Effective Platform Engineering

Ajay Chankramath
Nic Cheneweth
Bryan Oliver
Sean Alvarez

MEAP

MANNING

MEAP Edition
Manning Early Access Program

Effective Platform Engineering

Version 1

For more information on this and other Manning titles go to 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

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.

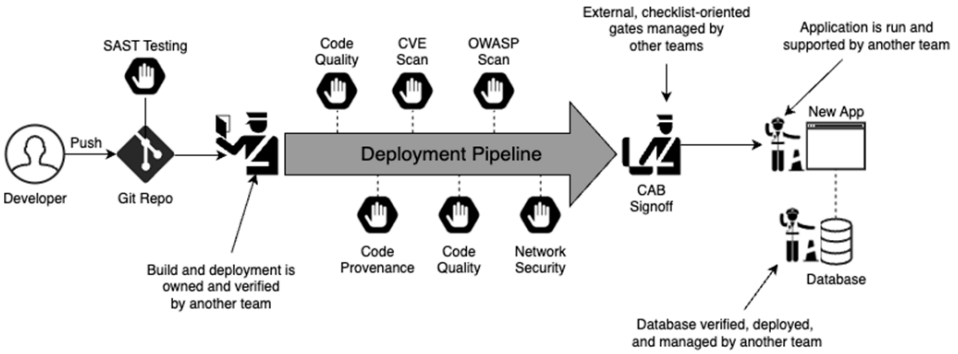

Figure 1.1 The PETech deployment process is very manual and requires multiple handoffs to different teams. There are also many different verification steps that result in a release process that can take weeks to complete once a code change is pushed. The developers have to understand and account for 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.

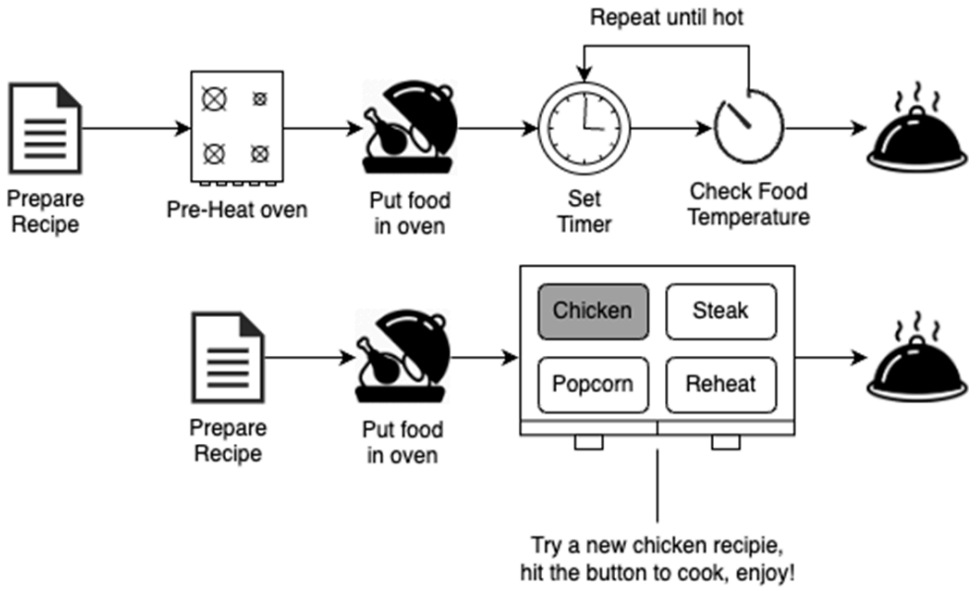

Figure 1.2 An oven with pre-programmed food choices would make experimenting with new recipes much easier. You wouldn't have to worry about knowing the right temperature or how long to cook a new meal. You could simply put the ingredients together, hit a button and enjoy! Building an engineering platform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

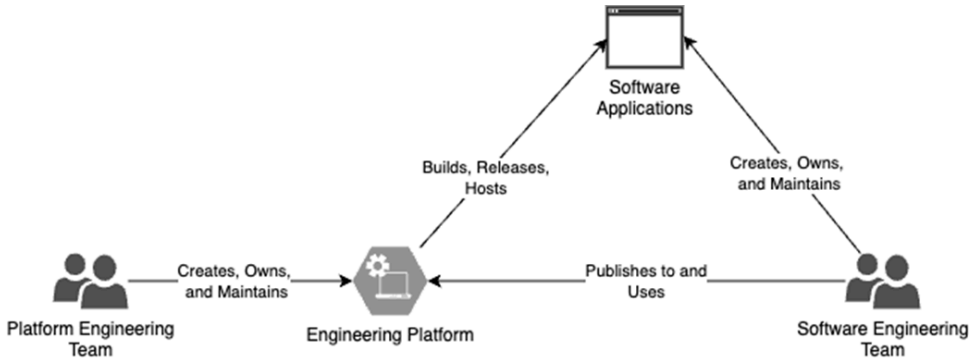

Figure 1.3 The relationship between a Platform Engineering team and the Engineering teams it serves is through an Engineering Platform. The PE Team provides a platform that defines the tools and processes needed to be secure and compliant, and teams use it to publish and operate their software.

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.

