the same time, managing a software-defined delivery architecture across a growing number of control planes, each with increasing capabilities, is all about finding that balance between our initial centralized strategy and the distributed patterns that perform better at scale rather than creating a one-size-fits-all solution. It's about creating an architecture that can adapt, making innovation and experimentation possible and affordable. In this chapter, we'll dig into a few of the key things to keep in mind so that scaling stays both sustainable and meaningful.

As shown in Figure 9.1, each domain within our platform product can have its own strategies for scaling.



**Figure 9.1 Practical examples of evolutionary scaling strategies for engineering platform components.**

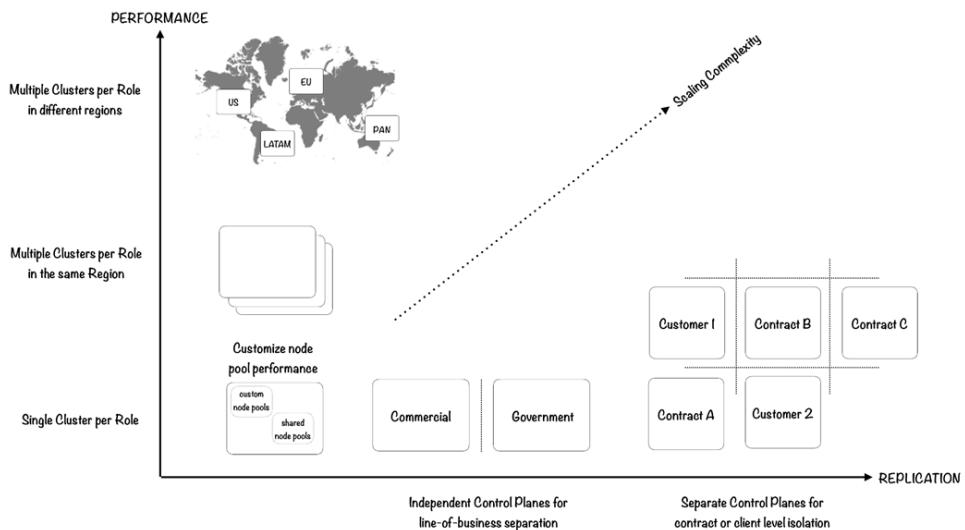
When to scale and which areas to scale should be driven by business priorities. For example, if our organization is in the financial services industry and needs to improve in both banking and investing software development equally, then we probably need to provide isolated replication of the engineering platform to support the strict regulations about separation sooner than we need support for greater scale of developers in general. On the other hand, if a top business priority were the ability to expose internal APIs to external, third-party developers, then we would probably prioritize the control plane extensions to enable developers to self-manage integration with our public API Gateway before giving developers a way to create custom namespaces.

In this chapter, we are going to cover some effective scaling strategies that commonly arise in the foundational components, which we started in chapters 6 through 8.

## 9.1 Scaling the Control Plane Roles

We've started with a single cluster instance for each control plane role. This structure can meet the needs of a large segment of business categories and sizes.

However, as organizations scale, the regulatory, contractual, or capacity and locality performance demands will mean we need to scale our engineering platform capabilities. Figure 9.2 demonstrates the control plane scaling that happens across the two axes.



**Figure 9.2 Control plane scaling happens along two general axes. Performance requirements for specific control plane roles will grow, requiring node pool customization capabilities or multiple clusters within a single role. Alternatively, legal requirements or contractual decisions can mean that we need multiple isolated instances of the same role.**

Having multiple clusters within a single role, whether within the same region or across different regions, will require changes to our ingress architecture. This impacts the control plane architecture in two ways. First, we will need another layer of load balancing before the normal cluster ingress point. This could be followed by re-balancing services among the clusters or a round-robin or lease-requests traffic routing for all clusters in the same region. For regionally distributed clusters, this usually means geo-routing capabilities. Second, the self-serve ingress capability we provide to developers now needs to support the ability to simultaneously deploy the same version of a service in multiple clusters, along with customizing traffic flows between clusters or among external technologies provided by the extensions.

The dynamic routing capabilities of a service mesh are well-suited to provide this experience.

When we need isolation between lines of business within the organization or at even higher granularity based on market demands of the customers the business serves, this will mean even greater scale in the number of clusters regardless of the individual load. This could mean complete duplication of the entire platform product definition. Even in this situation, we still recommend maintaining a single product roadmap rather than each line of business creating its own platform product definition.

## DYNAMIC RELEASE PIPELINE

Where a single, static orchestration pipeline is responsible for more than 20-30 clusters (and growing), the release pipeline will start to become brittle. Hardcoding, at the pipeline level, a large and effectively dynamic pool of clusters in each role, just doesn't scale well. An effective first step is to move to dynamic release pipelines.

At run time, a dynamic release pipeline will generate the workflow matching the details for all clusters currently defined within a role. This is somewhat similar to a matrix build in an application development pipeline.

Rather than a fixed list of clusters per role maintained directly within our pipeline code, a globally available data set is created that lists the current roles and clusters per role. The deploy and test steps for any part of a release pipeline are then generated prior to execution. If there are four clusters in four different regions in a role, our pipeline automation will generate the matching release pipeline using values from the global definition.

Our pipeline tool determines how we implement this. It is possible that the built-in matrix capabilities of some pipeline tools will be part of the implementation strategy. Regardless of the implementation details, the outcome will be that the deploy and test phases of our release pipeline are generated to match a global definition rather than hard-coded.

**EXERCISE 9.1 EXPERIMENT WITH GENERATING THE DEPLOY AND TEST STEPS OF A PIPELINE FROM A GLOBAL LIST.**

Imagine that in the epetech foundation pipelines we wanted the deploy and test stages of the pipelines to be dynamically created at pipeline runtime. The VPC pipeline is the first pipeline to have a one-to-one relationship with our control plane roles. We are just using a single sandbox and production cluster for exercise purposes, but let's assume we had a full set of roles that include dev and qa in our testing environments. This exercise is an experiment. You could work out a complete solution or just list the capabilities that would need to be in a solution.

If we created a globally available definition of our test roles, what information would we need to be able to successfully generate that part of our pipeline?

**Listing 9.1 Global roles definition file.**

```

test:
  filter: "*on-push-main"                                #A
  deploy:
    - platform-dev                                     #B
    - platform-qa
roles:
  platform-dev:
    deploy:
      - platform-dev-i01-aws-us-west-2
      - platform-dev-i01-aws-eu-west-1
  instances:
    platform-dev-i01-aws-us-west-2:                      #C
      aws_region: us-west-2
      aws_account_id: '10100000000'
    platform-dev-i01-aws-eu-west-1:
      aws_region: eu-west-1
      aws_account_id: '10100000000'
  platform-qa:
    deploy:
      - platform-qa-i01-aws-us-west-2
      - platform-qa-i01-aws-eu-west-1
  instances:
    platform-qa-i01-aws-us-west-2:                       #D
      aws_region: us-west-2
      aws_account_id: '10100000000'
    platform-qa-i01-aws-eu-west-1:
      aws_region: eu-west-1
      aws_account_id: '10100000000'

```

**#A** We will likely need a filter value that indicates how the deploy and test steps are triggered. CI and test environment deployments are triggered by a git push, whereas the release pipeline is based on a git tag.

**#B** We might include a list of all the roles we will want to deploy through the push-triggered pipeline.

**#C** For each role, we will need to provide some values.

**#D** We will need some way to specify the clusters that are part of the role.

**#E** And it's likely we will need to be able to provide cluster-specific values for each cluster in a role.

Looking at our existing VPC pipeline, everything up to the sbx deployment would be the same no matter how many clusters we were managing. Ignore the nightly testing for the moment. How could we use the CircleCI generated pipeline capability[\[1\]](#) to generate a deploy and test pipeline for our testing environments after the static analysis portion is complete?

Generating a pipeline will require code, but depending on the amount of flexibility needed and how broadly we want to use the automation, a solution could be as simple as a bash script that populates a template or a more feature-rich CLI. You can look at one CircleCI solution example here[\[2\]](#).

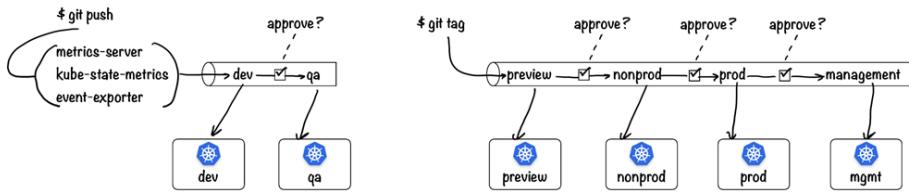
If the pipeline tool we are using has no mechanism to support dynamic pipelines, this limitation should be addressed prior to scaling our clusters beyond the limits of a static pipeline. That might mean using a different tool, or it could mean stepping up to even more scalable cluster provisioning strategies. Imagine a cluster-engine made up of a dedicated EKS instance that had elements of crossplane running sufficient to support provisioning the equivalent of our control-plane-base configuration. Now our pipeline could have a step that dynamically generated the set of Customer Resource Requests the matched each cluster within a role rather than individual pipeline steps.

Even this strategy has limitations. While it works well for ongoing scale in the number of supported regions within a role, it is not necessarily suited to other scale needs. Looking back at figure 9.2, if our primary scaling need for cluster roles is *customer* or *contract* isolation, this is likely to mean that it will be the production role that scales far more than the other roles. And in that situation, the demand for those additional clusters is originating within the development teams using the platform and therefore we will need to provide a self-service experience for those teams to trigger new control plane instances that are nonetheless still managed through our general cluster health processes.

We are not saying that a low scale technique will simply be unable to create higher scale. It is just that there will be consequences. If you stick with a static pipeline while your regional footprint keeps growing, sooner or later, operational and evolutionary changes to your control plane and pipeline will take longer and become more error-prone, even if you catch issues early. A simpler, low-complexity approach might be much cheaper to setup, but if at scale it makes routine operations five times slower, then you've lost whatever gain you hoped to achieve.

## 9.2 Scaling the Orchestration of Control Plane Services and Extensions

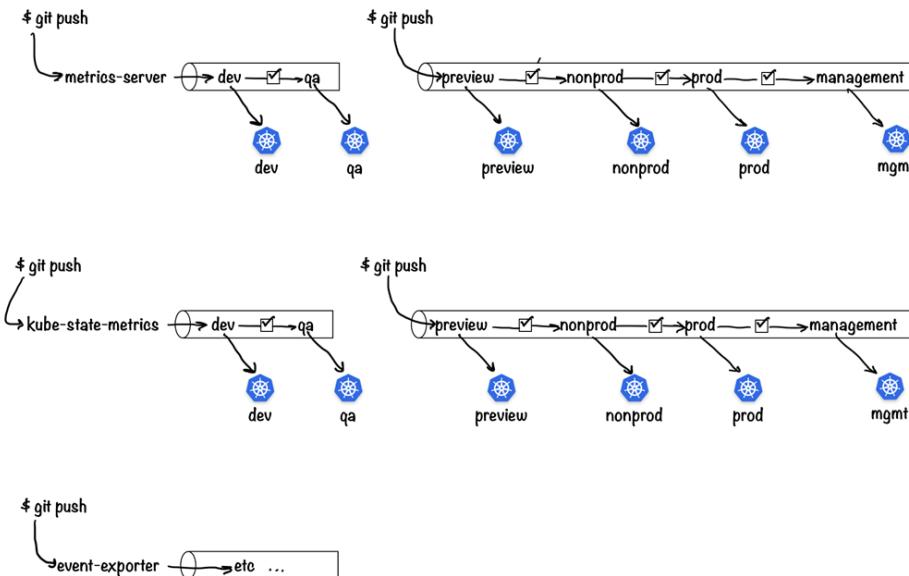
At epetech, we started out using straight-forward deploy and test pipelines to manage services and extensions as shown in figure 9.3.



**Figure 9.3 At first, we deploy and test all our Services through a single pipeline. We do the same with extensions.**

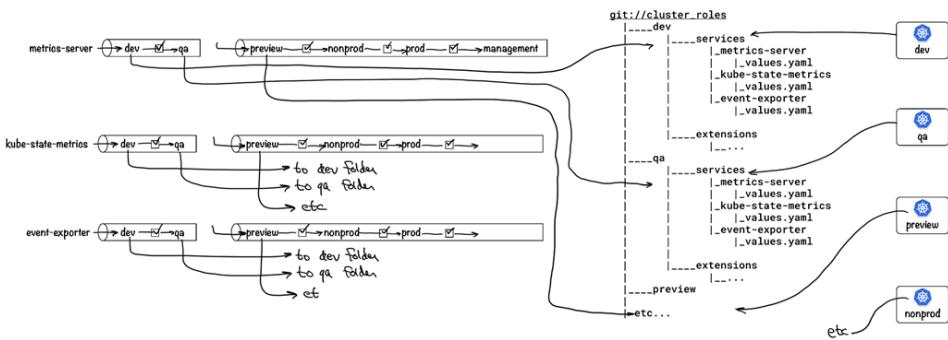
While this worked well when there were just a handful of clusters, this approach is showing cracks as the number of clusters grows. Longer deployment times, pipeline bottlenecks, and challenges with performing routine upgrades on more than one service or extension at a time make it clear we need a better strategy.

We can solve the challenges as shown in Figure 9.4, with multiple simultaneous changes by breaking out each service into its own lifecycle pipeline.



**Figure 9.4 Dedicated software lifecycle pipelines for each service and extension is one of the first modifications we can make to improve the health of our normal operational activities as the scale grows. While necessary, by itself, this will actually aggravate the issues that the scale has created.**

When each pipeline has to reach out to each cluster to manage the service, as the number of clusters grows, managing these pipelines becomes a growing problem as well. As in section 9.1, dynamically generating the release portion of the pipeline could improve the pipeline management challenges. However, at the individual service or extension level, the scale of the problem is growing faster than just the number of clusters. For our control-plane-base pipeline, adding a region would mean adding three more clusters (one each for preview, nonprod, and prod). But with three services, as we have at the moment, that means three *times* three more service deployments to manage. There will be more than just three services in a real-world setting. What we need is a way to limit the scope within an individual service pipeline to just the number of roles we define in our platform without needing to modify these software lifecycle pipelines as shown in Figure 9.5, every time clusters are added or removed.



**Figure 9.5 What if we modify our pipeline to make release configuration changes to a code repository rather than actually performing the deployment? Further, we then deploy a service on each cluster configured to watch for those changes and perform the deployment locally.**

With effective use of our observability tooling we can easily aggregate all deployment metrics and test results into a single view and a single definition for monitoring. A new service or extension can be added without needing to directly change cluster configuration and likewise, new clusters can be provisioned and the local deployer configured to integrate with the appropriate role without needing to change any individual service or extension pipeline.

A centralized datastore for observability data is a requirement for this distributed management architecture. But that is already a core requirement for an engineering platform in general.

By making all three of these changes:

- Individual lifecycle pipelines for each service or extension that make updates to a central configuration definition for each cluster role.
- Clusters individually pull their configuration from the central definition and perform the deployment and health tests locally.
- Central visibility for the status, monitoring, and alerting of this distributed architecture built on top of our observability tools, deployed by the service, extension, or distributed deployer lifecycle pipelines.

We can now effectively scale both the number of services and extensions being managed and the number of clusters where these are deployed without a proportionate increase in operational time. By moving the deploy activity out of a centralized location, and distributing the workload to the cluster where the deployment is occurring, the cumulative deployment event is much quicker. The cross-cluster impact from individual deployment issues is reduced. And, we have the ability to create a loose boundary between the lifecycle of a cluster and the administrative services that run on it.

You will no doubt recognize the pattern of the deployment engine running independent of the CI/CD pipeline though triggered by the same events. Within IT or DevOps circles, GitOps is the popular term now associated with what is a long-established software delivery approach. ArgoCD<sup>[3]</sup> and Flux<sup>[4]</sup> are the most popular deployers for Kubernetes. As part of a scaled cluster-level configuration, we recommend running this service without all the GUI and other overhead. Flux has consistently been a very small and fast agent and has the added benefit of having built-in support for defining cluster-specific values to enable a shared role deployment configuration to also easily incorporate cluster-specific values from the cluster at deploy time. But with some added setup you can do the same with Argo, and Argo has recently introduced a **core** version that is just the deploy engine.

### 9.3 Scale through Platform Event Streaming

In section 9.2, the distributed cluster configuration process we describe makes use of an important *scaling* architecture pattern. We deploy a service to each cluster that is configured to watch for changes in a specific git repository and perform a Helm upgrade in response. This is a simple example of an event-driven architecture. GitHub provides an event-stream publisher in the form of a web-hook. You can configure a subscriber application to listen for any of the events published by the web-hook. Tools like Flux and ArgoCD have a subscriber capability for watching git repositories and can be configured to respond to a variety of common git events, such as pushing a change or applying a tag.

Event-driven architectures play a critical role in scaling our engineering platform in much the same way they do for distributed services software in general and can afford the same type of benefits:

- Decoupling services: A service only needs to know about the events it is interested in, allowing for independent development and deployment.
- Asynchronous scalability: By processing events asynchronously, a system can more easily scale to handle high volumes of data by distributing processing across multiple services.
- Resilience: If one service fails, other services can still operate as long as the event stream remains functional. It is easier to configure a graceful response to a dependent service failure amongst loosely coupled services.
- Agility: New capabilities can be added or modified without significantly impacting existing ones by simply publishing or subscribing to relevant events.
- Real-time updates: Enables near real-time data updates across a distributed system by enabling services to respond to events as they occur.

- Simplified integration: Facilitates seamless integration with external systems by publishing and consuming events from any source.

As the scale of developers on an engineering platform grows, more of the scaling and coordinating strategies used in general software architecture become valuable and important considerations in our platform architecture.

The strategies we discuss in section 9.2 are effective for scaling the performance of cluster configuration. But as the number and features of services and extensions grow there will be a growing need for varying degrees of coordination between the platform team and users of the platform during upgrades and technology replacement. As a platform engineering team, when we have 100s or 1000s of teams deploying to our platform, how can we better scale coordination activities related to upgrades and changes to foundational components of our platform?

While the majority of changes amount to patches or minor revisions where the combination of CI and preview environment testing can be counted on to expose most issues that the platform team can deal with alone, throughout the course of each year, there will still be releases that require a change on the part of platform users.

Kubernetes itself is a good example. Kubernetes has an evolutionary cycle of 3 minor revisions per year. The project follows a process of deprecation with its APIs, where each release will deprecate or promote APIs using a standard schema. That API schema looks like this:

```
apiVersion: admissionregistration.k8s.io/v1alpha1
Kind: MutatingWebhook

apiVersion: apps/v1
Kind: Deployment
```

When an API is promoted, say in the above example, the “admission registration” API goes from v1alpha1 to v1beta1 - the v1alpha1 will be marked as deprecated in the next release. Two cycles later, it will be removed entirely. During the two cycles before the v1alpha1 API is removed it will still be usable to provide a period of backward compatibility. But anything deployed to the cluster using the v1alpha1 version will have to be updated to the newer v1beta1 by the removal time.

If any time there is a Kubernetes upgrade an event is published, we could set up a service to catch that event, check the Kubernetes API for any changed API versions, and scan everything running in the cluster to see which services are using outdated versions. That would give us a clear list of deployments that need updating—probably within the next eight months—to avoid breaking when the next Kubernetes release rolls out.

Now, if we also required every deployment to include team info in its metadata, our *upgrade watcher* service could take things a step further. It could automatically create Jira issues for each team, detailing exactly what’s changing in the API version including links to recommended upgrade strategies. That way, teams would get a heads-up with clear next steps, making the upgrade process much smoother.

Let’s look at how to design and implement an event-driven Release process that can be consumed and used by both the Platform team and the Platform users.

### 9.3.1 release-api

Our goal should be to reduce friction between the platform and application teams by providing a mechanism to iterate and evolve without constantly interfering with one another. In Figure 9.6, we have a sample of events as we roll out a new version of servicemesh.

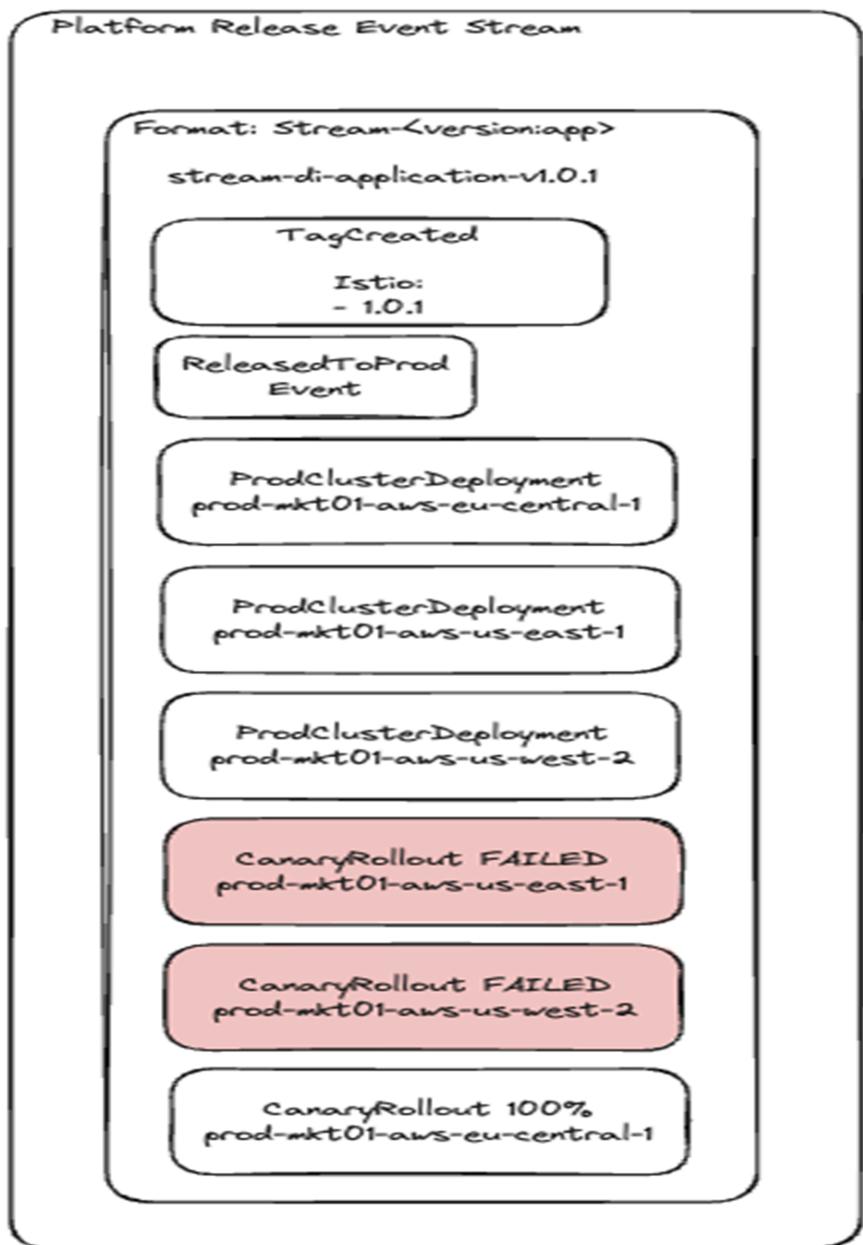
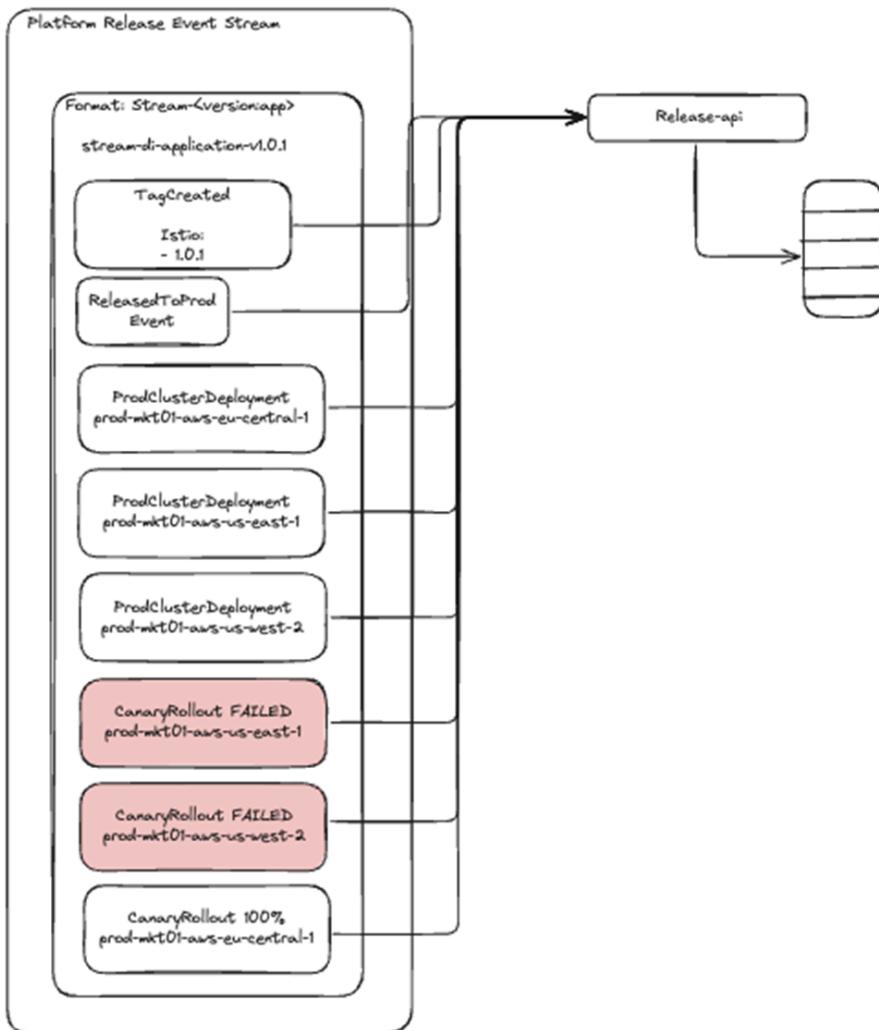


Figure 9.6 This figure shows a sample of events that represent the rollout of a new version of Istio to our Engineering Platform

In this example, we assume that the platform comprises several Kubernetes Clusters spanning three regions. When we release the update to production, each participating cluster will attempt its rollout of the Istio upgrade. The progress of these events could be represented as events. Events should be kept simple and lightweight by asking a few simple questions like these:

1. Where did it happen?
2. What happened?
3. Optional: pass/fail

Pass/Fail is optional because some events exist for “something” that happened, but it has no state other than it happened. For example, the “triggering” of a deployment means that a deployment was attempted. The state of whether that deployment passed or failed is reported back in a unique event later on. In Figure 9.7, you can see how a release-api can solve this.



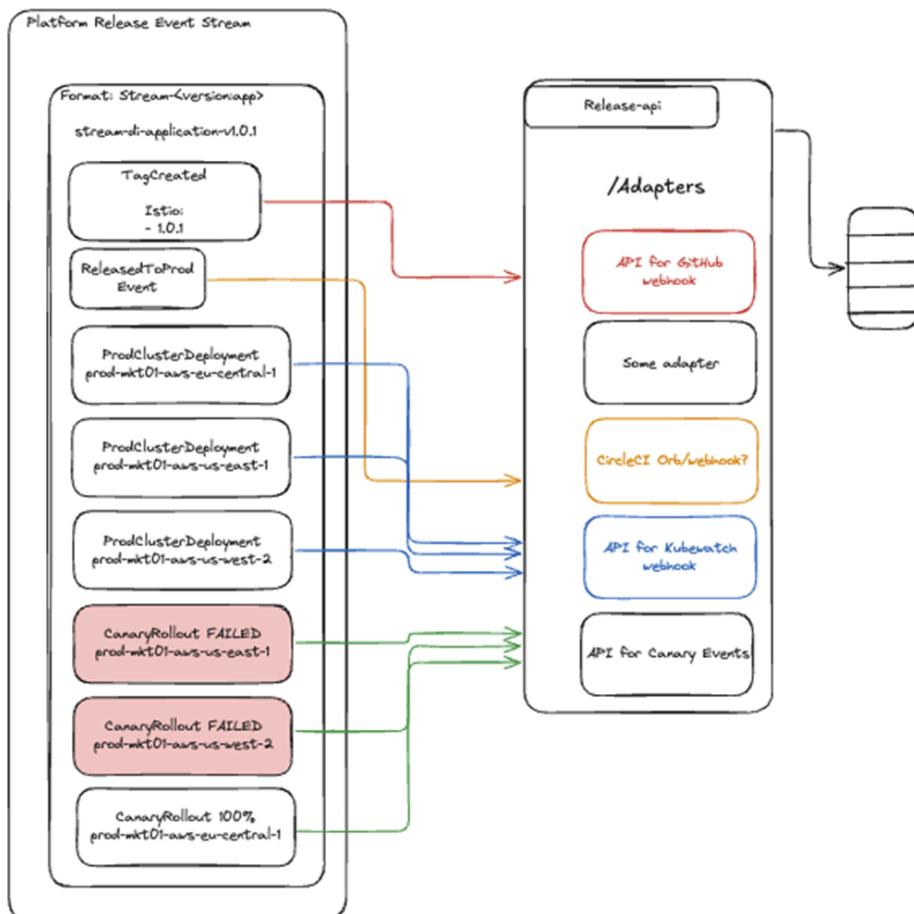
**Figure 9.7 A release-api is proposed to consume the events**

When we look at these example events, they could be thought of as an immutable event stream. A release api could be configured to watch these events and take action based on the events that occur. However, the most complex aspect is that the events come from many different places.

### 9.3.2 Adapter Pattern

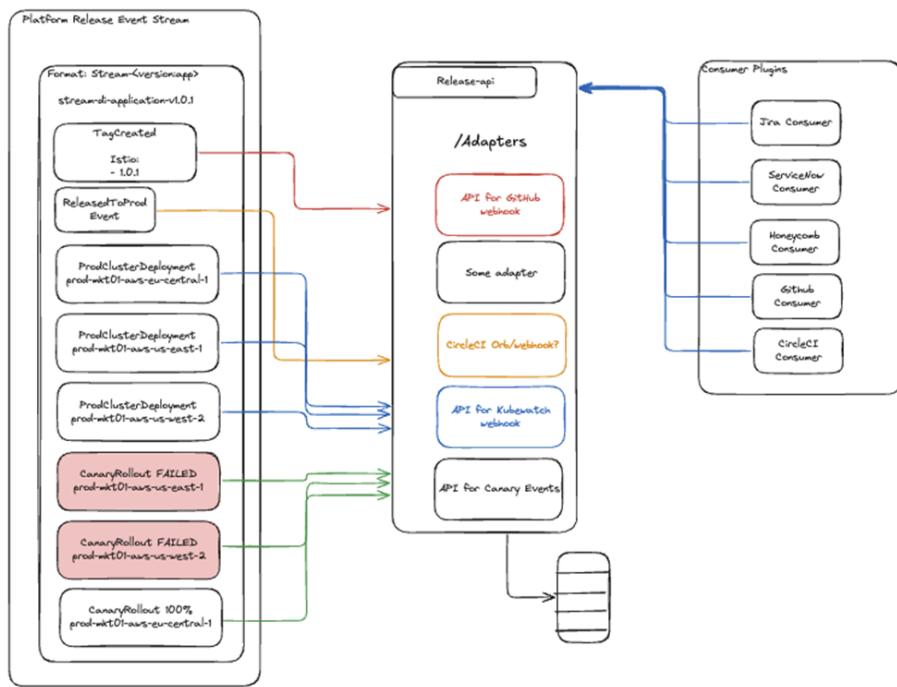
In figure 9.7, the *TagCreated* event is coming from GitHub. The *ReleasedToProd* event may come from a CircleCI workflow. The *ProdClusterDeployment* event could come from a workflow's CircleCI Step or the Deployments API (via an admission controller) in the Kubernetes cluster. The *CanaryRollout* event would come from our canary controller (i.e., ArgoCD or Flagger).

The point is that our events come from many places in potentially many different formats. Figure 9.8 shows the release-API expanded to show the specific resources you might want to consider.



**Figure 9.8** To consume the events in our **release-api**, we'll need a pattern from the many systems that produce them.

Each type of event will require an “adapter” to consume the events and convert them into our standardized event model. While this may sound complex initially, keep in mind that monitoring our enterprise systems requires many different types of instrumentation and agents. This pattern is something we do already to aggregate data. We are just focusing on Event data and getting events into a stream to which any API could subscribe. By separating the publishing of the events from the consuming we greatly simplify creating a successful event collection process. This dual benefit has a massive time-saving effect, creating a flexible and repeatable pattern for adding new systems into the service event stream. Figure 9.9 shows the relationship between this event stream and consumer plugins for standard services like JIRA, ServiceNow, and CircleCI.



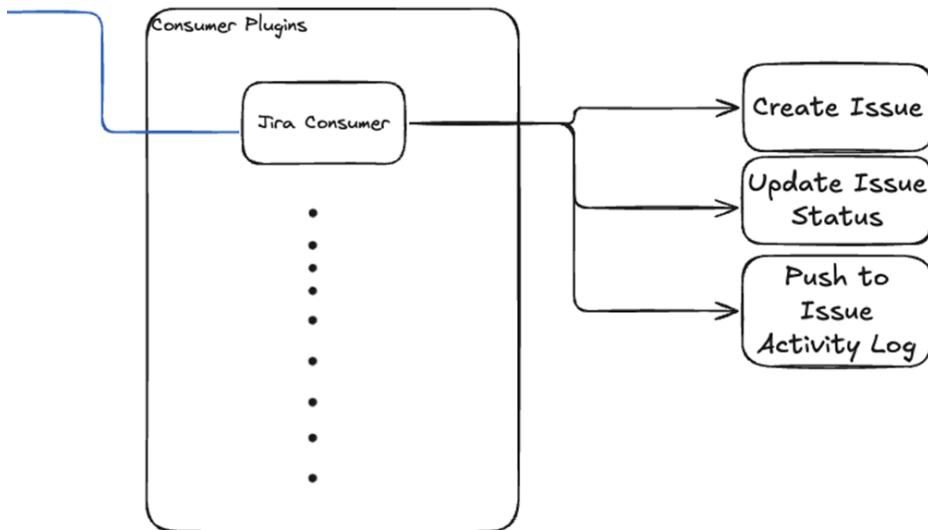
**Figure 9.9 With a consolidated consumption model, consumers can now be developed to talk to the release API and do something with the data. Examples of consumer plugins will be services used for issue tracking, CMDB, source control, and CI/CD**

These are custom consumers used to enhance the capabilities of our platform and create more effective and scalable platform lifecycle patterns and are not meant to replace the normal integrations for these tools or services.

With a standard model for event formats, regardless of the source, creating consumers to provide enhanced capabilities is now very easy. This opportunity is not limited to just the platform product teams. The Platform team can Starter Kits for building a consumer service that pulls an event stream from the release API. Application teams can use the kits to create consumers around the events that matter most to them.

Let's look at some of the more common event types that can be used to create an improved experience for both platform maintainers and users.

### 9.3.3 Adapter Pattern - Issue Tracking



**Figure 9.10 An issue-tracking consumer might create and manage issues in response to specific events.**

Our first example is issue management. Imagine the platform team has published an event that signals a new release of Kubernetes that will be going out soon. This could be done by publishing an event that says the “Preview” environment has received a new version of Kubernetes, and the standard policy is that the Preview environment is updated 1 month before Production environments are, giving teams time to review their application health on the new version.

Now that we have the release-API, instead of publishing a company-wide Slack announcement that nobody will read, we can have a Jira consumer that creates a new issue on our application team’s work board with an urgency level of medium. In Figure 9.11, you can see the platform team publishing the events.

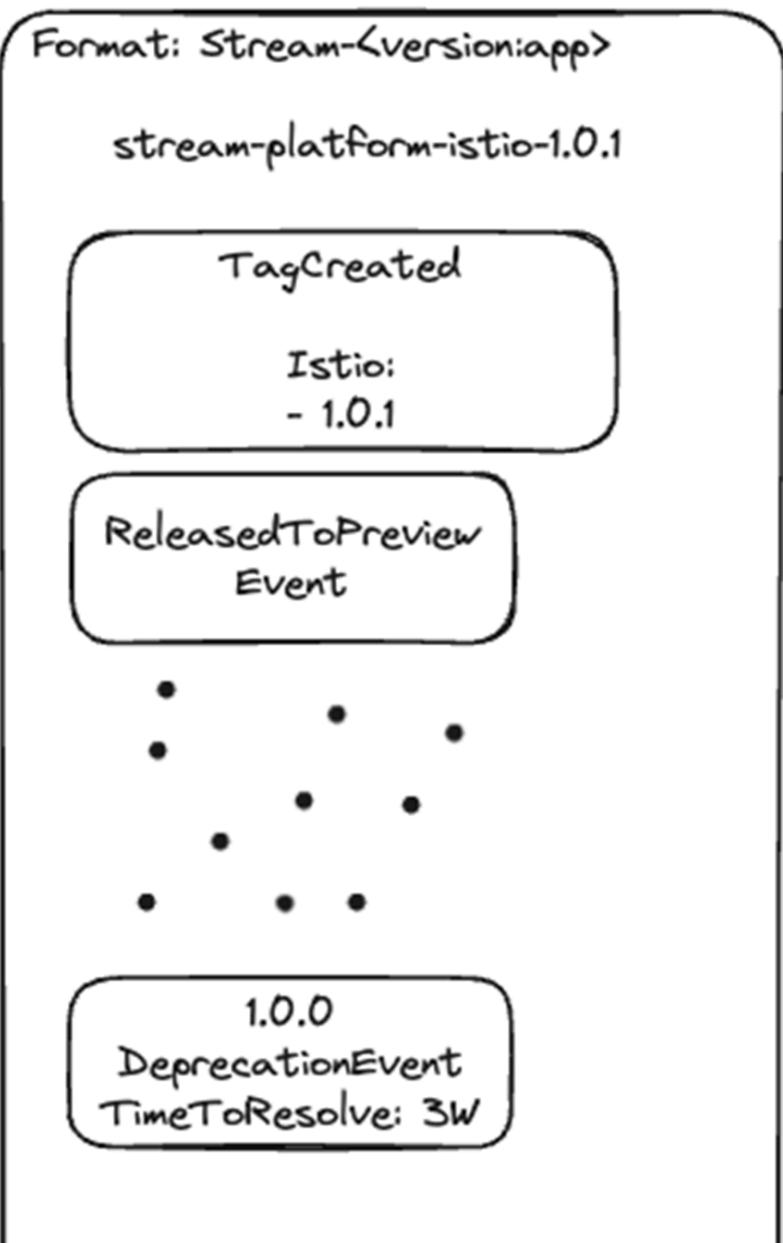


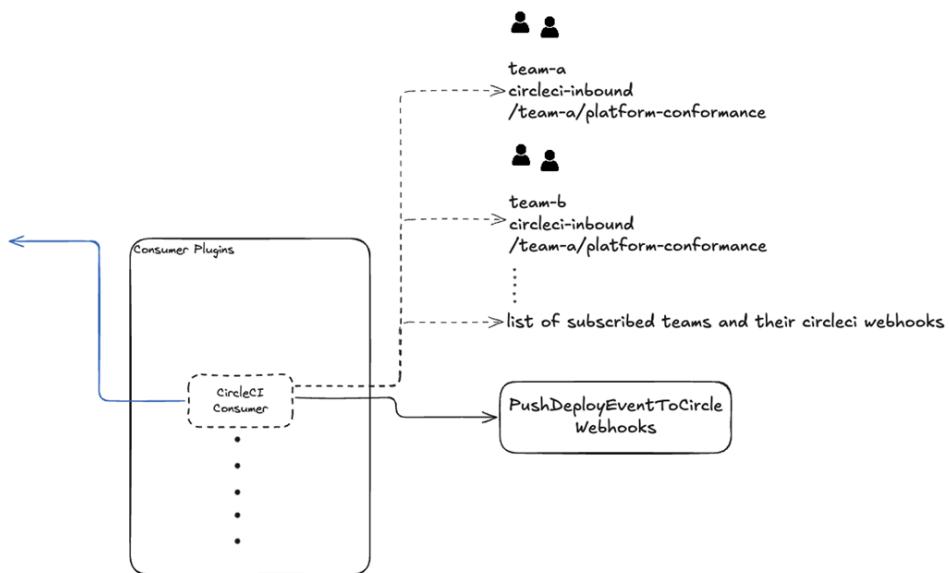
Figure 9.11 The platform team will continue to publish events regarding the state of the upgrade, including deprecation notices.

As the time to production release grows closer, if the task isn't completed the consumer can update the task urgency to ensure our application team prioritizes verifying their service on the new version simply by consuming the deprecation events and updating the upstream task statuses.

### 9.3.4 Adapter Pattern - CI Hooks

Continuing with our platform upgrade example. A consumer could also be created to respond to all of these routine upgrade announcements, most of which will not require developers to make any changes but nonetheless need to be verified.

Figure 9.12 illustrates how application teams can enhance collaboration with platform consumers by exposing inbound webhooks from their CI systems of choice and opting into a consumer-driven model for deployment events.

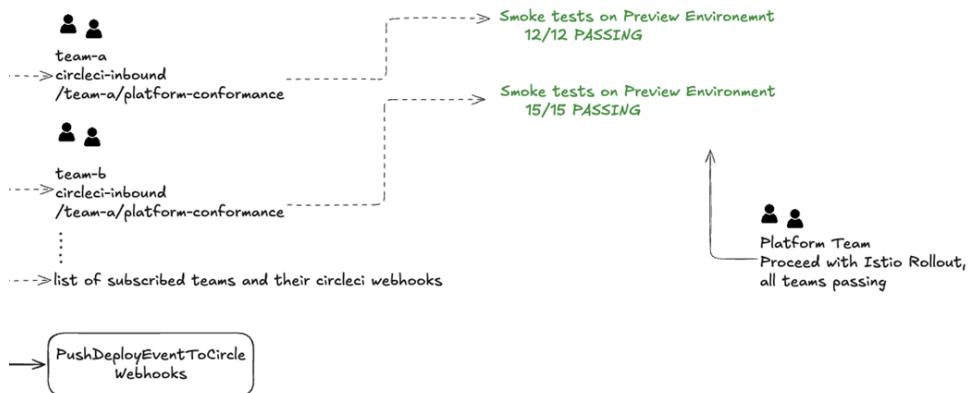


**Figure 9.12** The consumer API can subscribe to specific platform deployment events, and then inbound workflow triggers could be created, allowing a central consumer plugin (e.g., the CircleCI consumer) to trigger and coordinate deployment workflows seamlessly.

Any time a cluster component upgrade occurs in the Preview environment, users' conformance test pipelines could trigger to deploy and test the current version to reveal any conflicts created by the change. If all tests are passed, nobody needs to do anything! The consumer API could even find and close the Jira ticket we talked about above. If there are errors, it could add relevant logs and pipeline links to the Jira ticket to make responding and fixing faster.

This is often the case with the upgrade of platform services. Because technologies like Istio and Kubernetes make **considerable** efforts to provide backward compatibility and are **highly** selective with the graduation and deprecation of APIs, more often than not, upgrading our core services is a non-event. But because of the massive uncertainty in the absence of a system like this, our organization, before our release-API, had to manually check with every single team before moving forward with the changes. This also meant that changes had to be scheduled way in advance, and the schedule would be pushed back if any interruptions or new priorities emerged.

Where this sort of event-driven conformance testing is made an engineering practice requirement, then the platform product could extend the automation a step further. Figure 9.13 illustrates how a platform team can leverage automation to monitor Jira upgrade-notice issues and track their resolution status.

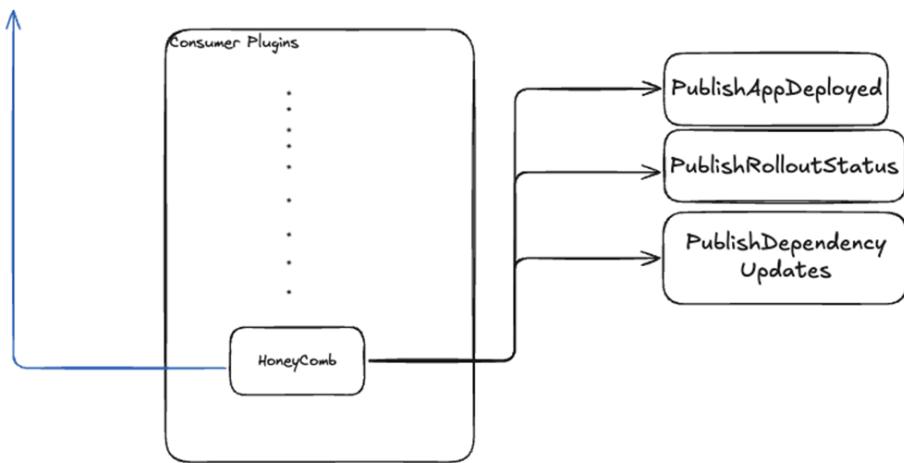


**Figure 9.13** The platform team could create a consumer API that watched for the creation of all the Jira upgrade-notice issues and whether they were all auto-closed after successful conformance tests. They could proceed with greater confidence in releasing the changes to production or alternatively have a detailed list of the impacted services and the issues found.

This is actually a very effective replication of human behavior. Even when people do respond to the global chat messages about checking their service in Preview against a new cluster component upgrade, almost universally they will only look at the monitors to see if anything is red or run automated tests if they have them. Automating that response guarantees fast and universal coverage. This doesn't mean that every team will have bullet-proof testing or that automated testing is ever 100 percent accurate but it does mean that what can be done will be done rapidly. That covers most of the changes and the overall impact is to maintain a higher operational maintenance velocity.

### 9.3.5 Adapter Pattern - Observability Hooks

An observability consumer could process and publish all of these new kinds of events into the observability tool data pool, including application deployment events (PublishAppDeployed), rollout statuses (PublishRolloutStatus), and dependency updates (PublishDependencyUpdates). All these events can then be added to the normal observability radiators, providing teams with better real-time visibility into the operational health and performance of their systems. In Figure 9.14, we show an example of an observability consumer pushing deployments to the metrics platform.



**Figure 9.14 Example of an Observability consumer that pushes deployment events to our metrics platform**

If all team release event data is published to the general observability system, it also means that teams own individual service health monitoring can more easily correlate issues against their dependencies. This correlation is demonstrated in Figure 9.15.