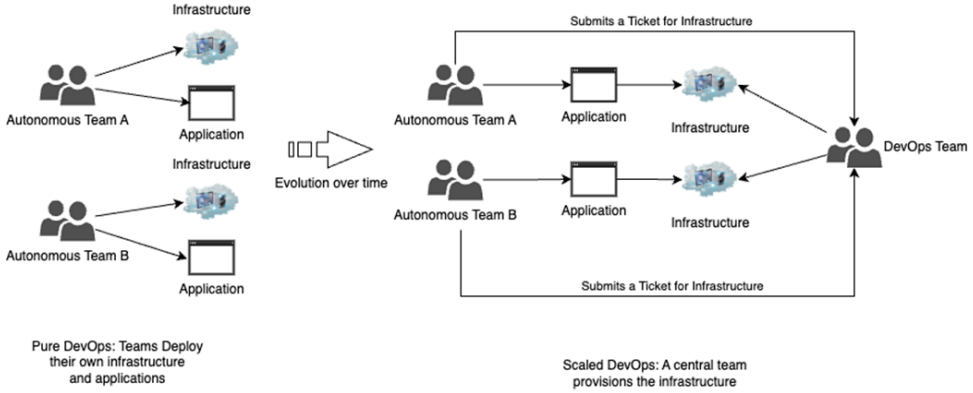

Figure 1.4 In a pure DevOps culture teams will deploy their own infrastructure, but this leads to sprawl in the IT estate. To combat this, many organizations will create central DevOps teams in an effort to enforce standardization. This leads to bottlenecks at scale, when the team becomes overwhelmed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

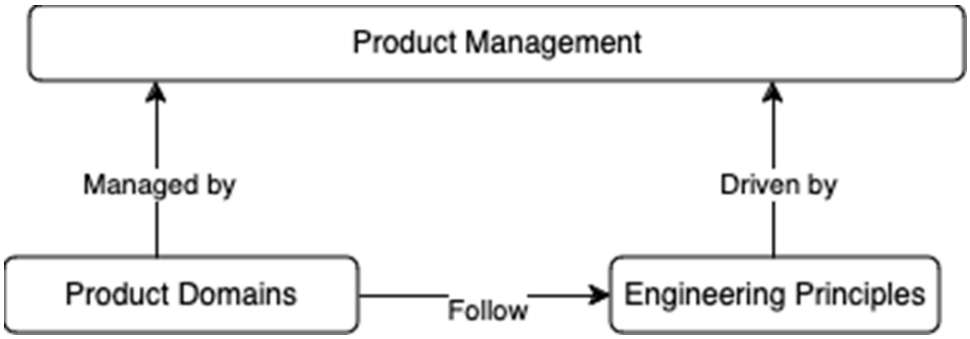

Figure 1.5 below shows the high-level mental model of platform engineering. It has three essential components: product domains, underlying engineering principles, and product management.

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 and 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.

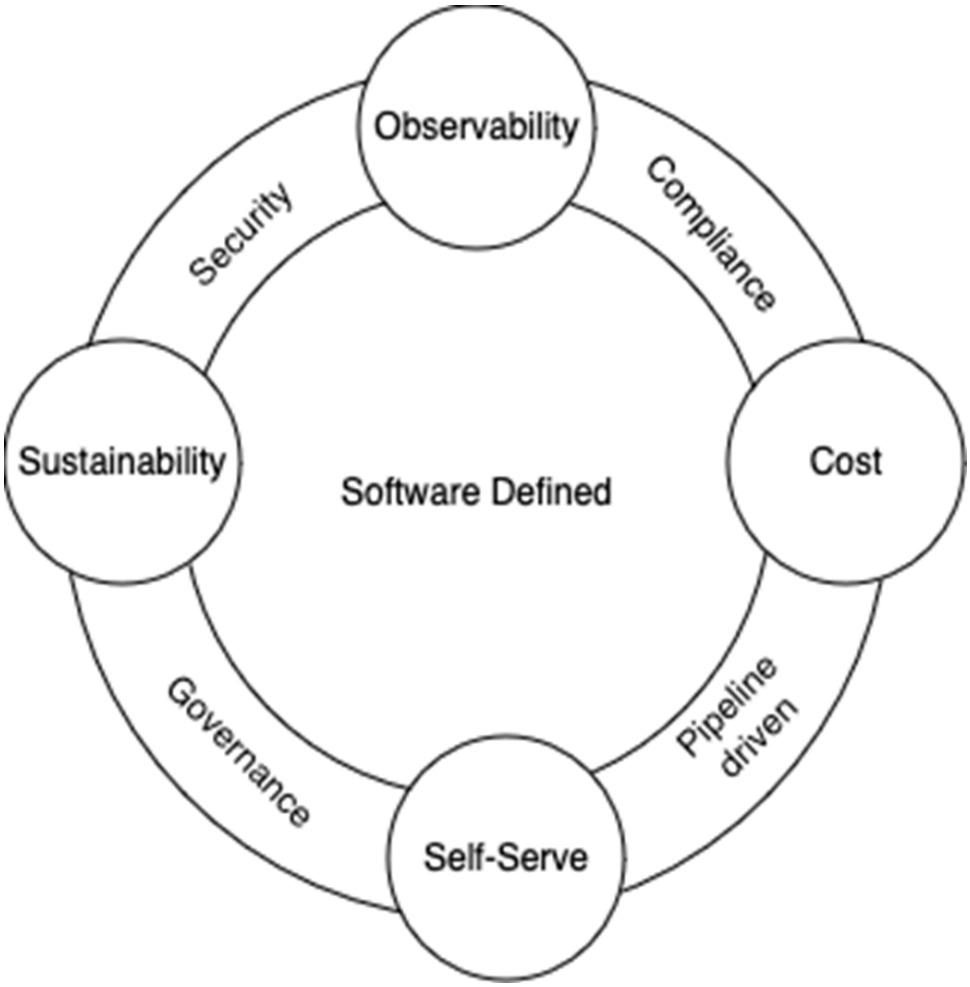

Figure 1.6 Expanded View of the Principles involved in the Platform Engineering Mental Model. As described in the previous figure, this figure shows the specific domains and principles that constitute the platform engineering mental model.

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.