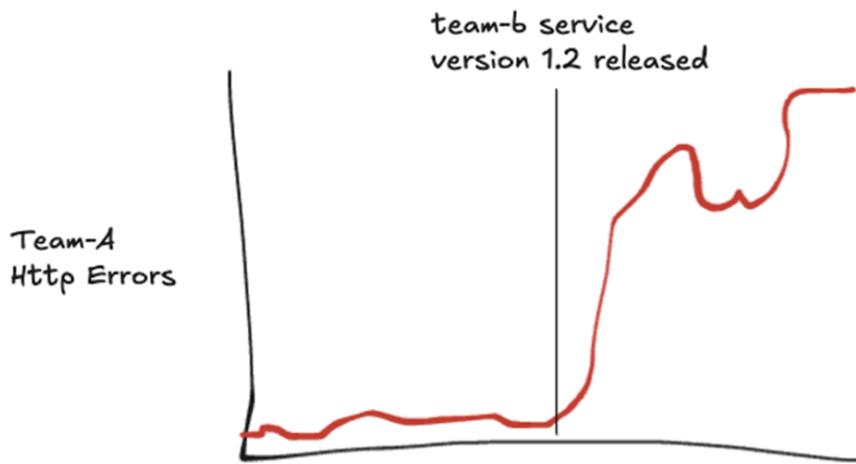
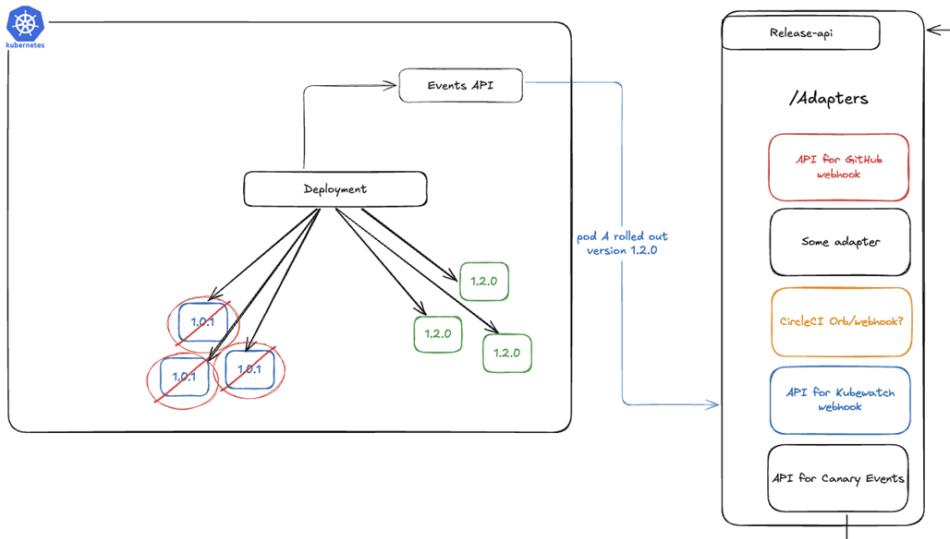


Figure 9.15 Team-A's dashboards can show events from internal services on which they depend and make it easier to diagnose remote changes that impact their services.

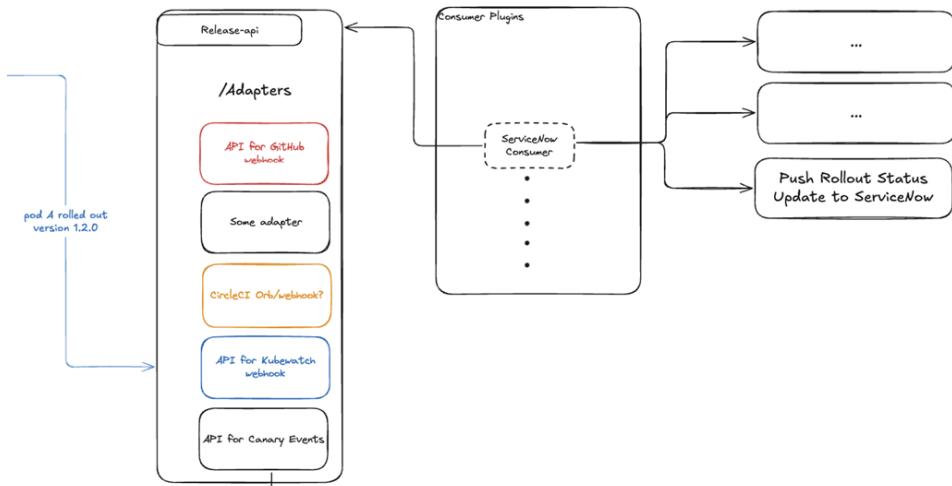
### 9.3.6 Adapter Pattern - CMDB and Audit Gathering

Consumer APIs can be created to automate audit-gathering requirements. For example, assume our organization uses ServiceNow to track change management data that includes the current list of all services running and versions as shown in Figure 9.16.



**Figure 9.16** If events are being propagated from our Kubernetes cluster to the release API, then all deployments will generate events. Every single pod event is published. So, when one of the 1.0.1 pods is terminated, that is a unique event. And when one of the 1.2.0 pods successfully deploys and starts taking traffic, that is also an event.

From an audit perspective, as illustrated in Figure 9.17, we can configure a ServiceNow consumer that pushes all new release events to ServiceNow.



**Figure 9.17** The event for Pod A is sent to the release-api, where our ServiceNow consumer consumes it. Our consumer pushes the update to ServiceNow's CMDB.

We can then consume the events in our ServiceNow CMDB API and Portal. In most CMDB systems, the expectation is that we have a catalog of the deployed services and the information that explains what they are, who owns them, and what they are connected to. In our case, all of this information is already handy in the deployment event (because it was necessary to create the deployment anyway), so we can readily forward the deployment event information to our CMDB in ServiceNow, allowing ServiceNow to update its application map.

In section 9.3 we have covered several situations in which deployment event data can be used to automate necessary but laborious processes. But this is just one kind of event. There are all manner of events being generated or that can be captured. Becoming confident in building solid event-driven capabilities is a necessary skill for platform engineers

## 9.4 Summary

- Scaling an engineering platform requires intentional architectural changes, not just adding servers.
- Cost assessments, including the cost of delay, are critical to justifying investments in scaling platform capabilities.
- Effective scaling includes aligning platform team structures, roles, and processes to match evolving business needs.
- Establishing domain boundaries and autonomy between teams enables efficient collaboration and scalability.
- Event-driven release architectures enhance reliability and trust by reducing friction between platform and consumer teams.
- Dynamic pipelines that adapt to the number and configuration of clusters improve deployment efficiency at scale.
- Lifecycle pipelines for individual services and extensions improves release quality and removes bottlenecks.
- Distributed deployers at the cluster level reduce centralized orchestration overhead and enable quicker rollouts while reducing blast radius from many kinds of common errors.
- Platform APIs should be the foundation for all capabilities, ensuring seamless integration and extensibility.
- Scaling strategies should prioritize the platform user/consumer experience, reducing wasted time and friction.
- Developer portals like Backstage improve discoverability but must align with API-first principles to avoid creating support burdens.
- Event-driven architectures are a key skill in scaling engineering platforms.
- Adapter patterns standardize data flow across diverse systems, simplifying integration and reducing complexity.
- Observability integrations allow real-time tracking of deployment events, enhancing system visibility and troubleshooting.
- Event-driven notifications ensure platform consumers stay informed about upgrades and deprecations, reducing coordination overhead.
- Integrating release events into CMDBs and audit systems supports compliance while maintaining lightweight, scalable architectures.

- [1] <https://circleci.com/docs/dynamic-config/>
- [2] <https://github.com/ThoughtWorks-DPS/circlepipe>
- [3] <https://argoproj.github.io/cd/>
- [4] <https://fluxcd.io>

# **10 Platform Product Evolution**

## **This chapter covers**

- Measuring the success of your platform organization.
- How platforms as products are a differentiator in your platform evolution.
- Intelligent assistants
- Internal developer platforms and products.

In the dynamic world of software development, organizations are continually seeking ways to accelerate delivery, improve reliability, and enhance the developer experience. As companies grow and their software ecosystems become more complex, an effective platform strategy becomes critical. This chapter explores the evolution of platform products, focusing on how treating platforms as products can be a differentiator and how measuring success, embracing cultural shifts, and leveraging modern tools and methodologies can drive organizational success.

We'll revisit our favorite company we have been discussing throughout the book, PETech, to illustrate these concepts in practice. PETech's journey from a monolithic application to a microservices architecture highlights the challenges and opportunities in evolving platform products, providing practical insights into how organizations can navigate similar transitions.

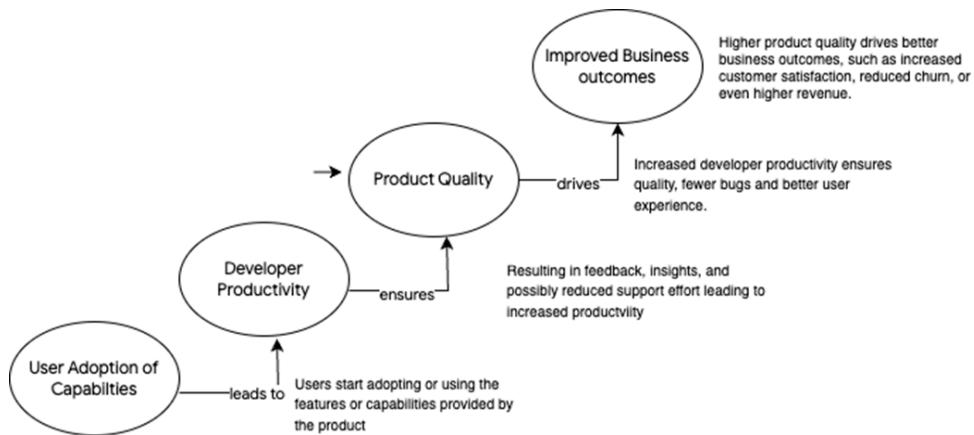
## **10.1 Measuring the success of your platform organization**

Now that you have embarked on a platform journey and have started seeing some improvements, it is essential to measure your progress. As we have discussed, evolving your platform to the next level almost always requires knowing your progress.

For any platform initiative to be successful, it's essential to establish metrics that align with organizational goals. Measuring the success of your platform implementation provides insights into its effectiveness, adoption, and areas for improvement. It ensures that the platform delivers value to its users—the developers and operators—and, ultimately, the business by facilitating better decision-making and strategic planning.

### 10.1.1 The Platform Value Model

As discussed in chapter 3, the Platform Value Model offers a framework for quantifying the value delivered by a platform. It focuses on several key areas. Figure 10.1 shows the evolution of value quantification as you translate the engineering capabilities to business outcomes.



**Figure 10.1 Quantifying the value across multiple levels, from building the capabilities to addressing the business outcomes.**

Now, let us look at each of the relevant metrics for each of these steps in the value articulation process.

**User Adoption Metrics:** Tracking the number of teams and developers using the platform helps gauge its relevance and usefulness. High adoption rates often correlate with increased efficiency and satisfaction among users.

**Productivity Metrics:** Measure improvements in four key DORA metrics, highlighting the platform's impact on operational efficiency. These metrics reflect how quickly and reliably the organization can deliver customer value.

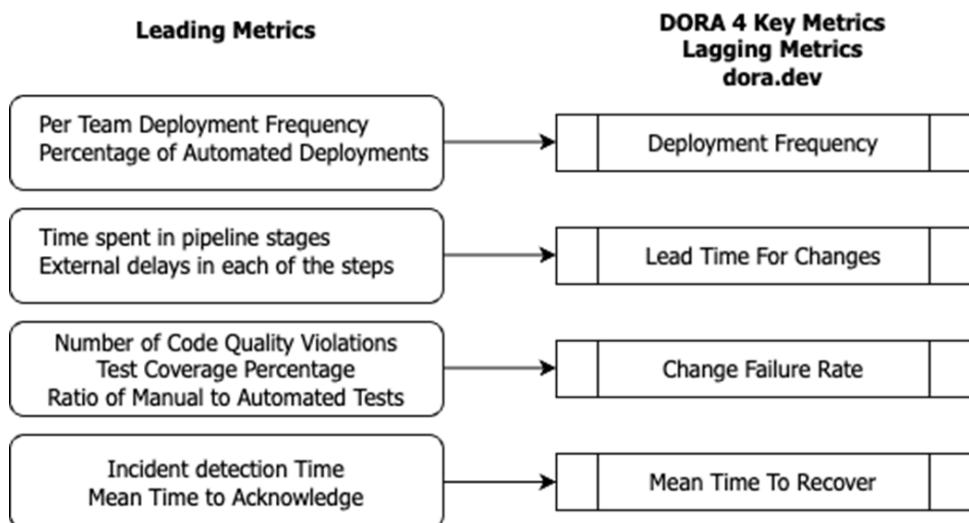
**Quality Metrics:** Assessing the reduction in incidents, failures, and defects provides insights into the platform's role in enhancing system stability and reliability. Improved quality metrics can lead to better customer satisfaction and trust.

**Business Metrics:** Evaluating the impact on revenue, customer satisfaction, and market competitiveness directly connects the platform's performance to organizational success. These metrics demonstrate how the platform contributes to achieving strategic business objectives.

Organizations can make data-driven decisions to optimize platform strategies by integrating these metrics into regular reporting and analysis.

### 10.1.2 PETech's Approach to Measurement

At PETech, the platform team recognized the importance of measuring success early in their platform evolution journey. They established key performance indicators aligned with the Platform Value Model to ensure their efforts were focused and effective. They also aligned their KPIs with the four key metrics from the DORA framework while incorporating leading metrics to provide a more proactive approach to performance improvement. Figure 10.2 shows the typical leading metrics and how they are related to the popular DORA 4 key metrics.



**Figure 10.2 Key leading metrics used by PETech to understand and use the DORA lagging metrics**

1. Deployment Frequency: Increasing the number of successful deployments daily was crucial for PETech to respond swiftly to market demands. A higher deployment frequency indicated a more agile and efficient development process. In addition to this lagging metric, PETech monitored leading metrics such as the percentage of automated deployments and the frequency of deployments per team. These leading indicators helped identify potential bottlenecks and areas where automation could be further enhanced.
2. Lead Time for Changes: Reducing the time from code commit to production release allowed PETech to deliver new features and fixes more rapidly. Shorter lead times improved competitiveness and customer satisfaction. As a leading metric, they tracked the time spent in different deployment pipeline stages, such as build, test, and approval. PETech could identify specific stages causing delays by analyzing these breakdowns and implementing targeted optimizations.

3. Change Failure Rate: Lowering the percentage of deployments causing a failure in production was essential for maintaining system reliability. A lower change failure rate reduced downtime and enhanced user trust. To proactively address this, PETech also tracked leading indicators like the number of code quality violations, test coverage percentage, and the ratio of manual to automated tests. These metrics provided early warning signs of potential issues, allowing teams to address quality concerns before they impact production.
4. Mean Time to Recovery (MTTR): Decreasing the time it takes to restore service after an incident minimizes the impact of outages. A shorter MTTR improved overall service availability and customer experience. PETech complemented this with leading metrics such as incident detection time and mean time to acknowledge (MTTA). Focusing on these early stages of incident response could improve their monitoring and alerting processes, ensuring faster resolution times.

By focusing on these metrics, PETech could objectively assess the platform's impact on developer productivity and operational efficiency, enabling them to identify successes and target improvement areas.

In the ensuing exercise, let us now look at the kinds of leading metrics you have in your organization and why.

### **10.1.3 Exercise 10.1: Identify the leading engineering platform metrics for your organization**

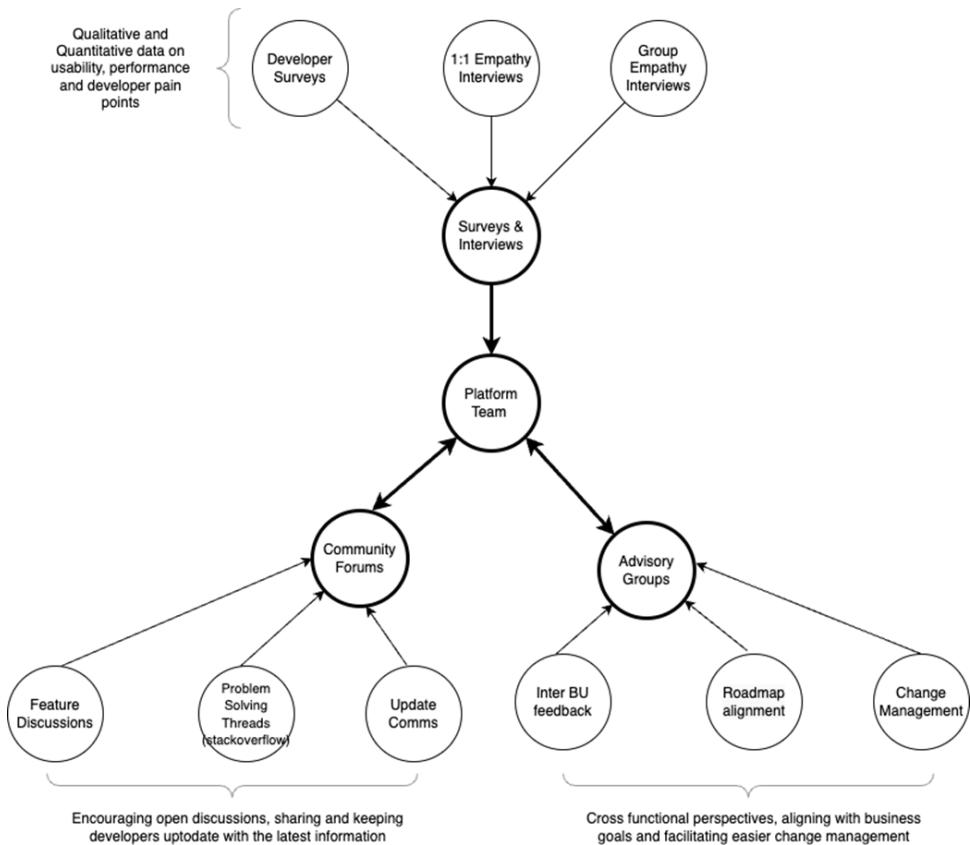
**Objective:** While the inventory of the leading metrics you might have in an organization is a finite set, it is important to identify the appropriate leading metrics that would lead you to your DORA metrics (4 Key Metrics).

#### **INTENDED OUTCOME:**

You will create a list of leading metrics and explain how those metrics relate to the DORA metrics.

### **10.1.4 Implementing Feedback Loops**

To gather qualitative data alongside quantitative metrics, PETech recognized the importance of ongoing dialogue with their development teams. While deployment frequency and lead time provided valuable insights into the platform's performance, they did not always capture the nuances of developer experience, usability issues, or the specific needs of different teams. To address this, PETech implemented regular feedback loops, employing various methods to engage with developers and stakeholders as shown in Figure 10.3



**Figure 10.3 Feedback loop ecosystem for the platform engineering team enabling comprehensive end-user-driven development**

Now, let us look at how PETech built each of the core pieces of the feedback loop ecosystem - Surveys & Interview, Community Forums and Advisory Groups.

### Surveys and Interviews

**Surveys:** PETech conducted periodic surveys (both quantitative and qualitative) to collect developer satisfaction scores and feedback on the platform's usability, performance, and features. By structuring surveys around different aspects of the platform—such as ease of use, documentation quality, and the effectiveness of automated workflows—PETech could identify which elements were working well and which required attention.

**Interviews:** In addition to surveys, the platform team held one-on-one and group interviews with developers, team leads, and operations personnel. They enabled the team to delve into the context behind survey responses, uncovering underlying issues that might not be evident from quantitative data alone. For instance, developers might express frustration over certain features during an interview, which could be traced back to a lack of documentation or unclear workflows. The interviews also fostered a sense of involvement, making developers feel their voices were heard and valued in the platform evolution process.

By combining surveys and interviews, PETech gathered rich data that informed their decisions on platform enhancements. This process ensured that platform improvements were directly aligned with user needs, driving higher adoption and satisfaction.

### **Community Forums**

PETech established community forums as an open space for developers and platform users to discuss features, challenges, and best practices. These forums were designed to be more than just a feedback channel; they became a collaboration and knowledge-sharing hub. Developers used these forums to:

- **Voice Concerns and Suggest Improvements:** Developers could report issues, request new features, or suggest enhancements in a public setting, allowing the platform team to gauge the demand for specific changes. This open dialogue made prioritizing improvements based on real-world usage and developer sentiment easier.
- **Share Experiences and Solutions:** Developers often face similar challenges while using the platform. The forums allowed them to share tips, workarounds, and solutions, fostering a collaborative environment. For example, a developer who had optimized a deployment pipeline could share their approach, helping others achieve similar efficiencies.
- **Stay Informed about Updates:** The platform team used the forums to announce updates, share release notes, and provide guidance on new features. Proactively communicating changes helped teams adapt to updates more smoothly and reduced the learning curve associated with new functionalities.

Community forums became vital to PETech's developer ecosystem, promoting transparency, collaboration, and a sense of community. They helped the platform team dynamically and interactively monitor developers' evolving needs and experiences.

### **Advisory Groups**

To ensure that the platform roadmap reflected the diverse needs of all teams, PETech formed advisory groups consisting of representatives from various departments, including development, operations, security, and product management. Their approach to the platform's evolution was done by:

- **Prioritizing Features:** The advisory groups met regularly to review feedback and discuss upcoming features and improvements. With insights from different areas of the organization, they helped the platform team prioritize features that would have the most significant impact. This cross-functional perspective ensured that platform development was balanced, addressing needs across the board rather than focusing on the loudest voices or the most immediate concerns.

- Aligning Roadmap with Business Objectives: The advisory groups worked closely with the platform team to align the platform's roadmap with broader business goals. For example, if the company aimed to accelerate the rollout of a new product line, the advisory group might prioritize platform features that streamline deployment and testing processes. This alignment ensured platform investments delivered tangible business value and supported strategic initiatives.
- Facilitating Change Management: Platform changes often require updates to workflows, processes, or team responsibilities. Advisory group members acted as change champions within their respective teams, helping to communicate the benefits and implications of platform changes. They facilitated the adoption of new features and practices, reducing resistance and smoothing the transition process.

The advisory groups ensured platform development was inclusive and aligned with user needs and business goals. This collaborative approach led to a more targeted and effective platform roadmap, ultimately driving higher adoption and satisfaction.

### **This Comprehensive Approach to Feedback**

By implementing a multi-faceted feedback loop that included surveys, interviews, community forums, and advisory groups, PETech created a holistic feedback ecosystem. This approach ensured that the platform evolved in response to real user needs, capturing both the quantitative and qualitative aspects of developer experience. It allowed the platform team to:

- Identify Pain Points Promptly: Regular feedback cycles helped detect issues early, preventing the embers from becoming total blown fires. For example, feedback about a cumbersome deployment process led to developing of a new automated pipeline, significantly improving efficiency.
- Celebrate Successes: By engaging with developers and recognizing areas where the platform excelled, PETech fostered a culture of continuous improvement and collaboration. Celebrating successes, such as the successful rollout of a new feature or the reduction of deployment time, reinforced positive behaviors and motivated further progress.
- Adapt to Changing Needs: As the organization grew and its needs evolved, the feedback mechanisms allowed the platform team to adapt quickly. For instance, as security requirements became more stringent, feedback from the advisory groups guided the integration of enhanced security features into the platform.

This comprehensive feedback approach not only improved the platform itself but also fostered a culture of continuous improvement. Developers felt a sense of ownership and investment in the platform, leading to higher engagement, more innovative ideas, and a platform that genuinely served users' needs.

Well, we have looked at how PETech is using the feedback mechanism. However, having some hands-on practice to implement in your organization is important. To do so, let us go back to the other fictional company VitalSigns.online, we introduced in chapter 6.

### **10.1.5 Exercise 10.2: Create an approach for feedback mechanism at VitalSigns.online**

**Objective:** Implement a feedback mechanism to gather qualitative and quantitative insights from internal developers, third-party developers, and partners regarding the usability, performance, and challenges of using VitalSigns' API services.

Recommended steps:

1. Provide the steps for setting up a Surveys & Interviews process as shown in the PETech case above
2. How would you form a Platform Team responsible for feedback analysis
3. Based on the example provided above, how would you create community forums for discussions
4. Should you be forming Advisory Groups for Strategic Feedback

#### **INTENDED OUTCOME:**

By implementing this simple feedback mechanism, VitalSigns should be able continuously improve its developer platform, ensuring both internal and external stakeholders are satisfied with the usability and performance of their APIs.

## **10.2 Platform as Products as the Differentiator**

As introduced in Chapters 1 and 2, treating the platform as a product involves applying product management principles to its development. This means prioritizing user-centric design, continuous improvement, and a clear value proposition. By adopting this mindset, organizations can create functional and delightful platforms, driving higher adoption and satisfaction rates.