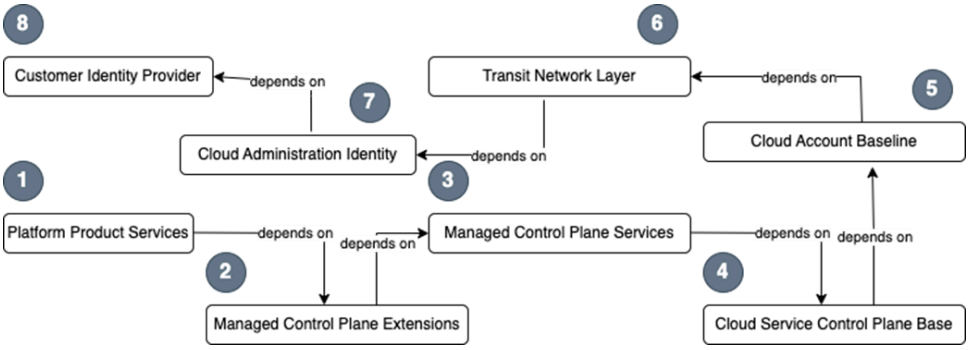

Figure 1.7 Expanded View of the Domains involved in the Platform Engineering Mental Model.

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 servie on AWS. When you stream a move or TV show on Netflix, the file that contains the movie exists on a physical device where it is assigned access permissioned based on the AWS IAM roles and permissions. However, as a user of Netflix you are not assigned an AWS IAM identity nor it 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.

· metrics-server · kube-state-metrics · datadog-agent · event-exporter · opentelemetry-operator	· cluster-autoscaler · karpenter · cloud vendor csi services · kubecost cost-model agent · gatekeeper

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.

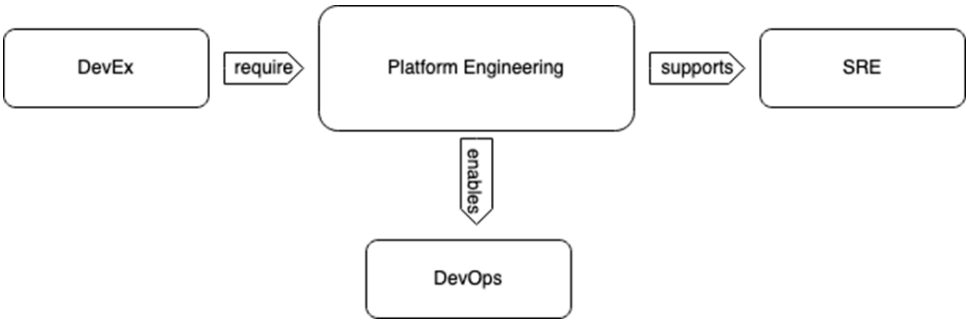

Figure 1.8 Platform Engineering Enablers. This view aims to ensure clarity regarding whether platform engineering is a replacement for DevOps or SRE. In our definition, this accessible reference tells us how it all fits together.

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.

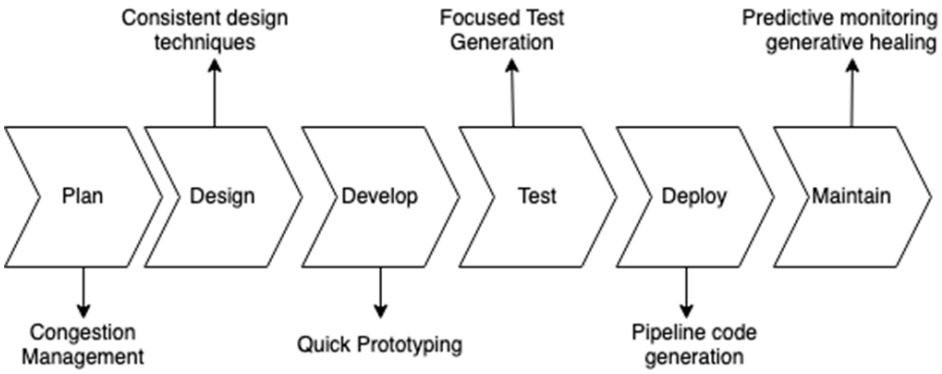

Figure 1.9 Focus Areas for Generative AI in Platform Strategy. This is a rapidly evolving space, and focus areas should be identified using the same product strategies discussed throughout this chapter.

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.

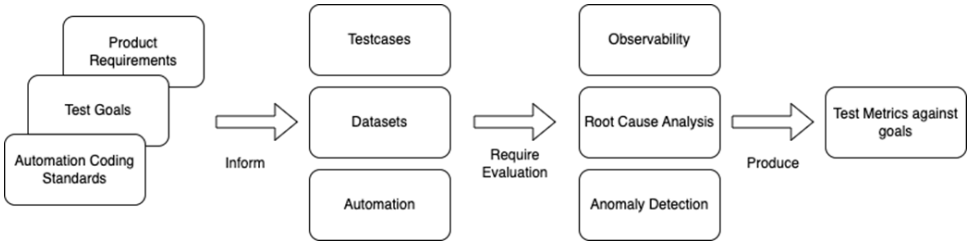

Figure 1.10 Test Evolution from a generative AI point of view. GenAI techniques can help perform efficient root cause analysis by quickly evaluating observability data.

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 productviity
- 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.

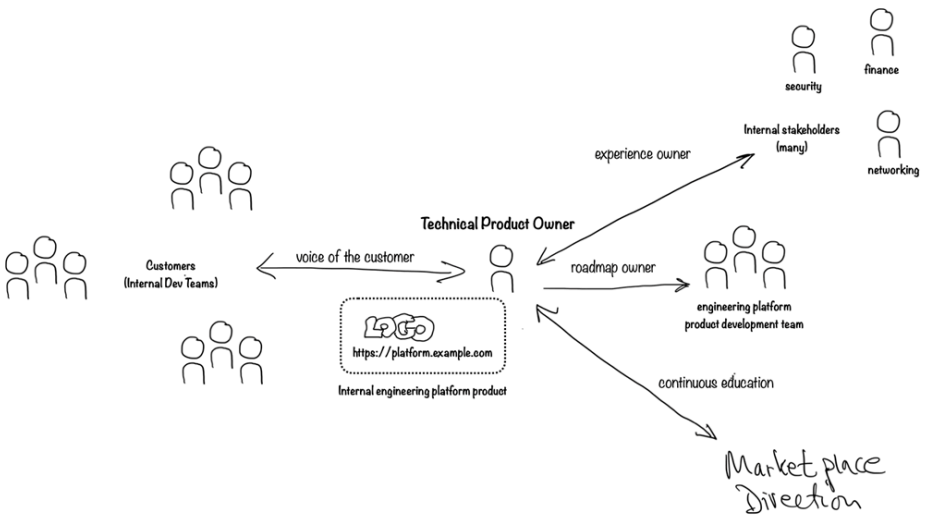

Figure 2.1 A Technical Product Owner (TPO) serves as the voice of the customer while understanding the technical needs of internal engineering teams. This role will manage the product roadmap to ensure that features are developed that will meet customer needs and provide the intended value.

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

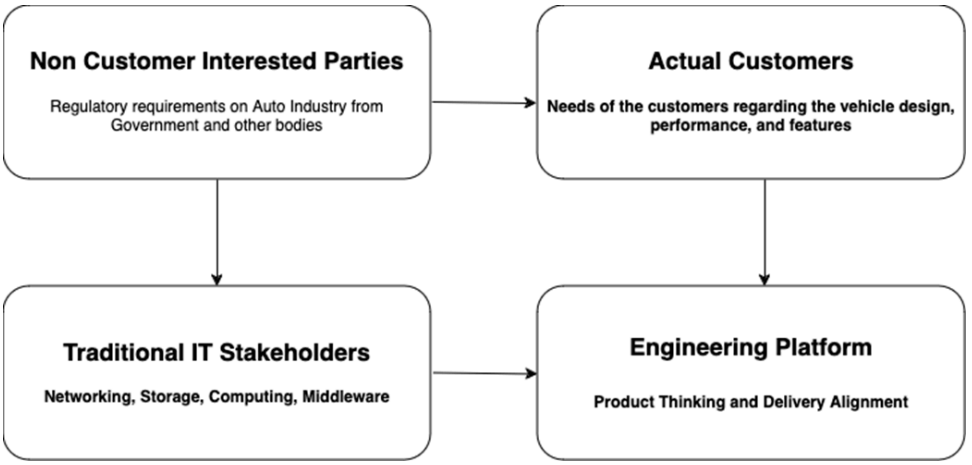

Figure 2.2 A four way relationship between the key components that shows how the customers and stakeholders drive the engineering platform requirements

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 his 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.

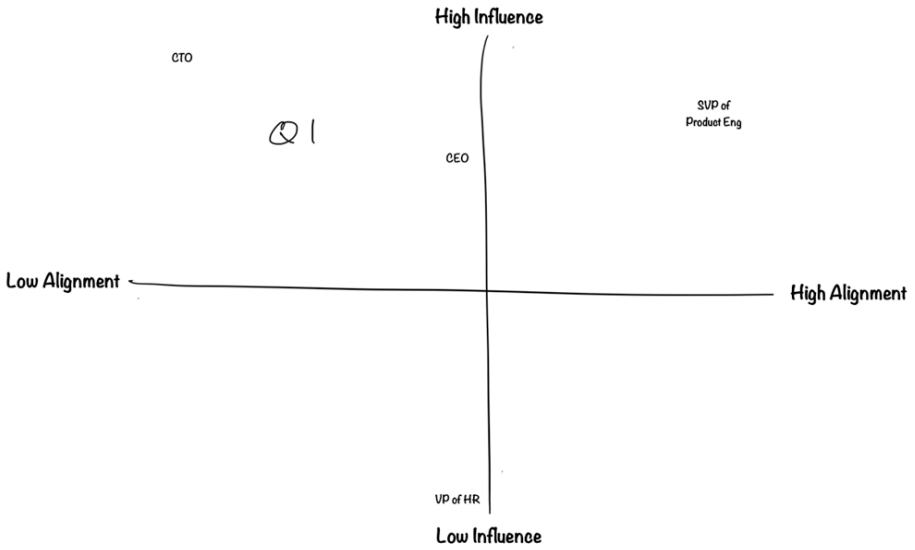

Figure 2.3 A Stakeholder Map to evaluate a stakeholder's influence over the backlog against their alignment to platform objectives. Ideally, high-influence stakeholders should be highly aligned to ensure platform success.

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

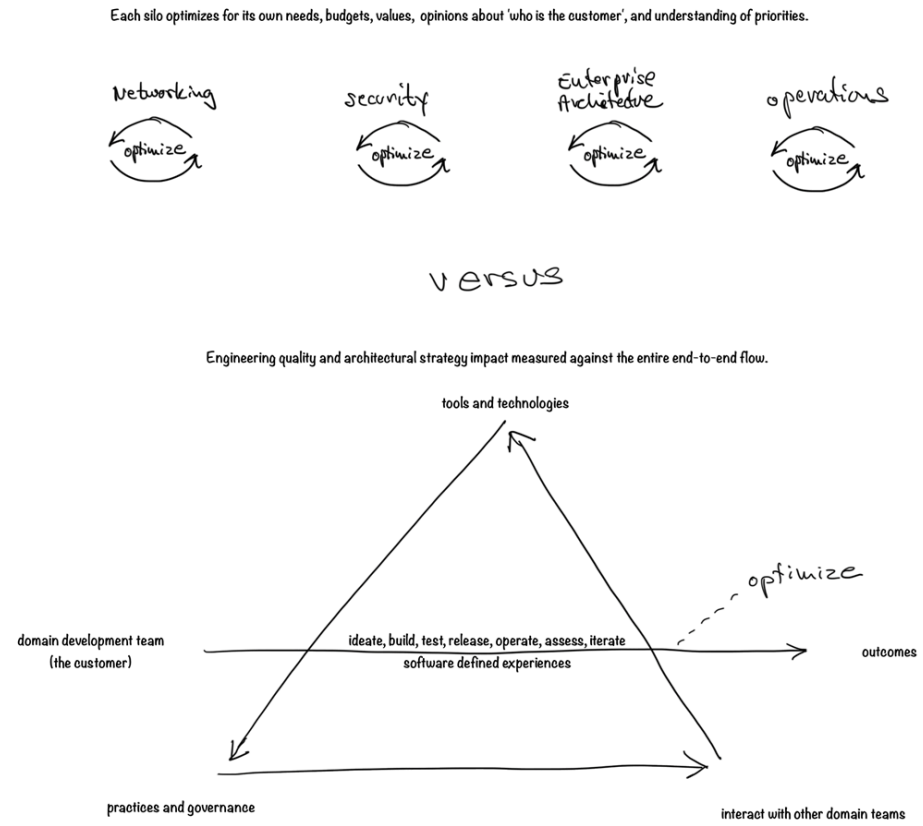

Figure 2.4 Look at all the activities, from ideation to operations, and assess engineering quality and impacts from organization or architectural decisions. This Developer Activity model is inspired by Yrjö Engeström's human activity model. [2.4c]

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.