At PETech, the platform team adopted the platform-as-a-product mindset by taking deliberate steps to ensure that the platform was not just a set of tools but a product designed to solve specific problems and deliver value to its users. This approach involved strategic planning, continuous alignment with organizational goals, and adopting best practices to foster a culture of adaptability and innovation. Here's how PETech approached this transformation:

### **10.2.1 Defining the Platform Vision and Mission**

The first step in adopting the platform-as-a-product mindset was to establish a clear vision and mission for the platform. The platform team at PETech recognized that without a well-defined purpose, the platform could quickly become a collection of ad-hoc features and tools, lacking coherence and strategic direction. To avoid this, they focused on answering fundamental questions: *What is the platform's primary purpose? Who are its users? What problems is it designed to solve?*

**Platform Vision:** The team articulated a vision that described the long-term impact they wanted the platform to have on the organization. For PETech, the platform vision centered around creating a seamless developer experience that empowered teams to build, deploy, and operate their applications efficiently. The vision was about technology and fostering a culture of innovation, speed, and reliability across the organization. This overarching vision provided a guiding star for all development efforts, ensuring every feature, enhancement, or decision contributed toward a unified goal.

**Platform Mission:** Building on this vision, the team crafted a mission statement outlining how the platform would achieve its vision. The mission focused on specific outcomes, such as reducing the time to market for new features, ensuring high system reliability, and providing robust tools that abstracted the complexities of infrastructure management. By defining a clear mission, the team established a concrete set of objectives that shaped the platform's development path.

This clarity in vision and mission had a profound effect. It gave the platform team the to prioritize their fundamental problems. It also helped align stakeholders across the organization, from developers to business leaders, around a shared understanding of what the platform aimed to achieve. This alignment ensured that every feature and improvement was evaluated based on how well it contributed to the platform's overarching goals.

### **10.2.2 Establishing a Product Roadmap**

With the vision and mission in place, the next step was to develop a product roadmap that translated these high-level goals into actionable steps. The roadmap was a list of features and a strategic plan that prioritized work based on user needs, business objectives, and technical feasibility.

**Planning Features and Enhancements:** The platform team conducted extensive user research, gathering feedback from developers, operations teams, and other stakeholders to identify pain points and opportunities for improvement. This research helped the team understand user groups' specific needs and challenges. For example, they discovered developers were spending significant time on manual deployment processes, leading to delays and errors. This insight led to prioritizing features like automated deployment pipelines and self-service infrastructure provisioning.

**Strategic Alignment:** The roadmap also considered the organization's strategic goals. For PETech, this included objectives like accelerating the rollout of new digital services, improving system reliability, and reducing operational costs. The team ensured their work directly contributed to the company's success by aligning the platform roadmap with these goals. Features that had the potential to drive significant business value were prioritized, while lower-impact items were scheduled for later iterations.

**Communication Tool:** The roadmap was a powerful communication tool, aligning stakeholders on expectations, timelines, and priorities. It provided transparency into what the platform team was working on and why, helping manage stakeholder expectations. Regular roadmap reviews allowed the team to update stakeholders on progress, incorporate new feedback, and adjust plans as necessary. This open communication fostered trust and collaboration between the platform team and its users.

Establishing a product roadmap transformed the platform development process at PETech. It provided a clear path forward, enabling the team to focus on delivering features that mattered most to their users and the organization. It also reduced ad-hoc requests and scope creep, as stakeholders had a transparent view of the platform's direction and priorities.

### **10.2.3 Exercise 10.3: Create a platform product roadmap blueprint for VitalSigns.online**

Objective: The objective of this exercise is to think about how VitalSigns.online would create a platform product roadmap. Remember, you are not creating the roadmap itself as that would require understanding of the domain specific activities of VitalSigns.

Recommended Steps: In order to achieve this objective, we recommend the following steps

1. What are the teams and types of requirements analysis you will be doing if you are creating a platform product roadmap (Hint: Conduct stakeholder management)?
2. What are the strategic goals for the overall business that will be addressed by the platform product capabilities being built?
3. What are the communication tools you will need to use to increase trust among your stakeholders on the progress of the platform capability buildout

### **10.2.4 Implementing Agile Practices**

To bring the platform vision and roadmap to life, PETech adopted Agile practices, focusing on delivering value incrementally and responding quickly to change. Traditional, long development cycles replaced iterative processes, allowing the platform team to adapt to feedback and evolving requirements.

**Iterative Development Cycles:** The team broke the roadmap into smaller, manageable increments, delivering new features and improvements in short sprints. This approach enabled them to release updates more frequently, providing users with immediate value and gathering feedback. For example, they started with a minimal viable product (MVP) that automated basic deployment tasks instead of building a full-fledged deployment automation tool over several months. This MVP was then iteratively enhanced based on user feedback, progressively adding more complex features like automated testing and rollback mechanisms.

**Continuous Feedback and Adaptation:** Agile practices emphasize the importance of constant feedback loops. After each sprint, the platform team conducted reviews and retrospectives to assess what went well and what could be improved. This process allowed them to adapt their plans in real-time, incorporating user feedback, addressing new requirements, and improving workflows. If a newly released feature received feedback indicating usability issues, the team could quickly iterate on the design in the next sprint rather than waiting for a major release cycle.

Cross-Functional Collaboration: Agile practices also promoted collaboration across different functions. The platform team included developers, operations engineers, security specialists, and user representatives. This cross-functional approach ensured the platform was built with a holistic view of user needs, operational requirements, and security considerations. Regular stand-ups, planning sessions, and collaborative tooling helped keep everyone aligned and moving towards common goals.

Implementing Agile practices had a transformative effect on the platform's evolution. The platform team became more responsive to user needs and changing requirements, able to pivot quickly when necessary. They moved away from the pitfalls of lengthy development cycles, where features could become outdated when released. Instead, they embraced a continuous delivery and improvement culture, where the platform evolved organically based on real-world usage and feedback.

### **10.2.5 The Role of the Platform Product Manager**

To drive this approach of agile practices introduced in the last section, PETech appointed a Platform Product Manager, Alex, whose role was crucial in ensuring the platform's success. Here were the three key activities for Alex.

User Research: Alex invested time in understanding developer workflows, pain points, and requirements. By conducting interviews, surveys, and observational studies, Alex gathered valuable insights that shaped the platform's features and user experience.

Stakeholder Engagement: Collaborating with leadership, development teams, and operations helped align the platform's priorities with business objectives. Alex facilitated group communication, ensuring the platform met technical and strategic needs.

Metric Tracking: Monitoring KPIs and adjusting strategies accordingly allowed Alex to make data-driven decisions. They regularly reviewed metrics to ensure the platform delivered tangible value and identified areas needing attention.

Alex's role was pivotal in bridging the gap between technical implementation and user expectations, ensuring the platform aligned with user needs and business objectives.

### **10.2.6 Differentiating Through User Experience**

By focusing on the developer experience, PETech's platform became a differentiator in several ways. Most notably, the following:

Simplifying Complexity: Abstracting infrastructure complexities allowed developers to focus on coding and innovation rather than dealing with underlying technical details. This simplification reduced cognitive load and accelerated development cycles.

Providing Self-Service Capabilities: Enabling teams to provision resources and deploy applications without bottlenecks increased autonomy and efficiency. Self-service tools empowered developers to act quickly, fostering a sense of ownership and responsibility.

Ensuring Consistency and Compliance: Standardizing configurations to meet security and governance requirements minimized errors and vulnerabilities. Consistent practices ensured that all teams adhered to organizational policies, enhancing overall system integrity.

The platform improved productivity by delivering an exceptional user experience and contributing to a positive organizational culture where developers felt supported and valued.

By adopting a platform-as-a-product mindset, defining a clear vision and mission, establishing a strategic roadmap, and implementing Agile practices, PETech created a technically robust platform aligned with its users' needs and the organization's strategic objectives. In Figure 10.4, we introduce the evolution of platform-as-a-product mindset.



**Figure 10.4 Visual representation of the lifecycle of the platform-as-a-product approach, highlighting how to transition from vision and mission to a fully operational, developer-centric platform**

#### CASE STUDY PETECH'S LAUNCHPAD: AN INTERNAL DEVELOPER PLATFORM

To bring the platform-as-a-product vision to life, PETech developed "LaunchPad," their Internal Developer Platform (IDP), which became a cornerstone of their development ecosystem.

Features of LaunchPad:

- Standardized Service Templates: Pre-configured setups for common microservices ensured best practices were followed across teams. These templates reduced setup time and provided a solid foundation for new services.
- Integrated CI/CD Pipelines: Automated testing, security scans, and deployments streamlined the release process—integration with existing tools minimized disruptions and enhanced efficiency.
- User-Friendly Interface: An intuitive portal for managing services, environments, and deployments made it easy for developers to navigate and utilize the platform's capabilities. A focus on usability reduced the learning curve and encouraged adoption.

- Extensive Documentation: Guides, tutorials, and FAQs supported developers at every step, providing resources for troubleshooting and learning. Comprehensive documentation fostered self-sufficiency and continuous learning.

By offering a seamless and empowering developer experience, LaunchPad increased adoption and accelerated feature delivery, becoming a critical asset in PETech's technology stack.

### **10.2.7 Exercise 10.4: Adopting a Platform-as-a-Product Mindset**

Objective: Develop a product vision and roadmap for your internal platform, incorporating user-centric design principles.

Here are the recommended steps for this exercise.

Conduct User Research:

- Coding Activity: Build a survey application using a web framework (e.g., Flask, Django, Express.js). Collect responses and store them in a database like SQLite or MongoDB.
- Data Analysis: Write scripts in Python or R to analyze survey data and extract critical insights.

Define the Platform's Vision and Mission:

- Summarize the findings from your analysis.
- Craft clear and concise vision and mission statements.

Develop a Product Roadmap:

- Modeling Activity: Use project management tools like Trello, Jira, or Azure DevOps to create a visual roadmap. Organize features into sprints or releases.

Create User Personas:

- Modeling Activity: Design personas using visual tools, including demographic information, goals, challenges, and preferred tools.

Design a Prototype or Mockup:

- Coding Activity: Develop an interactive prototype of a critical platform feature. Use frontend technologies like HTML, CSS, JavaScript, or frameworks like React or Angular. Alternatively, use prototyping tools like Figma or Adobe XD for non-coded mockups.

Gather Feedback:

- Present the prototype to stakeholders and users.

Expected deliverables for this exercise are:

- Platform Vision and Mission Statements in a documented format.
- Product Roadmap visualized with project management tools.
- User Personas with detailed descriptions and visuals.
- Interactive Coded Prototype of crucial platform features.
- Feedback Summary with action items and planned improvements

## 10.3 Cultural Shift from a Traditional Operations World

PETech's initial challenges stemmed not only from technical complexities but also from cultural and organizational silos. The traditional separation between development and operations hindered collaboration and slowed down processes. Recognizing that technology alone couldn't solve these issues, PETech understood that a cultural transformation was necessary to achieve its goals.

### 10.3.1 Embracing DevOps Cultural Principles

To address the challenges of slow deployments, siloed teams, and inefficient workflows, PETech embraced DevOps principles, fostering a culture of collaboration, automation, and shared responsibility. This shift aimed to break down barriers between teams, streamline processes, and create a more agile and responsive organization capable of rapidly delivering value to customers.

Key DevOps Practices Adopted

#### 1. Cross-Functional Teams

PETech reorganized its teams to be cross-functional, integrating developers, operations personnel, QA engineers, and security professionals into unified teams working towards common goals. This integration allowed for diverse perspectives and expertise within each team, promoting a shared understanding of the project's objectives and challenges.

- Shared Responsibility: Team members collectively owned the development, deployment, and maintenance of their services, reducing handoffs and delays.
- Improved Communication: Regular interactions among team members from different disciplines enhanced collaboration and reduced misunderstandings.
- Quicker Problem Resolution: With all relevant expertise within the team, issues could be identified and resolved more rapidly without involving external departments.

Example: The "Customer Account Team" at PETech included software developers, an operations specialist, a QA engineer, and a security analyst. This team was responsible for developing new features, ensuring code quality, deploying updates, and maintaining security compliance for the customer account services.

## 2. CI/CD

PETech implemented CI/CD pipelines to automate the build, test, and deployment processes. This automation reduced manual errors, accelerated release cycles, and ensured consistent team deployment practices.

- Continuous Integration (CI): Developers frequently merge their code, build it, and test it early to detect integration issues.
- Continuous Deployment (CD): Successful builds were automatically deployed to staging and, after passing necessary checks, to production environments, minimizing the time between code completion and deployment.
- Rapid Feedback Loop: Automated testing and deployment provided fast feedback to developers.

Example: Upon committing code changes to the repository, the CI/CD pipeline would automatically run unit tests, integration tests, and security scans before deploying to the staging environment.

## 3. Infrastructure as Code (IaC)

PETech adopted infrastructure-as-code practices using code and automation tools to manage its infrastructure configurations. This approach ensured consistency, repeatability, and version control for infrastructure provisioning and changes.

- Consistency and Repeatability: Infrastructure could be provisioned and configured consistently across different environments (development, staging, production), reducing configuration drift.
- Version Control: Infrastructure code was stored in version control systems, allowing teams to track changes, roll back if necessary, and collaborate effectively.
- Scalability and Disaster Recovery: Automated provisioning enabled rapid scaling of resources and efficient recovery in case of failures.

Tools Used: PETech utilized tools like Terraform and Ansible to define and manage its infrastructure. For instance, it defined its cloud resources (servers, databases, networking) in code, which could be deployed consistently across multiple environments.

## 4. Monitoring and Observability

Implementing robust monitoring and observability practices allowed PETech to gain real-time insights into system performance, availability, and health. Proactive monitoring helped detect issues early and improve system reliability.

- Real-Time Visibility: Dashboards and alerts provided immediate insights into key metrics such as CPU, memory, response times, and error rates.
- Proactive Issue Detection: Automated alerts were configured to notify teams of anomalies or thresholds being exceeded, enabling swift response to potential problems.

- End-to-End Tracing: Observability tools enabled tracing requests throughout the system, facilitating root-cause analysis of performance issues or failures.

Tools Used: PETech employed tools like Prometheus for metrics collection, Grafana for visualization, and the ELK Stack for log aggregation and analysis.

By embracing DevOps practices, PETech experienced significant improvements across its operations. Automation reduced the time from code commit to deployment from weeks to hours, allowing the company to respond quickly to market demands with faster deployment cycles. Continuous testing and monitoring led to the early detection of issues, reducing defects in production by 60% and markedly improving quality. Cross-functional teams fostered a culture of shared responsibility and teamwork, increasing developer satisfaction by 40% and enhancing collaboration. The organization became more agile and responsive, adapting swiftly to changing customer needs and technological advancements.

Adopting DevOps principles at PETech was not merely a procedural change but a fundamental transformation of the company's culture and operations. Breaking down silos between departments encouraged open communication and collaboration, empowering teams to make decisions and take ownership of their work—a significant cultural shift. Streamlined workflows eliminated bottlenecks and inefficiencies, while the integration of automation tools reduced manual intervention and errors, leading to process optimization. With the ability to deploy updates rapidly and reliably, PETech enhanced customer satisfaction and competitive positioning by delivering new features and improvements more frequently. For example, after implementing these practices, PETech successfully launched a new feature for their e-commerce platform ahead of schedule; the cross-functional team collaborated seamlessly, and the CI/CD pipeline ensured thorough testing and issue-free deployment. Real-time monitoring allowed for immediate observation of user interactions, and any minor issues were quickly addressed, resulting in positive customer feedback and increased sales.

PETech adopted DevOps principles and practices to transform its software development and delivery processes. The focus on collaboration, automation, and shared responsibility led to:

- Operational Excellence: Enhanced efficiency and effectiveness in delivering high-quality software.
- Innovation Enablement: Freed from the constraints of manual processes, teams could focus on innovation and strategic initiatives.
- Sustainable Growth: The ability to adapt quickly to market changes positioned PETech for continued success in a competitive industry.

This transformation serves as a model for other organizations seeking to overcome similar challenges and underscores the value of embracing DevOps principles to drive organizational success.

### 10.3.2 Breaking Down Silos with Team Topologies

PETech realized that the organization of their teams significantly impacted how effectively they collaborated and delivered value. To improve collaboration and efficiency, they restructured their teams using concepts from the **Team Topologies** framework, which we first introduced in Chapter 3, section 5. This approach helped them align their organizational structure with business goals and enhance team communication.

#### Team Types Implemented

1. PETech created Stream-Aligned Teams that focused on delivering end-to-end value for specific business domains. These teams were aligned with key business areas and had the autonomy to make decisions and move quickly. For example:
  - o E-Commerce Team: Responsible for the online shopping platform, including the user interface, shopping cart, and checkout process. They worked on features that directly impacted customer experience and sales.
  - o Mobile App Team: Focused on developing and maintaining PETech's mobile applications, ensuring a seamless experience across devices.
  - o Inventory Management Team: Handled the systems that tracked product stock levels, warehouse logistics, and supply chain integrations.

By aligning teams with specific business streams, PETech ensured that each team's work directly contributed to organizational goals. These teams owned their services from development to production, enabling them to respond rapidly to customer needs and market changes.
2. PETech established a dedicated Platform Team tasked with developing and maintaining the internal platform other teams used. The Platform Team reduced the cognitive load on the Stream-Aligned Teams by handling common infrastructure and tooling needs. Their responsibilities included:
  - o Developing Self-Service Tools: Creating tools allowing other teams to provide resources like databases, servers, and networking components without manual intervention.
  - o Managing CI/CD Pipelines: Maintaining continuous integration and deployment pipelines that standardized the build and release processes.
  - o Providing Shared Services: Offering services such as logging, monitoring, authentication, and authorization that could be reused across multiple teams.

For example, when the Stream-Aligned Teams needed to deploy their applications, they used the deployment pipelines provided by the Platform Team. This allowed them to focus on writing code and delivering features rather than dealing with the complexities of infrastructure management.

3. To support adopting new technologies and practices, PETech formed Enabling Teams. These teams provided expertise and guidance to other teams, helping them overcome obstacles and improve their capabilities. Their functions included:
  - Training and Workshops: Conducting sessions on DevOps practices, cloud migration strategies, and security best practices.
  - Consultation and Support: Working closely with teams adopting new technologies, offering hands-on assistance and advice.
  - Developing Standards and Guidelines: Creating documentation and templates to help teams follow best practices and maintain consistency.
 For instance, when PETech decided to migrate some of its services to the cloud, the Enabling Team specializing in cloud technologies assisted the Stream-Aligned Teams in understanding cloud architecture, selecting appropriate services, and implementing best practices for scalability and security.

### **Interaction Modes**

Teams are fine, but the real win is realized when a clear interaction and communication pattern is established. Let us now look at how PETech did these.

PETech defined explicit interaction modes to ensure effective collaboration between teams. These modes clarified how teams should work together in different scenarios.

1. Collaboration occurred when teams needed to work closely on complex projects or transitions. This mode was characterized by:
  - Joint Planning and Execution: Teams coordinated their efforts, shared responsibilities, and worked towards common goals.
  - Knowledge Sharing: Teams exchanged expertise and insights, learning from each other's experiences.
  - Problem-Solving Together: Collaborative efforts led to innovative solutions that might have yet to emerge in isolation.

Example: When PETech undertook a significant overhaul of its customer loyalty program, the E-Commerce, Mobile App, and Inventory Management Teams collaborated to ensure a seamless experience across all platforms. They held joint planning sessions, integrated their development efforts, and tested the new features collectively.

2. In the X-as-a-Service mode, the Platform Team provided services other teams could consume as needed. This approach streamlined interactions and clarified responsibilities:

- o Service Provisioning: Teams requested services through well-defined interfaces or APIs.
- o Clear Contracts and SLAs: The Platform Team established service-level agreements that specified performance expectations and support commitments.
- o Decoupling Dependencies: Teams could only use the services with an understanding of the underlying implementation details.

Example: If the Mobile App Team needed a new database instance, they could request it through the Platform Team's database-as-a-service offering. The Platform Team would handle provisioning, backups, and maintenance, allowing the Mobile App Team to focus on developing app features.

3. The Facilitation mode involved Enabling Teams to assist other teams in building their capabilities without doing the work for them:

- o Mentoring and Coaching: Enabling Teams provided guidance and support, helping teams develop new skills.
- o Temporary Support: They might be embedded with a team for a short period to help them become familiar with a new technology or practice.
- o Promoting Best Practices: Encouraging teams to adopt standards that improve efficiency and quality.

Example: When the E-Commerce Team wanted to implement automated testing for their applications, an Enabling Team specializing in test automation facilitated workshops and paired programming sessions. They helped the E-Commerce Team set up testing frameworks and write initial test cases, empowering them to continue independently afterward.

By restructuring their teams and defining explicit interaction modes, PETech achieved several benefits:

- Improved Focus and Efficiency: Stream-aligned teams could concentrate on delivering features and value to customers without being burdened by infrastructure concerns or needing more expertise in certain areas.
- Reduced Cognitive Load: The Platform and Enabling Teams took on specialized responsibilities, allowing other teams to work more efficiently and effectively.
- Enhanced Collaboration: Clear interaction modes minimized confusion and conflict between teams. Teams knew when and how to engage with others, leading to smoother workflows and better relationships.
- Accelerated Innovation: With empowered and supported teams, PETech brought new features and improvements to the market more quickly, responding to customer needs and staying ahead of competitors.

Let us look at a precise case study that brought it together at PETech.

### CASE STUDY: PETECH NEEDS FOR IMPROVING THE INVENTORY MANAGEMENT SYSTEM

To illustrate the application of the Team Topologies framework at PETech, let's consider a real-world scenario that showcases how different team types and interaction modes lead to successful outcomes.

#### **Challenge**

The *inventory management team* needed to implement real-time inventory level tracking to improve order fulfillment accuracy. This enhancement was critical to:

- Reduce stock discrepancies.
- Minimize order errors.
- Enhance overall customer satisfaction.

#### **Solution**

To address this challenge, PETech leveraged the strengths of various team types and defined interaction modes:

1. Collaboration
  - The Inventory Management Team collaborated with the Platform Team to integrate a new messaging system capable of handling real-time data streams.
  - This close cooperation allowed them to combine the Inventory Management Team's domain expertise with the Platform Team's technical proficiency in infrastructure and tooling.
2. Facilitation

- An Enabling Team with expertise in real-time data processing facilitated training sessions.
  - They helped the Inventory Management Team understand how to work with the new messaging system, transferring knowledge without taking over the project.
  - This empowerment enabled the Inventory Management Team to develop new capabilities and become more self-sufficient.
3. X-as-a-Service
- The Platform Team provided the messaging system as a service, handling its maintenance and scalability.
  - By consuming this service, the Inventory Management Team could focus on implementing business logic without worrying about underlying infrastructure complexities.
  - Clear boundaries and expectations streamlined interactions between the teams.

### **Outcome**

As a result of these coordinated efforts:

- The Inventory Management Team successfully implemented real-time inventory tracking.
- Order errors were reduced by 25%, leading to fewer customer complaints and returns.
- Customer satisfaction improved significantly, as accurate inventory levels ensured better product availability and timely order fulfillment.
- The company saw an increase in repeat purchases and positive customer reviews.

### **10.3.3 Exercise 10.5: Implementing Team Topologies in Your Organization**

Objective: Restructure team interactions based on Team Topologies to improve collaboration and flow efficiency.

Here are the recommended steps for this exercise.

1. Map Current Team Structures:
  - Modeling Activity: Create organizational charts and team interaction diagrams using tools like draw.io, Visio, or Miro. Highlight communication channels and dependencies.