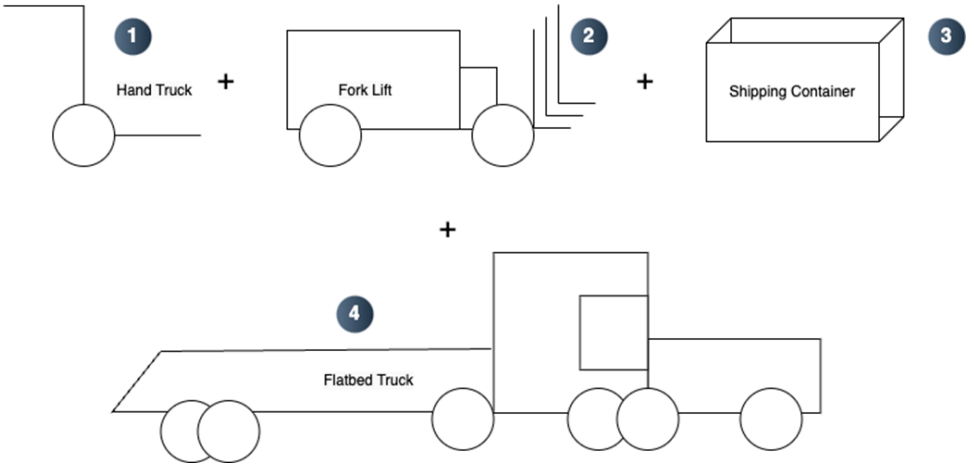

Figure 2.5 shows the Progression of an MVP delivery process. A fully functional product does not need to be delivered at the start, but all phases should deliver some incremental value to the customer.

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:

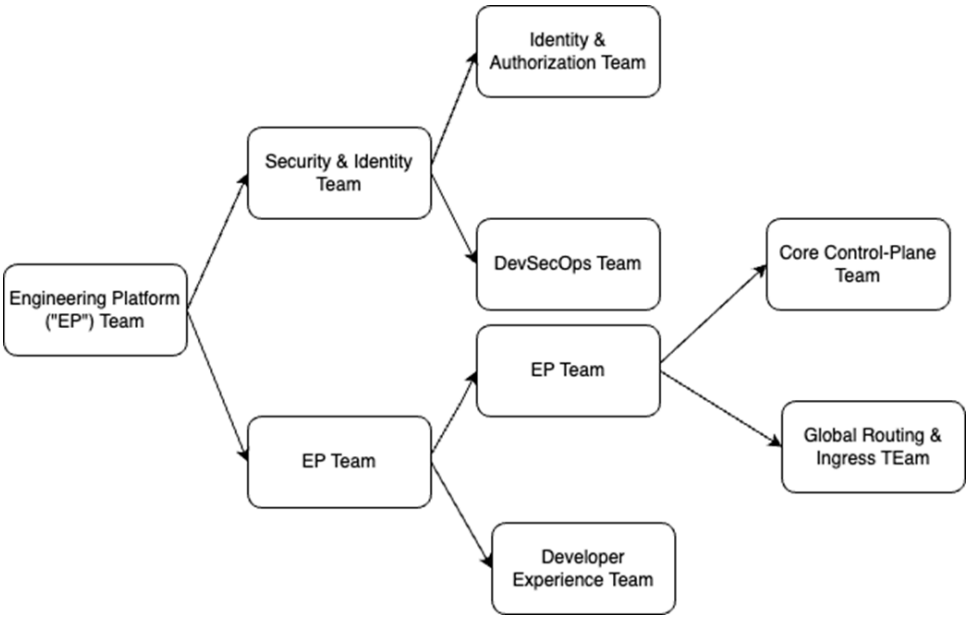

Figure 2.6 shows the evolutionary path of platform development teams. Often, while an EP starts as an effort driven by a single team, multiple teams spin up over time to handle specific aspects of platform functions, with all development efforts under a top-level product owner.

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

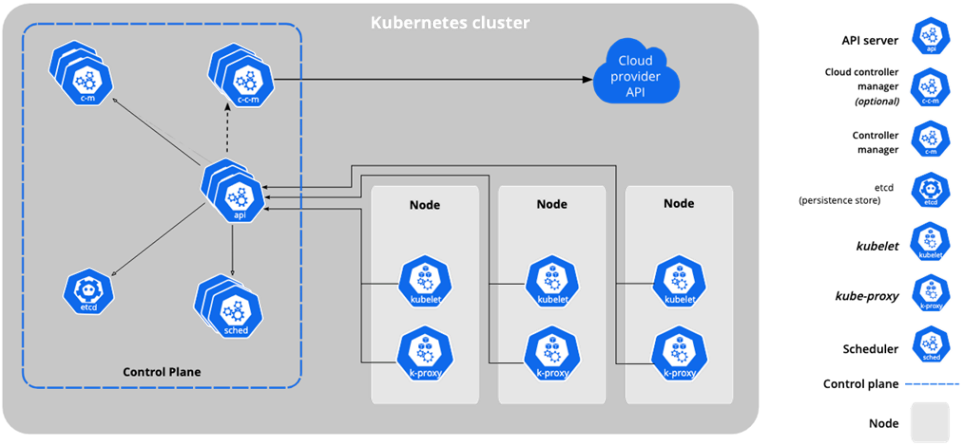

Figure 2.7 The components of a Kubernetes Cluster. Cloud PaaS offerings will usually serve the control plane as a service, leaving node management to the user. In "serverless" cloud offerings, worker nodes will also be offered as a service. Courtesy: https://kubernetes.io/docs/concepts/overview/components/

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.

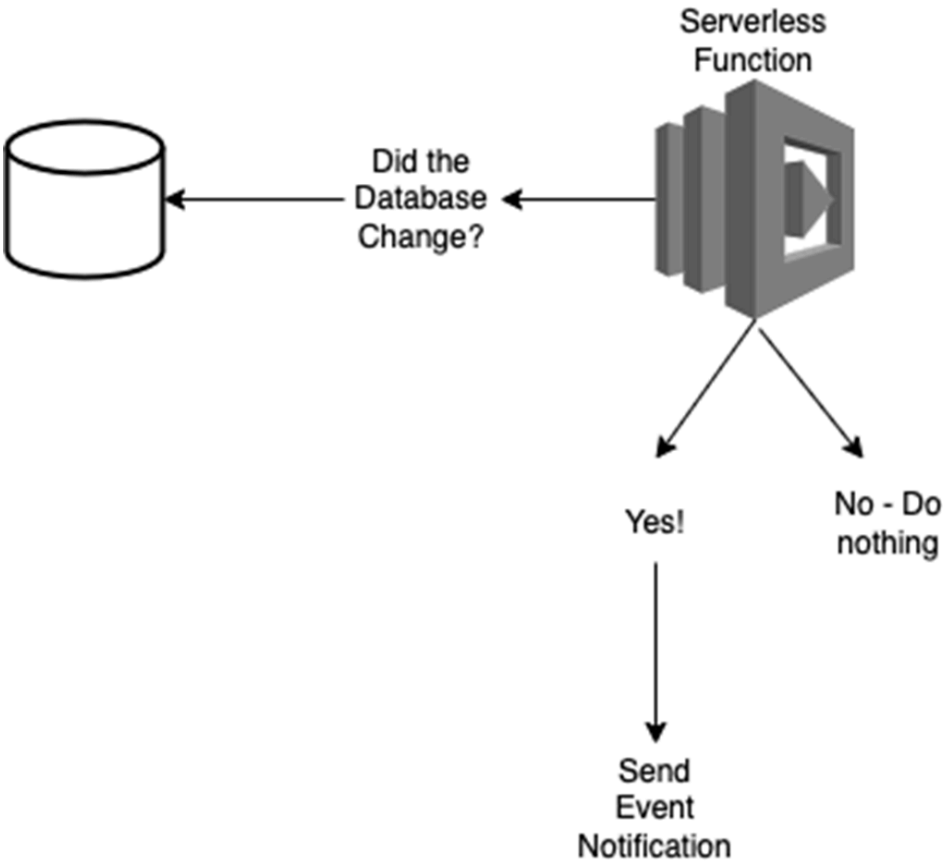

Figure 2.8 shows an example decision tree for when to use serverless functions. Serverless functions are good for lightweight, simple workflows that serve a single purpose.

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.

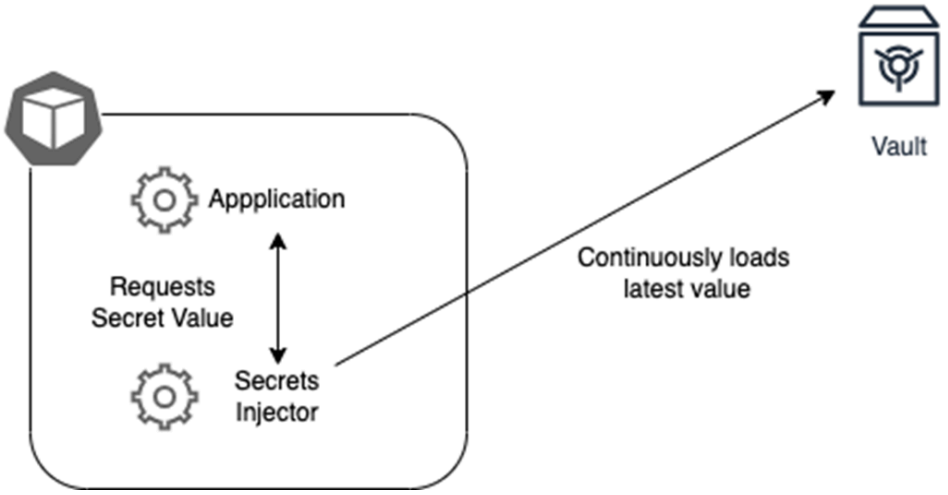

Figure 2.9 shows an example of a pod sidecar. This container runs a process alongside the application container to enhance functionality. In this example, a sidecar continuously loads a new secret value from a vault so that the application will always get the latest updates when secrets are requested.