- Coding Activity: Use network analysis libraries in Python (e.g., NetworkX) to model and visualize team communication patterns based on data from communication tools (e.g., Slack API).
2. Identify Appropriate Team Types:
- Analyze the data to determine optimal team structures.
  - Modeling Activity: Update diagrams to reflect proposed Stream-Aligned, Platform, Enabling, or Complicated-Subsystem teams.
3. Define Interaction Modes:
- Modeling Activity: Create sequence diagrams or flowcharts illustrating new interaction modes between teams.
  - Coding Activity: Simulate interactions using scripts or tools to model workflows (e.g., BPMN tools).
4. Propose a Restructured Team Topology:
- Modeling Activity: Present the new organizational structure using visual aids.
  - Coding Activity: Develop a simple application to showcase how teams interact with shared tools or platforms.
5. Develop a Transition Plan:
- Outline the steps and timelines
6. Pilot the Changes:
- Implement the new structure in a pilot area.
  - Coding Activity: Create collaborative coding projects on platforms like GitHub or GitLab to reflect new team interactions.
7. Collect Feedback and Iterate:
- Coding Activity: Use custom scripts to collect metrics on collaboration (e.g., pull requests and code reviews).
  - Analyze the effectiveness of the new topology and make adjustments.

The expected deliverables will be the following.

- Organizational charts of current and proposed structures.
- Communication Patterns Analysis using visualizations from coding activities.
- Interaction Diagrams showing new workflows.
- Simulated Interaction Models or Applications.

- Transition Plan with automated project management tasks.
- Pilot Implementation Report with data-driven insights.

#### **10.3.4 Impact on PETech's Culture**

This reorganization led to significant positive changes within PETech. Enhanced communication through open channels and regular interactions reduced misunderstandings and delays, while transparent communication built trust and aligned efforts across the organization. Stream-aligned teams were empowered with autonomy and ownership over their services, increasing motivation and accountability. This empowerment led to higher engagement and job satisfaction among team members. Additionally, reduced handoffs and streamlined processes resulted in faster delivery, accelerating time to market and allowing PETech to respond swiftly to customer needs and competitive pressures. The collaborative environment improved morale, increasing job satisfaction and retention as teams felt valued and connected to the organization's success.

By embracing a cultural shift alongside technical changes, PETech created a more resilient and adaptable organization capable of sustaining long-term growth and innovation. This combination of cultural transformation and strategic reorganization positioned the company to meet future challenges effectively, fostering an environment where continuous improvement and collaboration drive ongoing success.

#### **10.3.5 Exercise 10.6: Planning a Cultural Shift Towards DevOps and Collaboration**

Objective: Develop a strategic plan to initiate a cultural shift from a traditional operations model to a DevOps-oriented, collaborative culture within your organization. This includes modeling current processes and coding automation scripts to support new practices. Here are the recommended steps for this exercise.

1. Assess the Current Culture:

- Modeling Activity: Create process flow diagrams of your current software development lifecycle (SDLC) using tools like Visio, Lucidchart, or draw.io. Map out handoffs, bottlenecks, and feedback loops.
- Coding Activity: Develop scripts (e.g., using Python or Bash) to collect and analyze data on current deployment frequencies, lead times, and failure rates. Use APIs from tools like Jenkins, GitLab, or Jira to extract metrics.

2. Define the Desired Culture:

- Modeling Activity: Design future-state process models that incorporate DevOps practices. Highlight differences from the current state using annotations or overlays.
- Coding Activity: Prototype automation scripts for CI/CD pipelines using tools like Jenkinsfiles, GitLab CI/CD YAML configurations, or GitHub Actions workflows.

3. Develop a Change Management Plan:
  - Leadership Engagement: Present your models and findings to stakeholders using visual aids.
  - Communication Strategy: Create infographics or presentations illustrating the benefits of the cultural shift.
  - Coding Activity: Implement the new CI/CD pipelines in a controlled environment with selected teams. Build a simple web application or use a service like Google Forms to collect anonymous feedback. Write scripts to analyze feedback data.
  
4. Implement & Measure the Plan:
  - Rollout: Implement the initiatives according to the change management plan. Use the coded automation scripts in pilot teams to demonstrate the benefits.
  - Coding Activity: Re-run your data collection scripts to gather post-implementation metrics. Data visualization libraries like Matplotlib or D3.js can be used to compare before-and-after results.

The expected deliverables will be the following.

- Process Flow Diagrams of current and future SDLC processes.
- Automation Scripts for CI/CD pipelines and data collection, with documentation.
- Training Materials, including coded exercises or labs.
- Data Analysis Reports with visualizations of before-and-after metrics.
- Feedback Data is collected via coded tools or forms.
- Implementation Progress Report summarizing outcomes and lessons learned.

## 10.4 SRE Strategy, Models, and Aligning with Organizational Needs

### 10.4.1 Understanding Site Reliability Engineering (SRE)

As PETech increased deployment frequency, maintaining system reliability became critical to ensuring customer satisfaction and trust. Site Reliability Engineering provided a framework for balancing innovation with stability by applying software engineering practices to operations. Just to jog your memory, here are the three core considerations in Site Reliability Engineering.

**Service Level Objectives (SLOs):** Targeted performance and availability levels define the expected quality of service. SLOs provided clear goals for reliability that aligned with user expectations.

**Service Level Indicators (SLIs):** Metrics that measure compliance with SLOs, such as latency, throughput, and error rates, offer quantifiable data to assess system performance.

**Error Budgets:** Acceptable levels of risk to manage change velocity versus reliability. Error budgets allowed teams to balance the need for new features with maintaining system stability.

#### **10.4.2 Implementing SRE at PETech**

PETech embedded SRE practices into their teams to enhance reliability without stifling innovation. They began by defining Service Level Objectives (SLOs) and Service Level Indicators (SLIs) for critical services like payment processing. These clear objectives were established based on user needs and business priorities, guiding engineering efforts and resource allocation to ensure that vital services met performance and availability targets.

Additionally, PETech implemented advanced monitoring and alerting tools like Prometheus and Grafana to gain real-time insights and enable proactive issue detection. This allowed teams to identify and address problems before they impacted users. They also built automated remediation mechanisms to handle common failures, reducing manual intervention and downtime. This automation increased system resilience and recovery speed, contributing to a more robust and reliable overall system.

#### **10.4.3 Aligning SRE with Organizational Goals**

By aligning SRE practices with business objectives, PETech ensured several vital outcomes that benefitted the organization and its customers. Firstly, reliability metrics directly supported customer satisfaction. By maintaining high availability and optimal performance, PETech enhanced the user experience, which led to increased customer loyalty and a positive brand reputation. Customers enjoyed uninterrupted access to services, fostering trust and encouraging repeat business.

Secondly, error budgets became a vital tool for team decision-making. By balancing feature development with reliability improvements, teams could make strategic choices about when to prioritize system stability over the introduction of new functionalities. This approach ensured that innovation did not compromise reliability, allowing PETech to deliver new features without adversely affecting the user experience.

Lastly, PETech fostered a culture of continuous improvement. Regular reviews of performance against Service Level Objectives (SLOs) and Service Level Indicators (SLIs) led to process enhancements and system optimizations. This practice embedded a culture of learning and adaptation within the organization, where teams consistently sought ways to improve reliability and efficiency. By embracing this mindset, PETech was able to evolve with changing business needs while maintaining high levels of service reliability.

#### **10.4.4 Outcomes of implementing SRE**

Implementing Site Reliability Engineering (SRE) practices yielded significant benefits for PETech.

1. PETech experienced substantially reduced incidents, with fewer outages and performance issues, improving overall service quality. By proactively monitoring systems and addressing potential problems before they escalated, PETech minimized service disruptions. This reduction in incidents enhanced the customer experience and reduced maintenance costs associated with emergency fixes and downtime.

2. PETech achieved faster recovery times when incidents did occur. Quicker resolution minimized the impact on customers and internal operations, ensuring that service interruptions were brief and managed efficiently. Implementing efficient recovery processes, such as automated rollbacks and self-healing mechanisms, enhanced the system's confidence among users and stakeholders. Teams were better prepared to handle unexpected issues, strengthening the company's reputation for reliability.
3. The adoption of SRE practices enabled data-driven decision-making within the organization. Using metrics and performance data to guide prioritization and resource allocation, PETech ensured that efforts were focused where they mattered most. This data-driven approach improved transparency and accountability, as teams could see the impact of their work on key performance indicators. By integrating SRE into its operational strategy, PETech successfully balanced rapid innovation with dependable service delivery, aligning technological advancements with business objectives.

## **10.5 Intelligent Assistants to Help Enhance Engineering Platforms**

### **10.5.1 The Role of Intelligent Assistants**

Generative AI and automation can significantly enhance engineering platforms by automating routine tasks. This reduces manual effort and errors, freeing time for more complex and creative work. Automation improves efficiency and ensures consistency across processes so that the teams can focus on innovation and problem-solving. For example, automated testing and deployment pipelines speed up delivery cycles.

AI also provides valuable insights by analyzing vast amounts of data to offer recommendations. This enables teams to reduce their development times. AI-driven insights can reveal patterns and opportunities that might go unnoticed, helping teams optimize performance and anticipate future needs. Additionally, intelligent assistants can enhance communication by serving as interfaces between teams and tools, facilitating smoother interactions and collaboration. They centralize information and streamline workflows, making it easier for teams to access the needed resources and work together more effectively.

### **10.5.2 PETech's PETechBot**

To further support their developers, PETech developed "PETechBot," an intelligent assistant integrated with their platform. This AI-driven bot was designed to enhance the developer experience by automating routine tasks, providing real-time support, and streamlining workflows. By integrating PETechBot into their daily operations, developers gained a powerful tool that simplified their processes and empowered them to work more efficiently and effectively.

PETechBot had several key capabilities that transformed how developers interacted with the platform. One of its primary functions was deployment assistance, allowing developers to initiate deployments through simple chat commands. This feature greatly simplified the deployment process, reducing barriers and accelerating deployment cycles while minimizing errors. Additionally, PETechBot provided monitoring alerts by notifying teams of issues detected through log and metrics analysis. These proactive alerts ensured that teams could quickly address potential problems, minimizing their impact on production environments. PETechBot also offered knowledge base access, answering questions about platform usage, best practices, and troubleshooting. Acting as an always-available support resource reduced dependency on specific individuals and democratized knowledge across the team. Another valuable feature was code analysis, where PETechBot provided real-time feedback on code quality and potential security vulnerabilities. This immediate insight helped developers improve their code before it reached production, enhancing the applications' quality and security.

Implementing PETechBot has led to several significant advantages for development teams. Increased efficiency was one of the most notable benefits, as developers saved time on routine tasks, allowing them to focus on innovation and more complex problem-solving. This efficiency gain translated into higher overall productivity. Moreover, improved quality was achieved through the early detection of issues, which reduced defects and the need for rework. As a result, applications became more stable and reliable, strengthening customer trust. Lastly, PETechBot was a valuable training tool for new team members, facilitating onboarding and continuous skill development. By providing easy access to information and best practices, PETechBot supported a culture of learning and growth within PETech, enabling developers to improve their skills and knowledge continuously.

### **10.5.3 Exploring Advanced Tools**

PETech also explored a range of advanced tools to enhance its platform further, ensuring it remained at the forefront of technology trends and could meet evolving business needs. By incorporating these tools, PETech aimed to simplify complex processes, streamline resource management, and provide its developers with a more robust and flexible platform.

Crossplane was one of the critical tools PETech adopted to orchestrate applications and infrastructure across different environments. It provided a declarative API that allowed PETech to manage resources consistently, regardless of where they were hosted. With Crossplane, the team could define infrastructure requirements as code, enabling them to quickly replicate environments and apply changes uniformly across multiple clouds or on-premises systems. Its extensibility was particularly valuable, as it supported diverse environments and could scale with PETech's growing infrastructure needs. For example, if PETech needed to deploy a new service that required a combination of cloud services like databases, storage, and networking, Crossplane enabled them to do so with a unified configuration, ensuring consistency and reducing manual overhead.

In addition to Crossplane, PETech explored CNOE (Cloud Native Operating Environment) to provide a standardized environment for their cloud-native applications. CNOE offered a simplified deployment and management process across various cloud providers, enhancing portability and reducing vendor lock-in. By standardizing the environment, CNOE allowed PETech to deploy applications seamlessly on different cloud platforms without worrying about underlying infrastructure differences. This flexibility meant that PETech could choose the right CSP based on cost, performance, or specific service offerings without being tied to a single vendor. Moreover, CNOE's standardized environment helped streamline the development process, reducing additional cognitive load.

Humanitec was another tool PETech considered for building a flexible internal developer platform. Humanitec offered features like dynamic configuration management and resource orchestration as a platform orchestrator, providing higher automation and customization. With Humanitec, PETech could abstract away many of the complexities developers typically face when deploying and managing applications, such as configuring environments, managing dependencies, and scaling resources. This level of orchestration enabled PETech to provide developers with a more seamless and self-service platform experience. Developers could deploy their applications with a few clicks or API calls, knowing that the platform would handle the intricate details of infrastructure provisioning, resource allocation, and compliance with organizational standards.

By integrating these tools, PETech abstracted much of the complexity of modern cloud environments. These technologies provided developers with powerful capabilities without overwhelming them with the details of infrastructure management. As a result, PETech's platform remained adaptable and future-proof, capable of evolving alongside technological advancements and business demands. This strategic approach ensured the platform could support PETech's growth while maintaining high flexibility and resilience.

#### **10.5.4 Exercise 10.7: Integrating an Intelligent Assistant Into Your Platform**

Objective: Enhance your platform by developing or integrating an intelligent assistant (e.g., chatbot) to automate tasks and support developers.

The recommended steps for doing so are given below

1. Identify Use Cases:
  - o List potential functions the assistant could perform (e.g., deployment assistance, answering FAQs, monitoring alerts). Prioritize features based on impact and complexity.
2. Choose a Technology Platform:
  - o Select tools or frameworks for building the assistant (e.g., Botkit, Microsoft Bot Framework, Slack API). Decide on using APIs like Slack API, Microsoft Bot Framework, or custom solutions.
3. Develop a Minimum Viable Product (MVP):

- Implement core functionalities such as Responding to simple commands, Providing deployment status, and Fetching documentation links. You should scope it out to the most important activities you need in your organization. Code the assistant to handle core functionalities. Implement command parsing, API integrations with CI/CD tools, and response handling.
4. Integrate with Communication Channels:
- Use OAuth and webhooks to connect the assistant with platforms like Slack or Microsoft Teams. Ensure secure authentication and authorization.
5. Test and Refine:
- Write unit and integration tests. Consider using frameworks like PyTest or Mocha. Set up continuous integration to automatically run tests on code changes.
6. Plan for Advanced Features:
- Outline how AI capabilities (e.g., natural language processing, machine learning) could enhance the assistant. Integrate natural language processing using libraries like spaCy or TensorFlow.
  - Implement machine learning models for predictive analytics.
7. Document Security and Compliance Considerations:
- Implement input validation, error handling, and logging. Secure sensitive data to ensure compliance with data protection laws.

The expected deliverables for your exercise will be around the following

- A working, intelligent assistant integrated with your team's communication tool.
- Documentation of use cases and implemented features.
- User feedback and planned improvements.
- Security and compliance assessment.

## 10.6 Comparing Internal Developer Platforms and Developer Portals

### 10.6.1 Understanding Internal Developer Platforms (IDPs)

Internal Developer Platforms (IDPs), like PETech's LaunchPad, provide a crucial layer between developers and the underlying infrastructure, offering self-service capabilities and abstractions that simplify development and deployment. These platforms are characterized by self-service interfaces that empower developers to manage applications and environments independently, eliminating the need to wait for other teams and increasing autonomy and speed.

For example, suppose a developer needs to deploy a new microservice. In that case, they can use LaunchPad's self-service portal to configure and deploy it with a few clicks without involving the operations team. IDPs enforce standardization using templates and policies that promote best practices and compliance, ensuring consistency, standardization, and accuracy. Additionally, they emphasize automation, streamlining workflows with CI/CD integration and automated provisioning to minimize manual effort and errors. Once the developer sets up their service, LaunchPad automatically handles the build, testing, and deployment processes, allowing developers to focus more on the product domain-related features rather than dealing with infrastructure complexities.

The Internal Developer Platform (IDP) and an Engineering Platform (EP) share similarities in that they aim to improve developer efficiency and streamline the software development process. However, they differ in scope, purpose, and the specific problems they address within an organization.

IDPs primarily provide a layer of abstraction over the underlying infrastructure. They aim to empower developers with self-service capabilities to manage applications and environments quickly. IDPs are designed to streamline the development and deployment process by offering tools, interfaces, and automation that enable developers to provision resources, deploy code, and monitor applications without needing to understand the intricacies of the underlying infrastructure.

On the other hand, an engineering platform has a broader scope encompassing not just deployment and infrastructure management but also the entire software development lifecycle (SDLC). It enhances developer productivity, collaboration, and overall engineering efficiency. Engineering platforms often include capabilities such as source code management, build and test automation, security and compliance tooling, observability, and developer collaboration tools. The goal is to provide a comprehensive environment that supports all aspects of software engineering, from writing and testing code to monitoring and maintaining applications in production.

**Table 10.1 5 Key Features of an Internal Developer Platform (IDP). Reproduced with permission from InternalDeveloperPlatforms.Org**

| Key Features                         | Description                                                                         |
|--------------------------------------|-------------------------------------------------------------------------------------|
| Application Configuration Management | Manage application configuration in a dynamic, scalable, and reliable way           |
| Infrastructure Orchestration         | Orchestrate your infrastructure dynamically and intelligently                       |
| Environments                         | Ephemeral environments on demand                                                    |
| Deployment Management                | Implement a delivery pipeline for continuous delivery or even continuous deployment |
| RBAC                                 | Manage who can do what in a scalable way                                            |

### **10.6.2 Understanding Developer Portals**

Developer portals like Spotify's Backstage are a centralized hub where developers can access documentation, APIs, services, and tooling. These portals are designed with discoverability, making it easy for developers to find and use internal services and resources, enhancing efficiency and fostering collaboration across teams. They also focus on documentation and knowledge sharing, centralizing information to support learning, and promoting best practices in the organization. Moreover, developer portals often feature plugin ecosystems that allow for extended functionality through integrations with other tools. This enables organizations to customize the portal to fit their needs and scale its capabilities as they grow.

Portals like Backstage streamline workflows and improve the developer experience by providing an UX for developer resources. The benefits of developer portals are summarized in the table below.

**Table 10.2 Key capabilities of developer portals and how they might be used**

| What?                                       | How                                                                                        |
|---------------------------------------------|--------------------------------------------------------------------------------------------|
| Centralized Access to APIs & Services       | Discover new APIs and integrate them with their applications                               |
| Unified Documentation and Knowledge Sharing | A single source of truth that consolidates documentation and guidelines                    |
| Standardized Project Templates              | Quick and easy onboarding of a new service                                                 |
| Automated Service Provisioning              | Easier provisioning of cloud services in a controlled environment                          |
| Streamlining Collaboration                  | More accessible Code reviews, Team Communication by collaborating with a GitHub repository |
| Plugin Ecosystem for Customization          | Custom plugins used by multiple users are changed without changing user experience.        |

### 10.6.3 Comparing and Contrasting

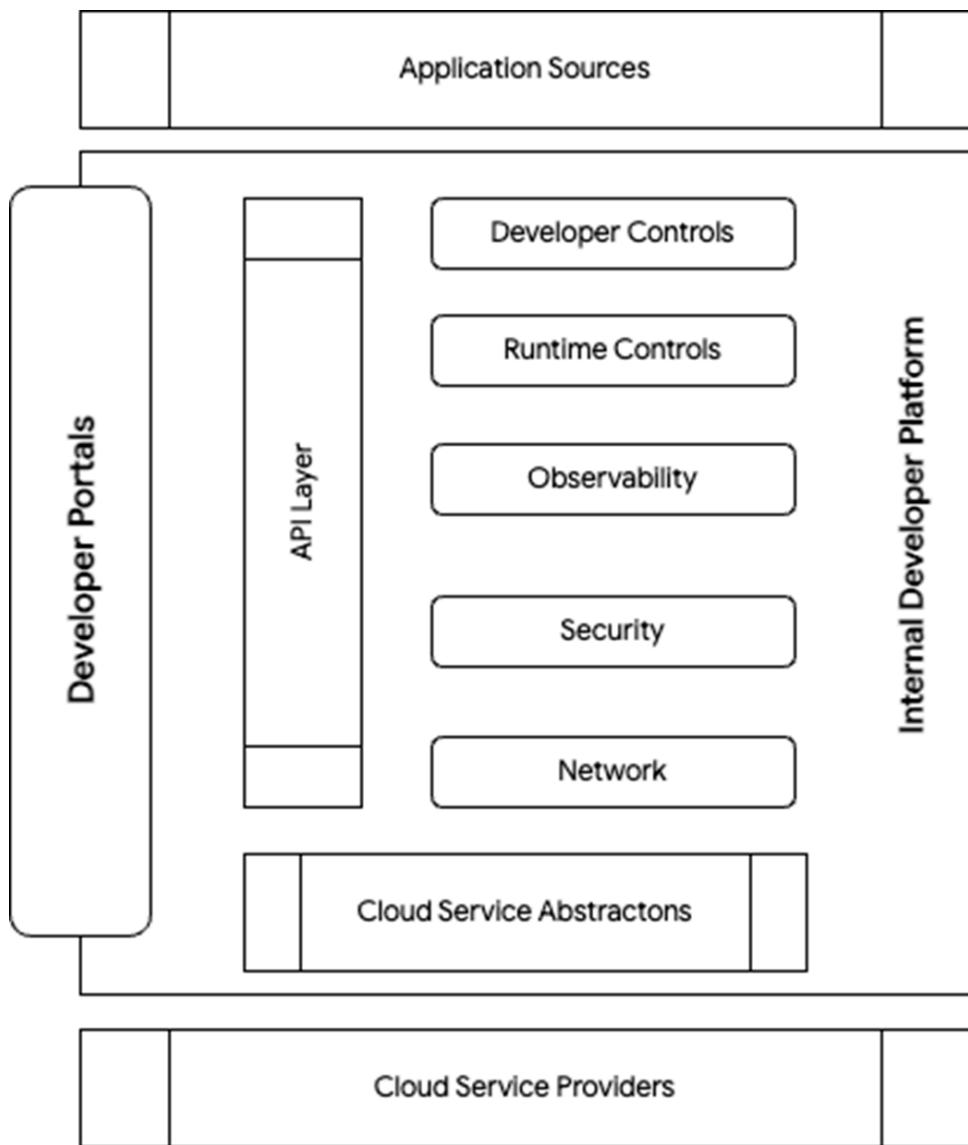
The interchangeable usage of the two terms—internal Developer Platforms and Developer Portals—indicates notable confusion surrounding their applicability in platform engineering.