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

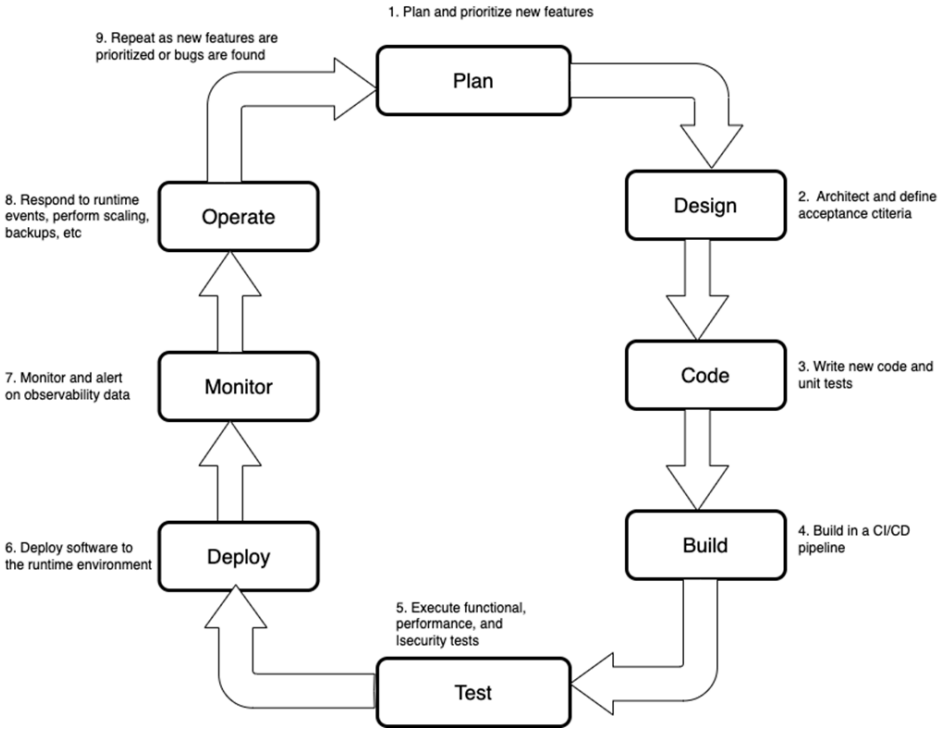

Figure 2.10 A Base Development SDLC covering all stages from feature planning through running in production. This can be used as a standard across the team to increase code quality for all releases.

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 codingincluding 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:

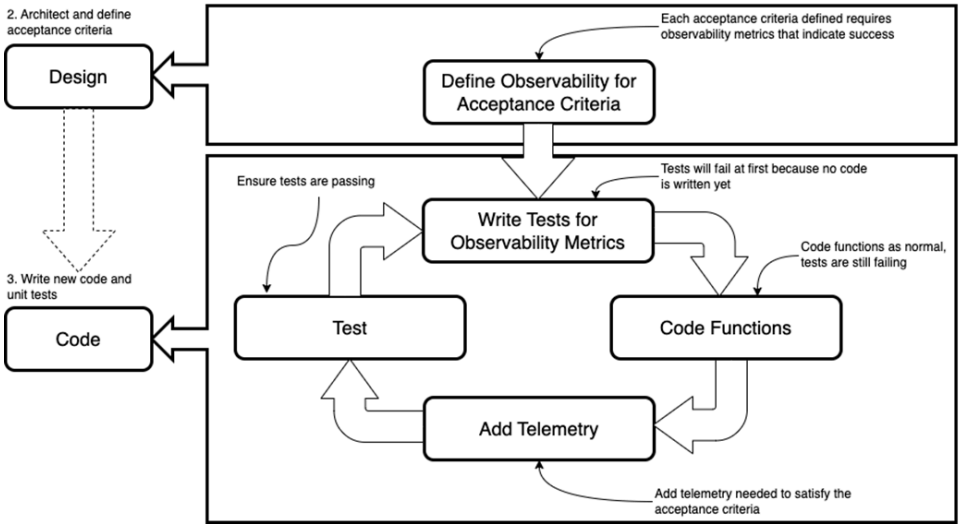

Figure 2.11 An Expanded SDLC that includes Observability-Driven Development (ODD) Practices. ODD can help ensure that a delivered feature is not only working but also working correctly and returning the expected value to the organization and its customers.

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 useability 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.

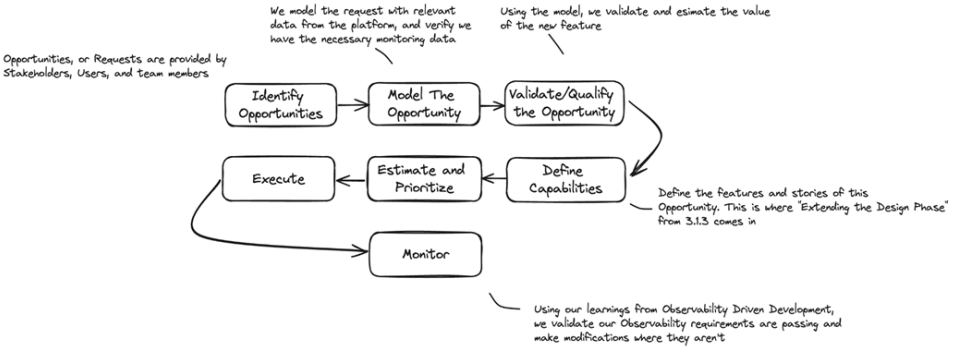

Figure 2.12 Platform Development Iteration Flow. Using a flow during iteration will focus development on how the short-term lifecycle fits into the bigger picture.

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.

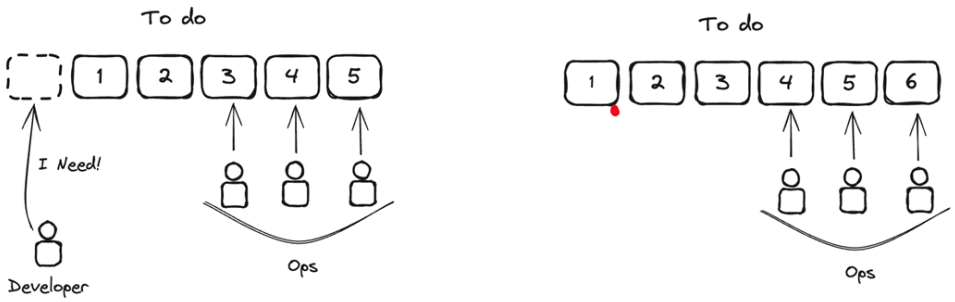

Figure 2.13 A standard Operations Team Ticket Queue. Developer Requests are immediately placed into the Todo pile.

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.