**Table 10.3 Comparing an internal developer platform and a developer portal**

| Criteria              | IDP                                                                                                                                                                  | Dev Portal                                                                                                                                                               |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Purpose               | Emphasizes operational aspects, enabling deployment and management of applications with ease and consistency.                                                        | It focuses on information dissemination and collaboration, providing access to resources, and facilitating communication.                                                |
| Usage Pattern         | It provides interfaces for performing actions such as deploying code, provisioning resources, and managing environments, directly impacting the development process. | It serves as an information hub, often read-centric, offering access to documentation, APIs, tools, and resources to support developers in planning and decision-making. |
| Key Capabilities      | Self-service deployment<br>Automated provisioning<br>Infrastructure abstraction<br>CI/CD integration<br>Environment management                                       | Centralized documentation<br>API catalog<br>Knowledge base<br>Collaboration tools<br>Project templates                                                                   |
| Delivery Workflow     | Directly streamlines and automates the development process, reducing manual tasks and speeding up deployment cycles.                                                 | It supports planning and decision-making.                                                                                                                                |
| Developer Workflow    | Directly impacts the development workflow by automating repetitive tasks, enabling faster deployments, and managing application lifecycles.                          | It supports the developer workflow by providing the necessary information and context, aiding in planning, decision-making, and learning processes.                      |
| Focus                 | Operational efficiency and developer productivity by simplifying deployment, resource management, and infrastructure complexities.                                   | Knowledge sharing, discoverability, and collaboration by centralizing documentation, APIs, and tools for developers.                                                     |
| Organizational Impact | It offers actionability, allowing developers to deploy, configure, and manage applications effectively.                                                              | It acts as an access point to the IDP's capabilities, providing information and context about the actions available within the IDP.                                      |

|                        |                                                                                                                                                             |                                                                                                                                                                                 |
|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Collaboration          | Supports collaboration through shared tooling for deployment and operations, allowing cross-functional teams to work well together                          | Facilitates collaboration by providing a centralized clearing house for documentation, knowledge sharing, and communication among developers.                                   |
| Developer Experience   | Enhances the developer experience by reducing complexity and providing self-service capabilities, making it easier to manage applications and environments. | Enhances the developer experience by centralizing resources and reducing the effort needed to find information, promoting best practices and consistency.                       |
| Integrations           | Often integrates with CI/CD pipelines, monitoring tools, and infrastructure providers to automate and streamline operations.                                | Integrates with various tools and platforms, such as source code repositories, API gateways, and project management systems, to provide a holistic view of available resources. |
| Context Switching      | Minimizes context switching by providing an integrated deployment and resource management platform.                                                         | Reduces context switching by serving as a single access point for documentation, tools, and resources, enhancing discoverability.                                               |
| Outcome                | Streamlined deployment processes, reduced infrastructure complexity, and accelerated delivery cycles.                                                       | Improved knowledge sharing, better documentation accessibility, and enhanced collaboration across teams.                                                                        |
| Example Use Cases      | A developer uses the IDP to deploy a new microservice, automatically provisioning the necessary infrastructure and setting up monitoring.                   | A developer accesses the portal to find the API documentation and best practices for integrating a new feature into the microservice.                                           |
| Example Industry Logos | Humanitec, Crossplane, Kratix                                                                                                                               | Backstage, Azure DevOps                                                                                                                                                         |

In Figure 10.5, we provide a simplistic view of how internal developer platforms (IDPs) and Developer Portals interact.



**Figure 10.5 A simplistic view of how Internal Developer Platforms and Developer Portals interact with each other in a platform-engineering-centric organizational construct**

PETech integrated its LaunchPad IDP with a developer portal to create a unified platform that maximized the benefits of both systems. This integration provided a unified interface, allowing developers to access deployment tools and documentation from a single location, simplifying navigation and usage. It also enhanced collaboration by incorporating forums and knowledge bases, facilitating team interactions, and promoting a culture of shared learning and community support. By combining these resources, PETech streamlined workflows and minimized context switching, reducing cognitive load and enabling developers to focus on delivering value more efficiently. This cohesive ecosystem supported developers throughout the entire software development lifecycle, driving productivity and innovation.

#### **10.6.4 Exercise 10.8: Comparing and Integrating IDPs and Developer Portals**

Objective: Analyze the differences between Internal Developer Platforms (IDPs) and developer portals and develop a plan to integrate them for an improved developer experience, including building a coded Proof of Concept (PoC).

The recommended steps for this exercise are as follows

1. Research Existing Solutions:
  - o Coding Activity: Set up local instances of open-source developer portals like Backstage. Explore their codebases to understand architecture and extensibility.
  - o Modeling Activity: Create comparison matrices highlighting features and capabilities.
2. Assess Your Current Environment:
  - o Modeling Activity: Diagram your current tools and workflows using UML diagrams.
  - o Coding Activity: Write scripts to extract metadata from existing tools (e.g., services registered in a service registry).
3. Identify Gaps and Overlaps:
  - o Analyze where functionalities overlap or are missing.
  - o Coding Activity: Develop small scripts to test interoperability between tools via APIs.
4. Develop Integration Strategies:
  - o Modeling Activity: Design system architecture diagrams showing integrated components.
  - o Coding Activity: Plan API endpoints or middleware services needed for integration.
5. Create a Proof of Concept (PoC):

- Coding Activity: Build a developer portal using Backstage, integrating plugins for your existing CI/CD pipelines, monitoring tools, and documentation. Develop custom plugins, if necessary, to connect to proprietary systems.
6. Gather Developer Feedback:
- Coding Activity: Implement user analytics within the portal to track usage patterns. Use feedback forms or chatbots within the portal to collect qualitative feedback.
7. Plan for Full Integration:
- Modeling Activity: Create detailed architectural and data flow diagrams for the full integration.
  - Coding Activity: Outline development tasks, resource requirements, and timelines using project management tools.

We expect the eventual deliverables to be as follows

- Comparative Analysis Report with findings from hands-on exploration.
- Diagrams of current and proposed integrated environments.
- Source Code of the PoC with installation and usage documentation.
- Developer Feedback Summary, including usage analytics and survey results.
- Integration Plan detailing coding tasks, dependencies, and schedules.

## 10.7 Summary of Building the Platform Products

PETech's experience offers valuable lessons for organizations seeking to evolve their platform products. Each aspect of their journey highlights critical strategies and best practices that can drive successful platform evolution.

### 10.7.1 Key Takeaways from PETech's Journey

#### **ADOPTING A PLATFORM-AS-A-PRODUCT MINDSET**

One of the foundational steps in PETech's transformation was embracing the platform-as-a-product mindset. Instead of viewing the platform merely as a set of tools or infrastructure components, PETech treated it as a product designed to meet its users' specific needs—the organization's developers and operators. This shift in perspective meant prioritizing user-centric design and focusing on delivering real value.

The platform team gained deep insights into their users' challenges and requirements by conducting thorough user research. They defined the platform's clear vision and mission, aligning it with user needs and business objectives. Based on this understanding, features and enhancements were planned, ensuring each addition addressed genuine pain points and improved the developer experience.

This approach drove adoption and satisfaction because users felt heard and saw tangible workflow improvements. Developers could focus more on writing code and delivering features rather than grappling with infrastructure complexities. Treating the platform as a product led to a more engaged user base and a platform that evolved in step with the organization's needs.

## MEASURING SUCCESS EFFECTIVELY

PETech recognized that they needed to measure success effectively to ensure the platform's impact was tangible and to guide continuous improvement. They established key performance indicators (KPIs) aligned with business goals, providing objective data to assess the platform's performance.

By integrating both lagging and leading metrics, including the four key metrics from the DORA framework (deployment frequency, lead time for changes, change failure rate, and MTTR), PETech could track operational efficiency and reliability. They complemented these with leading indicators like code quality metrics, test coverage, and developer satisfaction scores.

Regularly reviewing these metrics enabled the platform team to promptly identify areas of strength and opportunities for improvement. This data-driven approach ensured that efforts were focused where they would have the most significant impact.

## EMBRACING CULTURAL CHANGE

Understanding that technology changes alone were insufficient, PETech focused on embracing cultural change. They broke down silos, enhancing collaboration.

By adopting DevOps principles and reorganizing teams based on concepts from Team Topologies, PETech accelerated delivery and innovation. Cross-functional teams were formed, integrating developers, operations, QA, and security professionals. This restructuring improved communication, reduced misunderstandings, and promoted a sense of ownership among team members.

The cultural shift also involved promoting transparency and trust, encouraging teams to share knowledge and support each other. This collaborative environment led to a more agile organization capable of adapting quickly to changing market demands and technological advancements.

## IMPLEMENTING SRE PRACTICES

To balance rapid innovation with the need for reliable systems, PETech implemented Site Reliability Engineering (SRE) practices. They defined clear Service Level Objectives (SLOs) and monitored Service Level Indicators (SLIs) to set measurable system performance and availability targets.

PETech managed the trade-off between deploying new features and maintaining system stability by establishing error budgets. If the error budget was exhausted due to incidents, efforts were shifted towards improving reliability before introducing additional changes.

Embedding SRE practices enhanced customer trust by delivering consistent and dependable services. It also improved operational stability by proactively identifying and addressing potential issues before they impacted users. This approach ensured that reliability was not an afterthought but an integral part of the development and deployment process.

## LEVERAGING INTELLIGENT TOOLS

PETech recognized the value of technology in enhancing efficiency and quality. By leveraging intelligent tools like automation and AI, they reduced complexity and improved the overall quality of their software delivery.

Automation tools streamlined repetitive tasks such as testing, deployment, and infrastructure provisioning, increasing speed and minimizing human errors associated with manual processes. AI-driven monitoring and analytics provided more profound insights into system performance, enabling proactive issue detection and resolution.

These intelligent tools freed teams to focus on strategic initiatives and innovation rather than getting bogged down in routine operations. The result was a more efficient organization capable of delivering higher-quality products in less time.

## INTEGRATING IDPS AND DEVELOPER PORTALS

To optimize the developer experience, PETech integrated Internal Developer Platforms (IDPs) and developer portals. By combining operational capabilities with accessible information resources, they created a unified platform that empowered developers.

The IDP provided self-service capabilities for deploying applications and managing environments, reducing dependency on other teams and accelerating the development process. The developer portal offered a centralized hub for documentation, APIs, tooling, and collaboration resources.

This integration minimized context switching and made it easier for developers to find what they needed when needed. As a result, developers could focus on building features rather than navigating complex systems or searching for information, enhancing productivity and satisfaction.

### 10.7.2 Benefits Realized by PETech

Through their comprehensive approach, PETech achieved significant and quantifiable outcomes that propelled the organization forward:

#### Accelerated Time-to-Market

- Deployment Frequency Increased by 400%: PETech increased its deployment frequency from an average of once every two weeks to multiple daily deployments. This shift allowed the company to respond swiftly to market demands and customer feedback.
- Lead Time for Changes Reduced by 85%: The time from code commit to production release was reduced from an average of 14 days to just two days. This drastic reduction enabled faster delivery of new features and fixes, giving PETech a competitive edge.

#### Improved Reliability

- System Uptime Improved from 98% to 99.97%: PETech reduced downtime dramatically by implementing SRE practices and enhancing monitoring, leading to higher service availability.

- Incident Rate Decreased by 60%: The number of critical incidents per quarter dropped, enhancing stability and building customer confidence.
- Mean Time to Recovery (MTTR) Reduced by 70%: MTTR decreased from 10 hours to 3 hours, minimizing the impact of outages and reducing costs associated with downtime by an estimated \$500,000 annually.

#### Increased Developer Productivity

- Developer Satisfaction Scores Increased by 40%: Surveys showed a significant rise in developer morale and job satisfaction due to streamlined processes and better tools.
- 50% Reduction in Manual Tasks: Automation eliminated repetitive tasks, allowing developers to focus more on innovation and feature development.
- 30% More Features Delivered Per Quarter: Empowered teams increased their output, driving innovation and efficiency within the organization.

#### Enhanced Competitive Advantage

- Customer Satisfaction Scores Improved by 25%: Enhanced reliability and faster feature releases led to higher customer satisfaction ratings.
- Market Share Increased by 15%: PETech attracted new customers, leveraging its operational excellence to differentiate itself.
- Talent Acquisition Improved by 35%: The company's reputation for innovation and efficiency attracted top talent, increasing successful hires for engineering roles.

These quantifiable improvements demonstrate how PETech's comprehensive approach to evolving its platform product resulted in tangible benefits:

- Financial Gains: Increased efficiency and reduced downtime led to cost savings estimated at \$1 million annually, while faster time-to-market contributed to revenue growth.
- Strategic Advantages: The ability to innovate rapidly and reliably positioned PETech ahead of competitors, allowing them to capture new market opportunities.
- Cultural Impact: A collaborative and empowered workforce fostered a culture of continuous improvement and excellence, further propelling the company's success.

As PETech continues to evolve, it recognizes that platform development is an ongoing journey. Continuous improvement, adapting to new technologies, and aligning with organizational goals are essential for sustained success. PETech remains committed to investing in its platform, culture, and people to meet future challenges and opportunities.

As you build your engineering platforms as products, we recommend that you look at ways in which you should create a plan for the benefits you will be able to accomplish.

## 10.8 Summary

- Assess Your Current Platform Strategy: Identify areas where a platform-as-a-product approach can add value, considering both technical capabilities and user needs.
- Establish Clear Metrics: Define KPIs aligned with your business objectives to measure success and guide decision-making.
- Foster a Collaborative Culture: Break down silos and encourage cross-functional teamwork through initiatives like DevOps and team topologies.
- Invest in Automation and Intelligent Tools: Leverage technologies that enhance efficiency and quality, staying abreast of industry advancements.
- Integrate IDPs and Developer Portals: Combine operational tools with information resources to provide a seamless and empowering developer experience.
- Embrace Continuous Improvement: Regularly revisit and refine your platform strategy, remaining adaptable to change and committed to growth.

# ***11 Generative AI in Platform Engineering***

## **This chapter covers**

- Impact of Generative AI (Gen AI) and Large Language Models (LLMs) in platform engineering
- Measuring improvements in platform engineering through AI methodologies
- How to improve developer experience through Generative AI
- How is Generative AI improving the observability space?
- What are the new frontiers in platform engineering within IDPs and overall developer experience where we should expect more

Gen AI and LLMs are reshaping platform engineering, making it possible to generate complete or partial solutions with well-crafted prompts. This capability is precious in streamlining complex decision-making processes, such as selecting optimal runtime environments or identifying the most suitable architectural patterns. In traditional platform engineering, predictive AI has been used to simplify decision-making by using decision trees and models to hone in on specific approaches. GenAI extends this capability by providing context-aware solutions tailored to an organization's unique needs.

As we have done in the past chapters, we'll continue to apply the principles we discuss to PETech, using specific examples.

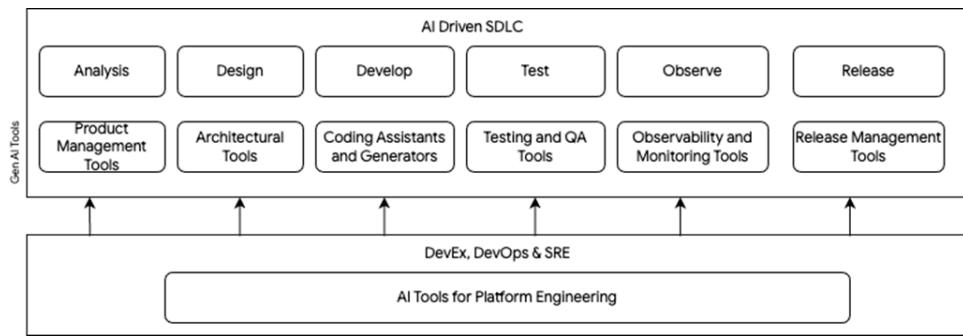
## 11.1 Impact of GenAI and LLMs in Platform Engineering

As PETech continued its Platform Engineering journey, it encountered the complexities inherent in optimizing cloud infrastructure due to many options available, such as virtual machines, serverless, and Kubernetes. Each option presented cost, performance, scalability, and operational overhead trade-offs. Historically, selecting the optimal infrastructure often requires extensive manual analysis and experimentation, leading to inefficiencies and suboptimal resource utilization. In the case of PETech, it used Gen AI to streamline and optimize its cloud infrastructure decision-making process to address this challenge.

By implementing a GenAI model trained on PETech's specific application patterns, historical usage data, and infrastructure requirements, they developed a solution that could dynamically predict the best runtime configurations for each application. This model leveraged the Large Language Model (LLM), which generated insights to evaluate various infrastructure choices against multiple criteria, such as cost efficiency, performance benchmarks, scalability potential, and operational simplicity. Drawing from concepts highlighted in recent research, including the 2024 advancements in GenAI for cloud optimization, PETech's approach included using Retrieval-Augmented Generation (RAG) techniques to retrieve real-time information on cloud services and pricing. This contextual awareness enabled the model to recommend the most cost-effective and high-performing infrastructure setup tailored to each application's unique workload characteristics. As a result, PETech achieved a 30% reduction in infrastructure costs and a 40% improvement in application performance.

GenAI's impact on PETech extended beyond infrastructure optimization, influencing various stages of the Software Development Lifecycle (SDLC) by automating and enhancing traditionally manual processes. According to the latest industry findings, GenAI's ability to process and generate natural language has revolutionized requirements generation. LLMs can automatically translate user inputs, business goals, and functional requirements into well-defined user stories and technical specifications. PETech leveraged this capability to streamline the requirements-gathering process. By inputting raw customer feedback and high-level business objectives into their GenAI system, they generated comprehensive user stories and acceptance criteria, drastically reducing the time spent on manual documentation and improving alignment with stakeholder needs.

In the architectural design phase, GenAI demonstrated its ability to enhance design quality and reduce time-to-market. Modern research has shown that when trained on architectural patterns and best practices, LLMs can generate detailed architectural diagrams, data models, and design blueprints. PETech integrated GenAI into their design workflow to automatically create data flow diagrams, ERDs, and system architectures based on project requirements. For example, when embarking on a new microservices-based project, PETech's GenAI model analyzed the desired system functionality and automatically proposed an architecture that included service boundaries, API endpoints, data storage solutions, and security considerations. This expedited the design process and improved the usability and maintainability of the resulting product by ensuring consistency with industry best practices. Figure 11.1 shows a high level view of SDLC process and where the GenAI tooling will fit in.



**Figure 11.1 A high-level view of AI-driven SDLC that improves Developer Experience through DevOps principles, making it easier to do Site Reliability Engineering by applying platform engineering principles**

The development phase benefited significantly from GenAI's code generation and review capabilities. Using LLMs trained on PETech's codebase, open-source repositories, and industry-standard coding practices, PETech's developers could automatically generate boilerplate code and scaffolding for new projects. According to the latest literature in platform engineering, AI-augmented coding tools have proven effective in enhancing developer productivity and reducing coding errors. PETech saw this firsthand as their GenAI model provided real-time code suggestions, reducing the amount of boilerplate code developers needed to write manually. Furthermore, the GenAI system conducted smart code reviews, offering immediate feedback on code quality, potential bugs, and adherence to coding standards. This automated review process accelerated code quality assurance, reducing the time to identify and fix issues by 50% and enabling developers to focus on more complex problem-solving tasks.

GenAI also played a pivotal role in modernizing PETech's legacy codebases, a challenge common in large enterprises. Refactoring monolithic architectures into more modular and maintainable structures is typically time-consuming and error-prone. However, using LLMs trained on architectural refactoring patterns and best practices, PETech's GenAI system could analyze the legacy code, identify tightly coupled modules, and recommend decomposition strategies. This enabled the platform engineering team to break down the monolithic codebase into manageable microservices, facilitating smoother deployments, easier maintenance, and more efficient resource utilization. Recent platform engineering research emphasized that AI-driven code refactoring capabilities are crucial for organizations seeking to modernize their technology stack without disrupting current production systems.

In addition, PETech adopted GenAI's advanced capabilities for streamlining operational aspects like continuous integration and delivery (CI/CD). GenAI tools were integrated into their CI/CD pipelines to automate the generation of test cases, execute performance benchmarks, and monitor system health. For instance, LLMs could identify gaps in test coverage by analyzing the codebase and suggesting missing unit tests, reducing the likelihood of defects slipping into production. PETech's operational teams also leveraged GenAI for predictive scaling and anomaly detection, using AI-generated insights to anticipate traffic spikes and adjust resource allocation in real-time, thereby maintaining system performance and reliability.

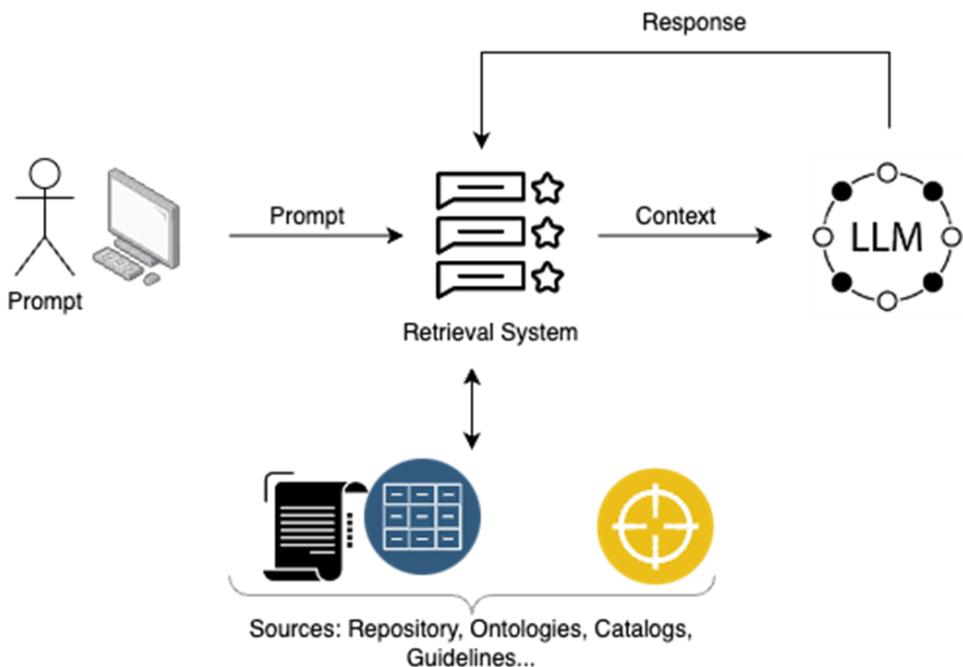
The advancements at PETech illustrate the profound impact of GenAI in platform engineering, enabling organizations to optimize infrastructure, streamline SDLC processes, and enhance overall efficiency. By incorporating GenAI into its platform engineering strategy, PETech improved its operational metrics and established a scalable, adaptable platform that can evolve with the changing technological landscape. The convergence of GenAI and platform engineering, as seen through PETech's journey and supported by the latest research, underscores the potential of AI-driven methodologies in transforming the future of software development and infrastructure management.

### **11.1.1 Enhancing LLMs with Retrieval-Augmented Generation (RAG)**

One of the significant challenges in applying LLMs within platform engineering is ensuring that the generated solutions are contextually relevant and accurate to the organization's specific needs. LLMs are powerful; their responses are sometimes generic or lack the contextual nuances needed for complex engineering tasks.

Retrieval-augmented generation (RAG) addresses this challenge by enabling LLMs to retrieve information from external sources, such as knowledge bases, documentation, ontologies, and guidelines, enhancing generated responses' quality and contextual relevance. This approach ensures that AI-generated solutions are accurate and aligned with the organization's specific standards, best practices, and operational requirements.

RAG can significantly improve the decision-making process in platform engineering by incorporating domain-specific knowledge into the LLM's output. For instance, PETech integrated RAG into their GenAI models to enhance the quality of generated solutions. They created an ecosystem of internal knowledge repositories, including documentation on architectural patterns, design principles, coding standards, deployment guidelines, and operational playbooks. When an LLM is queried for generating architecture diagrams, code suggestions, or design documentation, the RAG mechanism retrieves relevant information from these repositories, ensuring that the generated solutions are precise and contextually aligned with PETech's established practices. This integration of RAG into the GenAI model allowed PETech to tailor outputs more accurately, improving the precision of automated code reviews and reducing false positives in anomaly detection. In Figure 11.2, you can see how the prompts are better qualified using RAGs.



**Figure 11.2 A simple view of how Retrieval Augmented Generation works within a prompting process**

A crucial aspect of RAG's effectiveness in platform engineering is its ability to incorporate ontologies and structured data into retrieval. Ontologies define the relationships between concepts within a domain and provide a structured framework for understanding the LLM's context. By leveraging ontologies, PETech's RAG-enabled LLMs could understand and classify domain-specific terms, technologies, and practices, allowing the models to generate more sophisticated and contextually aware responses. For example, when generating a solution for microservices architecture, the LLM could use ontology to understand the relationships between services, communication protocols, and deployment strategies, leading to more coherent and applicable architectural diagrams and guidelines.

Furthermore, RAG can access a wide range of documentation and guidelines that drive the quality of generated results. This includes developer documentation, API references, compliance guidelines, and best practice manuals. At PETech, the RAG mechanism was designed to pull from an extensive repository of internal and external documentation sources. For example, when developers needed to integrate a new cloud service, the RAG-enabled LLM would retrieve relevant API documentation, cloud provider best practices, and PETech's internal guidelines for cloud adoption. This ensured that the generated code snippets, deployment scripts, and configuration files adhered to industry standards and PETech's internal policies, reducing the risk of misconfigurations and security vulnerabilities.

RAG's ability to enhance the LLMs' training process is another critical advantage. By narrowing the information space to contextually appropriate data, RAG ensures that LLMs focus on high-quality sources during training and inference. This refinement process leads to more consistent and reliable outcomes. For instance, rather than training the LLM on a vast, unstructured dataset, PETech curated a collection of high-quality documents, including design documents, post-mortem analyses, and operational runbooks. These documents were then used as part of the retrieval process to inform the LLM's outputs. As a result, the LLM was able to generate solutions that were accurate and reflective of PETech's accumulated knowledge and experience, enhancing the overall quality and relevance of the AI-generated insights.

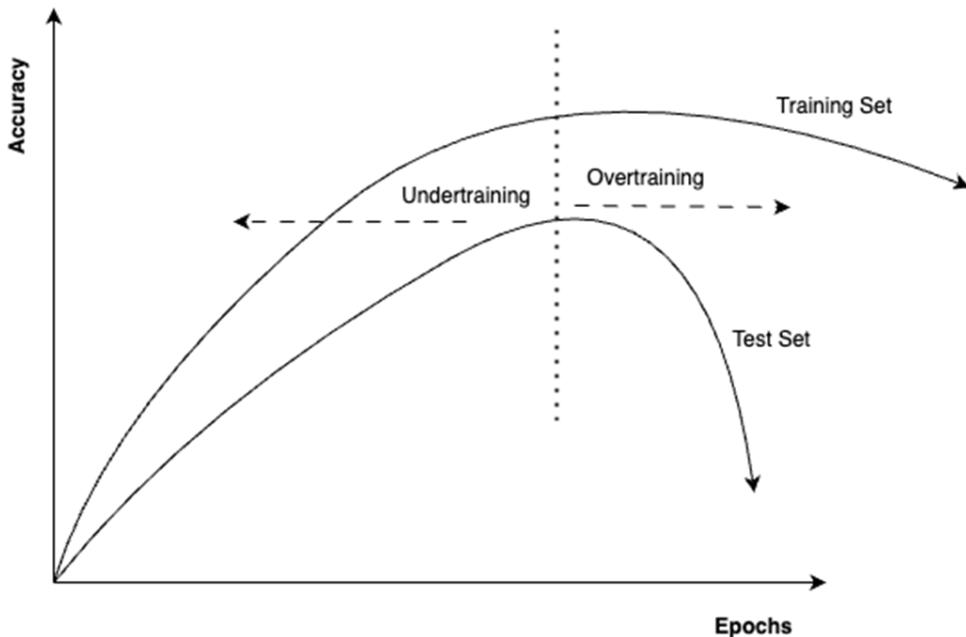
Organizations can leverage LLMs more effectively by incorporating RAG into platform engineering workflows, generating innovative and practical solutions. RAG enables the creating of an intelligent retrieval system that continually evolves as new knowledge is added to the repositories. PETech's GenAI models were always up-to-date with the latest architectural patterns, coding standards, and operational guidelines. When a new microservices pattern was introduced, or compliance requirements changed, the RAG mechanism ensured that this new information was available for retrieval, thereby maintaining the relevance and accuracy of the LLM's outputs.

### **11.1.2 Training and Epochs for Improving Quality of Results**

The quality of GenAI-generated solutions is highly dependent on the training process and the data used. Fine-tuning LLMs for platform engineering tasks requires carefully curated, domain-specific datasets that include code repositories, architectural blueprints, infrastructure configurations, and operational logs. However, the challenge lies in finding the right balance during training, particularly when determining the number of training epochs and using transfer learning effectively. Epochs refer to the number of times the learning algorithm processes the entire training dataset during the training phase. Each epoch represents one complete pass through all the training examples, allowing the model to learn and adjust based on the available data. Both overtraining and undertraining the models can lead to suboptimal performance, so it is crucial to identify the correct training parameters that yield accurate, relevant, and generalizable solutions.

Overtraining, or overfitting, occurs when a model becomes too specialized in the training data, capturing noise and specific patterns that may not generalize to new data. In the context of platform engineering, an overtrained model might produce solutions that work perfectly within the confines of the training dataset but fail to adapt to slightly different scenarios or new project requirements. For example, if an LLM is overtrained on a specific codebase, it might generate code overly tailored to its conventions, ignoring alternative methods or industry best practices. Conversely, undertraining results in a model that is not sufficiently learned to capture the nuances of the domain. An undertrained model may provide generic or incomplete solutions that lack the depth and specificity needed for complex platform engineering tasks, leading to poor developer experience and increased manual intervention.

Finding the correct number of training epochs is crucial to avoid overfit and undertraining. This process involves running the model through multiple training iterations (epochs) and evaluating its performance on a validation set after each epoch. Organizations can use techniques such as early stopping, where training is halted once the model's performance on a validation set degrades. This indicates that it might be beginning to overfit the training data. Cross-validation can also help determine the correct number of epochs by assessing how well the model generalizes to unseen data. PETech implemented these techniques by carefully monitoring metrics like loss, accuracy, and the ability of the model to generalize solutions across varied platform engineering scenarios. In Figure 11.3, you can see a conceptual model training approach to identify the number of epochs.



**Figure 11.3 shows the conceptual model training approach needed to identify the ideal number of epochs. The Training Set is used to train the ML model, while the Test Set is used to evaluate the performance of the trained model**

Now, let us look at PETech's training approach, as this can help you decide how you might want to approach this for your organization. PETech adopted a rigorous and iterative training methodology for their GenAI models, using a large corpus of proprietary codebase, architectural patterns, and operational data. They began with transfer learning, utilizing pre-trained LLMs already familiar with general programming and software engineering concepts. This provided a solid foundation, allowing the models to leverage existing knowledge before being fine-tuned on PETech's domain-specific datasets. They then conducted several training epochs to refine the models, gradually adjusting hyperparameters such as learning rate and batch size. They incorporated regularization techniques to prevent overfitting and monitored performance using a separate validation dataset that reflected real-world platform engineering scenarios not seen during training.

PETech's approach highlights how organizations can conduct practical GenAI training to enhance the developer experience. By iteratively refining the models and incorporating regular feedback loops, they ensured that the LLMs were proficient in generating accurate solutions and adaptable to new challenges. This training process can be replicated by other organizations, regardless of their size or domain. The key is to start with a diverse and representative dataset that reflects the organization's unique platform engineering needs. This may include internal codebases, architectural guidelines, and infrastructure configuration files. Organizations should also establish a working evaluation framework, using metrics that align with their specific goals, such as code quality, performance optimization, and compliance with internal standards.

Moreover, organizations can improve the developer experience by integrating these trained models into their development workflows. A well-trained GenAI model can automate code generation, conduct smart code reviews, and provide real-time feedback, reducing the cognitive load on developers. For example, if a developer writes a microservice, the model can suggest best practices based on the architectural blueprints it was trained on, ensuring that the code aligns with the organization's standards. Additionally, by fine-tuning the model continuously as new data becomes available, organizations can ensure that the GenAI solutions remain relevant and practical, adapting to evolving requirements and technological advancements.

The success of GenAI in platform engineering hinges on the training process, which must be meticulously managed to avoid overfitting and underfitting. By carefully selecting the correct training data, finding the optimal number of epochs, and employing transfer learning, organizations like PETech can develop LLMs that provide contextually accurate and valuable solutions. This enhances the platform engineering process and significantly improves the developer experience by automating complex tasks, providing intelligent recommendations, and ensuring consistency with organizational practices.

### **11.1.3 Sustainability Impact of LLM Training**

Epoch training and overfitting in large language models (LLMs) and generative AI (GenAI) have significant sustainability and carbon emission implications. Training these models requires vast computational resources, often involving thousands of GPUs or TPUs running for days, weeks, or even months. Each additional epoch—the complete pass over the training data—consumes more electricity, and when overfitting occurs, the extra computational expense does not translate to meaningful improvements in model performance. For example, training OpenAI's GPT-3 reportedly required hundreds of petaflop/s-days of computational power, translating into substantial energy consumption. A study by the University of Massachusetts Amherst estimated that training a single large AI model could emit as much carbon as five cars over their lifetimes. This highlights the environmental cost of excessive model training, especially when models are trained past the point of optimal performance.

The broader GenAI ecosystem, including using LLMs, contributes to a growing carbon footprint. When models are overfitted, not only is there wasted computational effort during training, but the resulting models may also be less efficient in deployment, requiring more computational power during inference. Cloud data centers, where much of this training and deployment occurs, significantly contribute to global carbon emissions. Companies like Google and Microsoft have recognized this and are investing in carbon-neutral data centers and more efficient algorithms. For instance, Google's switch to Tensor Processing Units (TPUs) and their focus on carbon-aware data centers are steps toward mitigating the carbon footprint. However, the challenge remains to balance the development of increasingly sophisticated models with the need to reduce their environmental impact, emphasizing the importance of efficient training practices and model optimization to limit the carbon emissions of GenAI technologies.

## 11.2 Measuring Through Modern AI Technologies

The impact of Gen AI in platform engineering can be accurately quantified using industry-standard AI tools and measurement techniques focusing on key performance indicators (KPIs) like the four key DORA metrics we discussed in Chapter 10. Advanced analytics platforms like *Datadog*, *New Relic*, and *Dynatrace* now incorporate AI and machine learning (ML) capabilities to process vast amounts of data from CI/CD pipelines, monitoring systems, and code repositories. These tools, often enhanced by LLMs, can analyze historical deployment data, source code changes, and operational metrics to identify inefficiencies and recommend targeted improvements.

Utilizing AI-driven observability tools to measure the tangible impact of GenAI on their platform engineering efforts is incredibly useful. These observability platforms, powered by LLMs, could process logs, metrics, traces, and user interactions to gain deep insights into system behavior. For example, by integrating tools like *Splunk* and *ELK (Elasticsearch, Logstash, and Kibana)* with their GenAI models, an organization could automatically parse logs and identify patterns correlating with deployment failures or performance bottlenecks. Through this data-driven approach, you can pinpoint where GenAI-driven automation had the most significant impact. The analysis revealed that the automated code generation feature resulted in a 25% increase in developer productivity by reducing manual coding efforts. Similarly, implementing smart code reviews using GenAI led to a 30% reduction in production defects by providing developers with immediate, context-aware feedback during development.

GenAI's capabilities also extend to assessing code quality and test coverage comprehensively. Industry-standard tools like *SonarQube*, *CodeClimate*, and *Coverity*, when augmented with GenAI, can automatically generate unit and integration tests based on the existing codebase and identify gaps in test coverage. These tools employ LLMs to conduct root cause analysis of test failures, allowing for rapid identification and resolution of defects. Moreover, GenAI can detect code complexity issues early in development, recommending refactoring where necessary to enhance code maintainability. Organizations can iteratively refine their platform engineering practices by continuously monitoring these metrics. For instance, GenAI can provide actionable insights on cyclomatic complexity and code duplication, helping teams maintain a clean and efficient codebase, leading to a more robust and reliable Software Development Lifecycle (SDLC).

PETech's approach exemplifies how integrating GenAI with industry-standard monitoring and analytics tools can transform platform engineering. They employed advanced AI-driven monitoring solutions like Prometheus and Grafana, enhanced with LLMs, to create dashboards that provided real-time visibility into their development and operational metrics. These tools tracked DevEx by analyzing developer workflows. They monitored CI/CD pipelines, automatically alerting teams to potential issues such as slow build times or high failure rates. PETech proactively addressed performance degradation using anomaly detection algorithms, thereby reducing MTTR and enhancing system reliability.

Furthermore, using AI-enhanced DevOps tools like *GitLab CI/CD* and even *Jenkins X* enabled PETech to improve continuously. These tools utilized GenAI to analyze deployment frequency, lead time for changes, and other DORA metrics to provide insights into the overall efficiency of the development process. For example, by examining historical deployment data, GenAI could predict the likelihood of a deployment failure based on recent code changes and suggest mitigation strategies, such as additional testing or phased rollouts.

GenAI's integration into platform engineering offers a sophisticated means of measuring and improving critical KPIs. By leveraging advanced AI-driven observability and analytics tools, organizations can gain actionable insights into the efficiency of their SDLC, optimize developer productivity, and enhance the overall quality and reliability of their software platforms. This continuous feedback loop, driven by data and enhanced by AI, enables teams to adapt and evolve their platform engineering practices, ensuring they remain agile and responsive to the ever-changing demands of software development.

### **11.2.1 Reporting on the Measurements**

Tools like *Faros.ai* and *LinearB* are at the forefront of integrating Gen AI into developer environments to enhance software engineering quality and boost organizational productivity. They serve as an engineering operations platform aggregating data across various stages of the software development lifecycle (SDLC). Incorporating GenAI-driven insights provides a unified view of engineering activities, enabling teams to make data-driven decisions and optimize their workflows.

For example, *Faros.ai* leverages data from diverse sources such as CI/CD pipelines, version control systems, issue trackers, and observability platforms to create a comprehensive picture of the development process. When augmented with GenAI, this data is aggregated and analyzed to detect patterns, identify bottlenecks, and recommend improvements. For example, the platform can use LLMs to analyze commit histories, code review comments, and deployment logs to pinpoint areas where developer friction occurs, such as prolonged code review cycles or frequent deployment failures. By providing actionable insights, *Faros.ai* enables teams to address these issues proactively, leading to a more streamlined development process and improved Developer Experience.

One of the key benefits of incorporating GenAI into the software engineering intelligence platform ecosystem is its ability to measure and report on key performance indicators (KPIs) more intelligently. Traditional metrics like deployment frequency, lead time for changes, and Mean Time to Recovery (MTTR) are enhanced with AI-generated context. For instance, these tools can analyze deployment frequency alongside code complexity metrics to understand whether high deployment rates correlate with increased complexity and potential technical debt. To balance speed and code quality, the platform can use GenAI to recommend specific interventions, such as refactoring efforts or targeted code reviews.

Some of these tools also integrate with GenAI and enable predictive analytics within the developer environment. The platform can forecast potential issues by analyzing historical data and ongoing trends before they escalate. For example, suppose the platform detects a pattern of declining test coverage or an increase in incident reports post-deployment. In that case, it can use GenAI to predict the likelihood of future production incidents. This predictive capability allows engineering teams to take preemptive actions, such as adding additional test cases or conducting more in-depth code reviews, thereby reducing the risk of outages and improving system reliability. In Figure 11.4, we are sharing a screenshot from Faros.AI tool that shows the metrics of copilot adoption.



**Figure 11.4** A screenshot from the Faros.AI tool showing the metrics with Copilot adoption in an organization.  
Reproduced with prior permission from the vendor

These tools improve productivity by automating reporting and feedback loops within the developer environment. GenAI models can synthesize vast amounts of data into concise, actionable reports highlighting critical areas of concern and success. For example, at PETech, while using a tool like this, instead of manually sifting through deployment logs and test results, developers and engineering managers receive AI-generated summaries that outline the most critical insights, such as modules with the highest failure rates or areas where automation has significantly improved deployment speed. These reports can include recommendations generated by GenAI, such as optimizing specific parts of the CI/CD pipeline or adjusting resource allocation in cloud environments to reduce build times. This saves time and ensures that teams focus on the most impactful improvements.

GenAI-powered insights help organizations foster a culture of continuous improvement. By providing data-backed feedback on engineering practices, teams can identify trends and patterns in their workflows that correlate with higher productivity and quality. For instance, the platform might reveal that certain code review practices lead to faster approvals and fewer post-merge defects. Using this information, teams can standardize best practices across the organization, leveraging GenAI-generated guidelines to refine their development processes. This ongoing measurement, analysis, and adjustment cycle creates an environment where teams can consistently enhance their performance and output quality.

These tools transform how organizations measure and improve their development processes by incorporating GenAI into their ecosystem. By providing deep, data-driven insights into every aspect of the SDLC, Faros.ai enables teams to identify inefficiencies, predict potential issues, and implement precise improvements. This leads to higher-quality software, more efficient workflows, and a more productive and agile organization.

### 11.3 Future of Improving DevEx Through GenAI

The future of Gen AI in platform engineering is set to revolutionize DevEx by automating routine tasks and providing a personalized, intelligent development environment. GenAI's ability to understand and adapt to the nuances of each project makes it an invaluable asset in transforming Integrated Development Environments (IDEs) into proactive assistants. These intelligent assistants can provide real-time coding support, suggest contextually relevant improvements, generate unit tests, and anticipate developer needs based on the project's context. By embedding GenAI into the development workflow, organizations can significantly reduce the cognitive load on developers, enhance productivity, and ensure code quality through adherence to best practices and standards.

PETech is currently in the early stages of exploring this space and has the opportunity to leverage GenAI in ways that can drastically improve its development processes. They should start by integrating GenAI-enhanced IDEs into their workflow to unlock this potential. These advanced IDEs, powered by LLMs fine-tuned on PETech's proprietary codebase and internal best practices, can offer sophisticated, context-aware code suggestions. For example, when a developer works on a feature, the GenAI-enabled IDE can provide inline code suggestions, highlight potential refactoring opportunities, and generate real-time documentation based on PETech's architectural guidelines. This accelerates the development process and ensures code quality and consistency by directly embedding architectural standards and coding guidelines into the development environment. By doing so, PETech can reduce development time by 20% and improve overall code quality, making the development experience more enjoyable and efficient.

Furthermore, GenAI can be crucial in resolving complex coding issues by generating solutions to intricate problems. Advanced LLMs can analyze codebases, learn from historical data such as previous code reviews and bug reports, and use this knowledge to identify common pitfalls. For instance, if PETech encounters recurring performance issues in a specific module, the GenAI system can analyze the underlying code patterns, correlate them with past incidents, and provide recommendations to prevent similar issues in the future. By fostering a proactive approach to software quality, PETech can mitigate risks and reduce the time spent on debugging and troubleshooting.

Beyond providing real-time coding assistance, GenAI can automate numerous mundane yet essential tasks such as code formatting, linting, dependency management, and environment configuration. The paper discussed earlier emphasizes the importance of automating these routine tasks to free up developers for more creative and value-driven work. PETech can automatically leverage GenAI to enforce code style guidelines, manage dependencies, and streamline environment setup, ensuring consistency across teams and projects. This automation enhances the developer experience by reducing repetitive work and minimizes errors and technical debt, leading to a more maintainable and robust codebase.

Additionally, PETech should consider employing GenAI to facilitate intelligent code reviews. The paper highlights the significance of incorporating AI into the code review process to enhance the quality and reliability of software. GenAI-powered code review tools can analyze changes, provide feedback on potential issues, and suggest improvements based on historical data and coding standards. By integrating these tools into their development workflow, PETech can accelerate the review process, ensure a higher standard of code quality, and create a culture of continuous learning and improvement among developers.

To maximize GenAI's benefits in their development environment, PETech should also focus on creating a feedback loop where the GenAI models continuously learn and adapt from developer interactions. By capturing insights from daily coding activities, pull request feedback, and production performance data, the GenAI system can refine its suggestions and recommendations over time. This adaptive learning approach, as discussed in the paper, ensures that the GenAI-enabled IDEs and tools evolve alongside the development team, providing increasingly valuable support tailored to PETech's unique context.

By integrating GenAI-enhanced IDEs, automating routine tasks, and implementing intelligent code reviews, PETech can significantly improve the Developer Experience. This strategy accelerates development, improves code quality, and fosters an environment where developers can focus on creativity and innovation, ultimately driving the organization toward higher productivity and better software outcomes.

### **11.3.1 The impact of copilot tools**

Copilot tools have emerged as a significant advancement in platform engineering and enhancing the DevEx. These tools, powered by GenAI and LLMs, act as intelligent assistants within development environments, automating routine tasks, providing real-time code suggestions, and improving overall productivity. They embody the principles of reducing developer friction, enhancing code quality, and accelerating software delivery cycles.

As most of the readers might be well aware of, some of the critical benefits of these tools are

1. Reducing Developer Friction: Copilot tools simplify complex coding tasks, provide intelligent suggestions, and automate routine processes, reducing friction in the development workflow. They free developers from repetitive tasks and allow them to focus on more creative and strategic aspects of software development.
2. Enhancing Code Quality and Security: By integrating best practices and security guidelines into code suggestions, copilot tools help maintain high code quality and security standards. They provide real-time feedback and highlight potential issues, ensuring that codebases are robust and compliant with organizational standards.
3. Accelerating Development Cycles: Copilot tools shorten development cycles by automating code generation, testing, and debugging. This aligns with the principles of platform engineering, where accelerating software delivery without compromising quality is a crucial objective.
4. Personalized and Context-Aware Assistance: These tools use GenAI to understand the developer's context and preferences, offering personalized recommendations that align with the project's specific needs. This personalization enhances the DevEx by making the development process more intuitive and efficient.

**Table 11.1** The following table summarizes the data available on the adoption of GitHub copilot and AI coding Tools as of July 2024

| Consideration                  | Data/Statistics                                                          |
|--------------------------------|--------------------------------------------------------------------------|
| Organizations                  | More than 50,000 organizations                                           |
| Developer Sentiment            | 90% of developers are happier                                            |
| Corporate Adoption             | 67% of the developers used it at least five days/week                    |
| Ease of Adoption               | 81% installed IDE extension within a week of licensing                   |
| Impact on GDP                  | Increases Global GDP by \$1.5Trillion by 2030                            |
| Productivity Gains             | This is equivalent to adding 15 million developers to a global workforce |
| Acceptance rate of suggestions | 30%                                                                      |

Copilot tools like *Github*, *Amazon CodeWhisperer*, *Tabnine*, and *Replit Ghostwriter* embody GenAI's transformative potential in platform engineering. By automating coding tasks, enhancing code quality, and personalizing the development environment, they align with the core principles of improving DevEx and reducing developer friction. As intelligent assistants, these copilot tools accelerate the software development lifecycle and empower developers to focus on innovation, creativity, and building high-quality software.

### 11.3.2 Conversational interface driven operational improvements

A few tools in the industry, most notably *Kubiya.ai*, focus on transforming how developers and operations teams interact with their infrastructure by introducing an AI-driven conversational interface. GenAI techniques allow users to manage cloud infrastructure, Kubernetes environments, and DevOps workflows through natural language queries and commands. Another simple open source tool that is effective for doing Infrastructure as Code is AlaC, a command-line interface, which will enable developers to generate the appropriate IaC code and scripts. This conversational AI model is trained to understand domain-specific language related to cloud operations, making it easier for teams to perform complex tasks without remembering intricate command-line instructions or the specifics of various cloud provider APIs.

Here is a summary of the value provided by such tools.

1. They enable developers and DevOps teams to interact with their infrastructure using simple, natural commands. For instance, a developer can ask Kubiya.ai to "scale the Kubernetes deployment for the payment service" or "check the status of the CI/CD pipeline." The tool uses GenAI to understand the intent behind these commands, translate them into actionable tasks, and execute them on the underlying infrastructure. This conversational approach reduces the complexity and learning curve of managing cloud infrastructure, improving the developer experience and accelerating workflows.
2. These interfaces go beyond generating infrastructure code by letting you diagnose and resolve issues in kubernetes and below. A recent entry (as of 2024), in CNCF is a tool called K8sGPT, which uses AI to analyze and diagnose issues with K8S.
3. By leveraging GenAI, these tools can maintain context over a series of interactions. If a user asks, "How many pods are currently running?" followed by "Scale them up by 2," the AI understands the context of "them" refers to the pods queried in the previous request. This context-awareness simplifies multi-step operations and makes it easier to manage infrastructure conversationally. This capability enhances productivity by streamlining operations and reducing the need for constant back-and-forth or manual intervention.
4. They can integrate with observability tools, discussed in detail in the next section, and monitoring systems to provide proactive incident management. When an anomaly is detected, such as a resource utilization spike or a service outage, they can automatically trigger predefined remediation actions or guide users through troubleshooting using conversational prompts. This reduces Mean Time to Recovery (MTTR) by enabling rapid, AI-driven responses to operational issues.
5. They can serve as a knowledge repository that learns from interactions over time. By analyzing user queries and actions, they build an understanding of typical workflows and best practices, which can be shared across teams. This is particularly valuable for onboarding new team members, as they can quickly learn and execute tasks by interacting with these tools, leveraging the organization's accumulated knowledge.

### **11.3.3 Automated Resource Optimizations**

A different class of tools within the FinOps (Financial Operations) space focuses on optimizing cloud infrastructure, particularly for Kubernetes environments. Using GenAI techniques, tools such as CAST AI automate cloud cost management, resource optimization, and scaling decisions. They continuously analyze workloads, usage patterns, and cloud pricing to provide actionable insights and automated optimization strategies.

Here is a summary of the value provided by such tools.

1. These tools use GenAI to analyze cloud resource usage and identify cost-saving opportunities. By examining workload patterns, resource allocation, and cloud pricing models, they can make real-time recommendations, such as rightsizing instances, leveraging spot instances, or scheduling workloads to run during off-peak hours. Their GenAI models dynamically adjust these recommendations based on current usage and demand, enabling organizations to reduce cloud expenses without sacrificing performance or reliability.
2. They can employ GenAI to optimize the autoscaling of Kubernetes clusters. Traditional autoscaling mechanisms rely on predefined rules or metrics thresholds, which may not always meet application needs. GenAI models incorporated in these tools analyze historical usage patterns, real-time metrics, and application requirements to make intelligent autoscaling decisions. For example, it can predict upcoming traffic spikes based on past behavior and proactively scale the cluster to ensure seamless performance. This results in a more efficient use of resources and prevents over-provisioning or under-provisioning of infrastructure.
3. Tools such as CAST AI use GenAI to continuously monitor the cloud environment and identify and resolve potential infrastructure issues before they impact performance. For instance, if the system detects a potential bottleneck in the network or a node reaching resource capacity, it can automatically take corrective actions, such as reallocating workloads, adjusting network configurations, or provisioning additional resources. This proactive management approach ensures high availability and optimal performance of applications running on Kubernetes.
4. These tools can also abstract the complexity of managing multi-cloud and hybrid-cloud environments. GenAI automates cloud operations such as workload placement, instance selection, and network configuration, simplifying cloud management for engineering teams. It provides a unified platform where complex cloud decisions are made automatically, reducing the need for manual intervention and allowing teams to focus on delivering value through their applications.

## 11.4 Generative AI Enabling Observability

Observability has become an essential practice for maintaining and optimizing modern distributed systems, providing deep insights into the internal state and performance of complex applications. With the proliferation of microservices, containers, and cloud-native architectures, the volume and complexity of observability data have increased exponentially. Metrics, logs, and traces are the three pillars of observability, each offering valuable information, but manually sifting through this data to detect anomalies and diagnose issues is both time-consuming and error-prone. GenAI is transforming this landscape by enhancing observability platforms with advanced data processing, pattern recognition, and autonomous analysis capabilities.

In Chapter 5, while discussing evolutionary observability, we first introduced where GenAI impacts the observability space. Before delving deeper into it, we will discuss how this is done and how the technology is evolving.

GenAI enhances observability by leveraging LLMs and other machine learning techniques to process and analyze vast amounts of data in real time autonomously. Observability vendors like *Datadog*, *New Relic*, and *Splunk* have begun integrating GenAI into their platforms to deliver more intelligent insights. These tools, combined with open standards like OpenTelemetry, enable organizations to collect, process, and export observability data in a standardized format, making it easier for GenAI models to consume and analyze. OpenTelemetry, the emerging standard for observability, provides a vendor-neutral framework that collects telemetry data from various sources. When integrated with GenAI, it enriches this data with context, allowing the AI to detect anomalies, correlate them with potential root causes, and predict future issues before they impact the system.

PETech's implementation of a GenAI-based observability solution demonstrates the power of these advancements. By leveraging LLMs and machine learning models, PETech's observability system autonomously analyzed logs, metrics, and traces in real-time, identifying patterns and anomalies that could indicate performance issues, security threats, or system failures. The system utilized unsupervised learning techniques to detect unusual patterns in telemetry data, while supervised learning models were trained on historical incident data and classified and diagnosed anomalies. For example, when PETech experienced latency spikes, the GenAI system correlated this anomaly with resource contention in specific microservices, suggesting targeted remediation actions like scaling up instances or optimizing database queries. This approach reduced Mean Time to Recovery (MTTR) by 50%, as the system could proactively recommend and even automate resolutions before they become critical.

Beyond real-time anomaly detection and root cause analysis, GenAI significantly enhances the predictive capabilities of observability platforms. Traditional observability tools primarily focus on reactive monitoring, alerting teams after an incident has occurred. GenAI, however, shifts this paradigm toward proactive and predictive observability. By continuously analyzing historical and real-time telemetry data, GenAI can identify leading indicators of potential failures, such as gradual increases in memory usage, intermittent latency issues, or unusual access patterns. This predictive analysis enables organizations to address potential problems before they escalate into full-blown incidents. For instance, PETech's GenAI observability solution could predict system degradation by detecting subtle changes in trace patterns and automatically triggering remediation workflows, such as preemptively scaling infrastructure or adjusting resource allocations.

Security observability is another area where GenAI proves invaluable. Observability platforms now incorporate GenAI models to analyze security logs, identify unusual access patterns, and detect anomalies indicative of potential threats, such as data breaches or insider attacks. GenAI-enhanced observability tools continuously learn from evolving security incidents and adapt their detection algorithms to emerging attack vectors, providing a dynamic and changing defense mechanism. By incorporating OpenTelemetry's rich set of signals and context into security analysis, GenAI can cross-correlate network, application, and infrastructure telemetry data to identify complex attack patterns that might go unnoticed. For example, PETech's observability stack used GenAI to analyze authentication logs, API request patterns, and network traffic, detecting anomalous behavior that indicated a possible insider threat. The system automatically alerted the security team and initiated automated remediation steps, such as revoking compromised credentials and isolating affected systems.

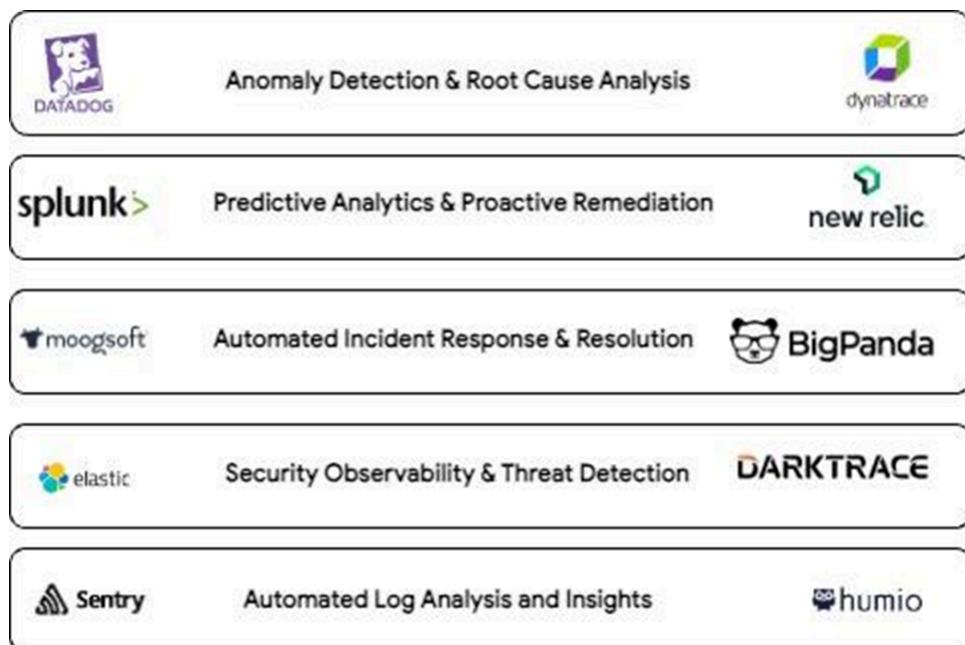
Moreover, GenAI-enhanced observability contributes to continuous improvement and system reliability by providing actionable insights based on its analysis. It goes beyond anomaly detection by suggesting optimal configurations, scaling actions, and architectural enhancements. For example, if the GenAI system detects frequent latency issues in a particular service, it can recommend more efficient caching strategies, database indexing changes, or traffic routing alterations. These insights are directly fed into the incident management system and the broader observability pipeline, facilitating automated resolutions and reducing the burden on DevOps teams.

By integrating GenAI into their observability stack, organizations like PETech achieve higher system reliability and operational efficiency. GenAI's ability to process vast amounts of data, learn from historical incidents, and adapt to evolving system dynamics empowers engineering teams with the foresight and agility needed to maintain optimal system performance. Furthermore, combining OpenTelemetry with GenAI ensures that observability data is collected and analyzed in a standardized, interoperable manner, enabling organizations to implement vendor-agnostic, scalable observability solutions that evolve with their systems.

GenAI is revolutionizing observability by turning it from a reactive practice into a proactive and predictive one. By leveraging LLMs and advanced AI models, organizations can autonomously analyze observability data, detect real-time anomalies, predict potential failures, and enhance security observability. This reduces MTTR and enables automated remediation, ensuring systems remain resilient and performant in increasingly complex and dynamic environments.

Many vendors and tools do well within this space. A special call out to Chronosphere, which has a unique product that applies Generative AI in the observability domain. This product is crucial in enabling observability for Generative AI (GenAI) systems by providing advanced monitoring and analysis tailored to the complexities of AI-driven environments. GenAI models, especially in large-scale applications, generate vast metrics, logs, and traces that must be monitored in real time for optimal performance. Chronosphere's AI-powered observability platform can efficiently handle this high cardinality data, using intelligent anomaly detection to identify deviations in system behavior that could impact the performance and accuracy of GenAI models. It correlates data across diverse sources, offering advanced root cause analysis to pinpoint issues within the underlying infrastructure or the AI quickly models themselves. By providing predictive analytics, Chronosphere helps anticipate potential resource constraints, model performance degradations, and other anomalies before they affect the system. This proactive approach ensures that GenAI systems remain resilient, performant, and reliable, enabling organizations to leverage AI insights confidently while maintaining robust operational health. More and more of the newer products will take unique approaches within the observability space.

We analyzed the popular observability tooling vendors in the industry. We devised the following classification across the five axes on where each vendor fared the best. We acknowledge this is a fast-evolving area with significant research and product enhancements in each tool. However, for the benefit of the readers, we share the following illustration, and we will talk about each of these axes in the subsequent sections. Figure 11.5 shows the classification of various tools across multiple axes of observability.



**Figure 11.5** A classification of some of the most popular observability tools in the industry today and how they fare against each of the five axes of comparisons we identified.

### **11.4.1 Anomaly Detection and Root Cause Analysis**

Traditional observability tools often need help with the sheer volume of logs, metrics, and traces generated by complex distributed systems. GenAI, however, can autonomously sift through this data to detect anomalies and identify root causes in real-time.

Dynatrace uses GenAI to provide automatic anomaly detection and root cause analysis. Its Davis AI engine analyzes metrics, logs, and traces to identify patterns indicative of normal and abnormal system behavior. When an anomaly is detected, Davis not only alerts the team but also provides a detailed root cause analysis, pinpointing the exact source of the problem, such as a failing microservice or a database bottleneck. By understanding complex dependencies across the application stack, Dynatrace can reduce Mean Time to Recovery (MTTR) and improve overall system reliability.

Datadog also employs AI-driven anomaly detection with its Watchdog feature, which automatically uses machine learning models to identify performance anomalies in applications and infrastructure. Watchdog surfaces issues such as latency spikes, error rate increases, or unusual system behavior, providing actionable insights without needing pre-configured thresholds. Learning from system behavior minimizes noise and ensures that alerts are relevant and actionable.

### **11.4.2 Predictive Analytics and Proactive Remediation**

GenAI can extend observability from reactive to proactive by predicting potential failures and automatically taking corrective actions. This predictive capability enables organizations to address issues before they impact users.

Splunk utilizes GenAI for observability predictive analytics. By analyzing historical data and system telemetry, Splunk's AI-powered solutions can predict future system states, such as impending resource exhaustion or potential service degradation. Splunk's Machine Learning Toolkit allows teams to build and operationalize custom machine learning models for specific use cases, such as predicting CPU spikes or network latency. When anomalies are predicted, Splunk can proactively trigger automated workflows to remediate issues, such as scaling up resources or restarting problematic services.

New Relic employs AI/ML capabilities to provide predictive insights. Its AI-powered observability platform monitors key performance indicators (KPIs) and analyzes historical trends to forecast potential system failures or capacity issues. New Relic's Applied Intelligence suite uses anomaly detection and predictive analytics to alert teams to potential problems and suggest proactive steps to mitigate them, enhancing system resilience and reducing downtime.

### **11.4.3 Automated Incident Response and Resolution**

GenAI-enhanced observability tools can automate the incident response process, reducing the need for manual intervention. When detecting an anomaly or incident, GenAI can suggest or automatically execute remediation actions based on learned behavior and predefined playbooks.

Moogsoft uses AI and machine learning to provide automated incident response and event correlation. Its AIOps platform aggregates data from various observability sources, using machine learning models to detect anomalies and correlate events across the application stack. Moogsoft's virtual assistants can automatically trigger incident response workflows, execute remediation actions, or escalate issues to the appropriate teams. For example, if a sudden surge in CPU usage is detected, Moogsoft can initiate an automated scaling action or perform a diagnostic script to restore regular operation.

BigPanda leverages AI to automate incident response by correlating alerts from different observability tools with high-level incidents. Its Event Correlation and Automation platform uses GenAI to analyze alert patterns, detect root causes, and execute automated responses. BigPanda's AI engine can suggest remediation actions based on historical incident resolution data, enabling teams to respond faster and more effectively to system issues.

#### **11.4.4 Security Observability and Threat Detection**

GenAI can enhance security observability by identifying unusual patterns in security logs and network traffic detecting potential threats like intrusions or insider attacks. This provides a more dynamic and evolving defense mechanism against security threats.

Elastic Observability incorporates machine learning and AI to detect security threats within observability data. Elastic's Security Analytics uses GenAI to analyze logs, network traffic, and system behaviors to identify anomalies that could indicate security breaches. It can automatically detect deviations from standard patterns, such as abnormal user behavior or unusual access patterns, and trigger alerts or automated responses to mitigate the threat.

Darktrace applies GenAI in its Cyber AI platform to enhance security observability. Darktrace's AI models continuously learn from the behavior of users and devices within the network, using this understanding to identify and respond to novel cyber threats in real-time. When an anomaly is detected, such as an unusual data transfer or a deviation in network traffic patterns, Darktrace can autonomously respond to contain the threat, ensuring that security teams are alerted to genuine risks rather than false positives.

#### **11.4.5 Automated Log Analysis and Insights**

Logs are a rich source of information for understanding system behavior, but the volume and unstructured nature of log data makes manual analysis challenging. GenAI tools can automatically parse, analyze, and extract insights from logs, providing a more comprehensive view of system health.

Humio uses AI and machine learning to provide real-time log analysis and insights. It enables the ingestion and analysis of large volumes of log data, automatically detecting patterns, anomalies, and errors. By leveraging machine learning, Humio can identify unusual behaviors in log streams and correlate them with potential system issues or security threats, providing actionable insights for proactive remediation.

Sentry incorporates AI/ML techniques to enhance error tracking and log analysis in its application monitoring platform. It uses machine learning to group similar errors and provide root cause analysis, helping developers quickly identify and resolve issues. For example, it can analyze stack traces and application logs to pinpoint the source of an error, reducing the time needed for debugging and enhancing application reliability.

#### **11.4.6 Exercise 11.1: Automated Observability Configuration using Gen AI**

Objective: Use a Gen AI model to generate monitoring and observability configurations for a cloud-based microservices architecture. You should create the model as a function

Recommended Steps:

1. Create a simple input system where the developers can provide a prompt related to their observability requirements
2. Implement a function that simulates a Gen AI model to generate appropriate configurations. You can use your monitoring/observability tool, Prometheus/Grafana, or any vendors mentioned in the previous section.
3. The solution should encompass metrics collection, alerting, and dashboard definitions
4. Output the configuration your platform engineering team can use to set up observability for the cloud infrastructure of choice.

### **11.5 Newer Frontiers of Platform Engineering Within IDP and DevEx Evolution**

Internal Developer Platforms (IDPs), as discussed in Chapter 10, are undergoing a transformation driven by Gen AI, shifting from static, predefined environments to dynamic, intelligent systems that adapt to the evolving needs of developers. Modern IDPs augmented with GenAI offer a personalized and unified development environment, enabling teams to work more efficiently and with greater autonomy. These platforms leverage AI's capabilities to understand context, automate workflows, and make intelligent decisions that align with the specific requirements of developers. This evolution represents a significant leap forward in enhancing the DevEx, making it more intuitive, responsive, and adaptive.

One of the critical advancements in GenAI-powered IDPs is their ability to provide personalized recommendations tailored to individual developer workflows. The IDP can dynamically adjust its interface and services by analyzing developer behavior, code patterns, and project requirements. For example, PETech's AI-driven IDP uses GenAI to monitor coding habits, project dependencies, and historical data, recommend optimal build environments and code refactoring options, and suggest libraries or frameworks that align with the project goals. This personalized approach streamlines the development process, reducing the cognitive load on developers and allowing them to focus on higher-order problem-solving. As a result, PETech has experienced a marked improvement in developer satisfaction and productivity, with a 25% increase in satisfaction scores and a 15% faster feature delivery rate.

Beyond personalization, GenAI in IDPs also automates repetitive and time-consuming tasks, such as configuring environments, managing dependencies, and provisioning resources. Traditional IDPs often require manual intervention to set up development environments, leading to inconsistencies and potential errors. GenAI overcomes this by automating environment provisioning based on contextual awareness. For instance, when a developer starts working on a new project, the AI-driven IDP can automatically create the necessary development environment, set up the required dependencies, and configure resources based on the project's technology stack. This automation reduces developers' time on setup and ensures a consistent and error-free environment, enhancing DevEx and minimizing setup-related delays.

Integrating GenAI into IDPs also facilitates more effective code analysis and security enforcement. Modern GenAI-powered IDPs include intelligent code analysis tools that can detect security vulnerabilities, enforce coding standards, and provide real-time feedback on code quality. These tools leverage LLMs to understand the intricacies of codebases, identify potential issues, and suggest improvements. For example, they can spot security flaws such as SQL injection vulnerabilities or recommend optimizations to enhance code performance. Integrating these AI-driven code analysis capabilities into the development workflow, IDPs help maintain high-quality code and secure applications, reducing the likelihood of defects reaching production.

Moreover, GenAI-powered IDPs enhance deployment automation and infrastructure management. They can predictively scale infrastructure resources by analyzing historical usage patterns and current workload requirements, ensuring optimal performance and cost efficiency. For instance, if an application experiences a sudden surge in traffic, the AI-driven IDP can automatically scale resources to meet demand, avoiding downtime or performance degradation. Additionally, these platforms can intelligently automate deployment pipelines, optimizing the release process by analyzing code changes, conducting automated testing, and deploying applications most efficiently. Real-time feedback on code changes and deployment status further enhances DevEx, providing developers with instant insights into their code's impact on the system.

Finally, the evolution of GenAI in IDPs opens new frontiers for collaborative and cross-functional workflows. AI-driven IDPs can bridge development, operations, and security teams, providing a shared platform where insights and recommendations are accessible to all stakeholders. For example, when a developer pushes a new feature, the GenAI-enabled IDP can notify the operations team about the potential impact on infrastructure, suggest scaling actions, and even automate the necessary configurations. This collaborative approach ensures that all teams are aligned and the platform evolves with the organization's needs. By continuously learning from developer interactions and system performance, GenAI-enabled IDPs create a feedback loop that drives continuous improvement, ultimately leading to a more streamlined and enjoyable developer experience.

### **11.5.1 Product Engineering improvements through automated code learning and remediation**

When dealing with complex Level 3 (L3) incidents, especially in product domains where Site Reliability Engineers (SREs) may lack deep familiarity, GenAI-powered tools can provide valuable support. These tools can analyze codebases, system behavior, and historical incident data to suggest fixes, automate resolutions, or guide SREs through the resolution process.

For example, Sourcegraph Cody uses GenAI to understand and navigate large codebases, offering intelligent insights that help SREs and developers diagnose and fix issues quickly. When an L3 incident occurs in a part of the product that SREs are unfamiliar with, Cody can analyze the code, identify potential areas where the issue might reside, and provide suggestions for fixing the problem. For example, Cody can automatically search for related topics, review recent changes, and cross-reference documentation to pinpoint the root cause of a performance degradation or functional error. This accelerates the incident resolution process, even in unfamiliar codebases, by guiding SREs to the most relevant sections of the code and proposing actionable solutions.

This can now be easily combined with open-source tools such as reWrite, which employs GenAI to autonomously rewrite and refactor code to fix complex issues, such as security vulnerabilities, performance bottlenecks, and deprecated APIs. In scenarios where SREs encounter L3 incidents in unfamiliar product domains, this can automatically generate code patches to address the identified problems. This ecosystem can suggest changes that resolve incidents without requiring deep manual intervention by analyzing the codebase, understanding its structure, and referencing similar patterns. This is particularly valuable for quickly addressing critical issues in production environments while preserving code quality and functionality.

GitHub Copilot can assist in resolving L3 incidents by providing real-time coding assistance and suggestions directly within the development environment. When an incident requires a code change, SREs can use Copilot to generate code snippets, refactor existing code, or implement fixes based on the context provided in the codebase. For instance, if an SRE identifies an error related to a specific function, Copilot can suggest alternative implementations or bug fixes by analyzing the surrounding code and inferring the intended behavior. This helps SREs navigate unfamiliar domains more effectively and reduces the time to remediate incidents.

GenAI-powered tools like *Sourcegraph Cody*, *reWrite*, and *GitHub Copilot* extend observability capabilities by offering direct assistance in resolving complex Level 3 (L3) incidents, especially in product domains with which SREs may not be familiar. These tools leverage GenAI to analyze codebases, provide intelligent suggestions, and even automate code changes to address critical issues quickly. When integrated into the incident management workflow, they help SREs navigate unfamiliar product areas, identify root causes, and implement fixes more efficiently. By combining these tools with observability platforms such as *Dynatrace*, *Datadog*, and *Splunk*, organizations can establish a comprehensive strategy that detects and predicts issues and offers actionable remediation, reducing downtime and improving system reliability.

## 11.6 Summary

- GenAI enables context-aware solutions tailored to an organization's needs, streamlining complex decisions like cloud infrastructure optimization.
- LLMs automate processes across the Software Development Lifecycle (SDLC), including requirements generation, architectural design, code development, and legacy codebase refactoring.
- RAG improves LLM-generated solutions' contextual relevance and accuracy by retrieving domain-specific knowledge.
- RAG accesses ontologies, developer documentation, and compliance guidelines to ensure solutions adhere to internal and external standards.

- Key benefits of using copilot tools include reducing developer friction, enhancing code quality and security, accelerating development cycles, and offering context-aware assistance.
- Modern Software Engineering Intelligence platforms use LLMs to measure KPIs intelligently, provide predictive analytics, and automate reporting and feedback loops within the developer environment.
- GenAI-powered IDEs enhance DevEx by providing real-time coding support, automating mundane tasks, and facilitating intelligent code reviews.
- Newer tools in the market also provide context-aware operations, maintain workflow context, and serve as knowledge repositories for improving DevOps workflows.
- Some tools identify opportunities, optimize Kubernetes environments, and proactively manage infrastructure to ensure high availability and performance through Gen AI.
- GenAI enhances observability by autonomously analyzing metrics, logs, and traces to detect anomalies, predict failures, and provide root cause analysis.
- GenAI-powered tools can resolve complex L3 incidents in unfamiliar product domains, offering intelligent code navigation, automated refactoring, and real-time coding assistance.