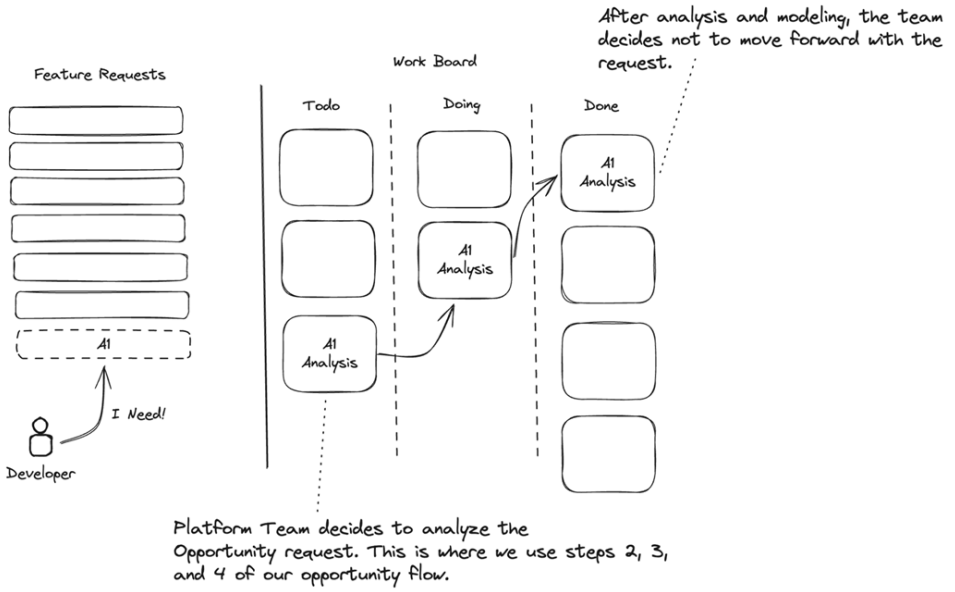

Figure 2.14 The Platform Development Ticket Queue. These get treated as product feature requests, to be analyzed, accepted and prioritized.

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 getfeedback 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.

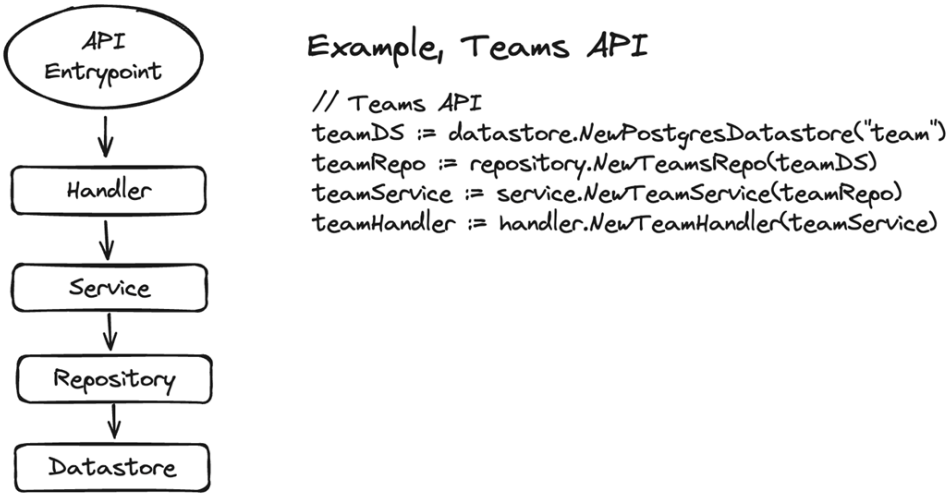

Figure 2.15 An example of defining the necessary APIs for a Datastore in an engineering platform (Teams API)

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 developers 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.

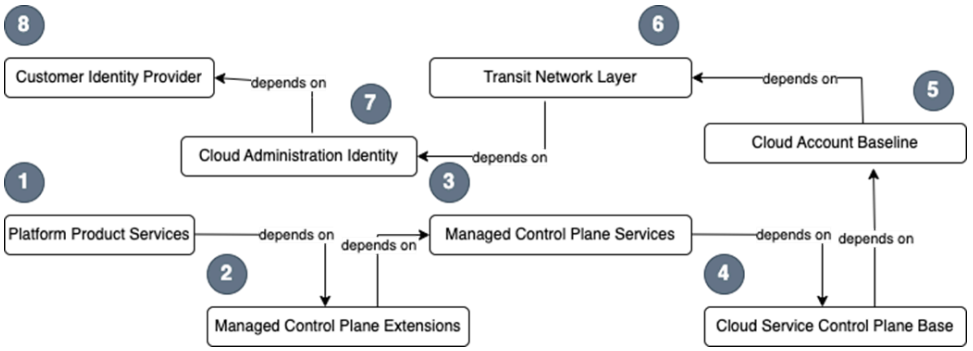

Figure 2.16 Expanded View of the Platform Engineering Mental Model first introduced in Chapter 1 is reproduced here for easy reference.

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.*

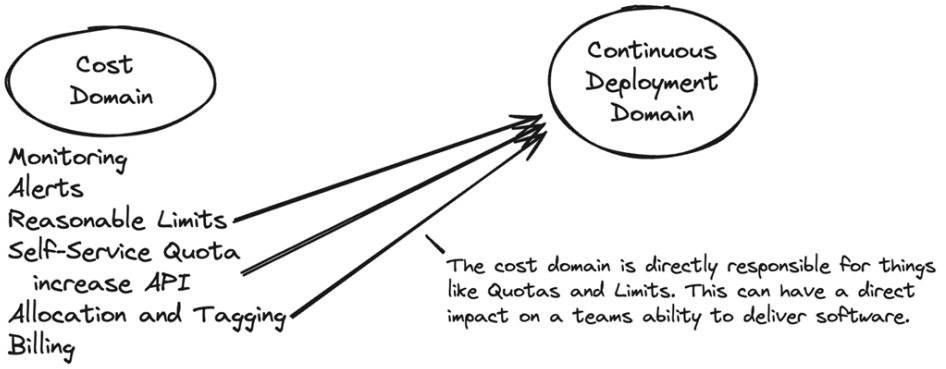

Figure 2.17 An example illustration of an ADR for cost domain aligned to the continuous deployment domain

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.

[1] 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.

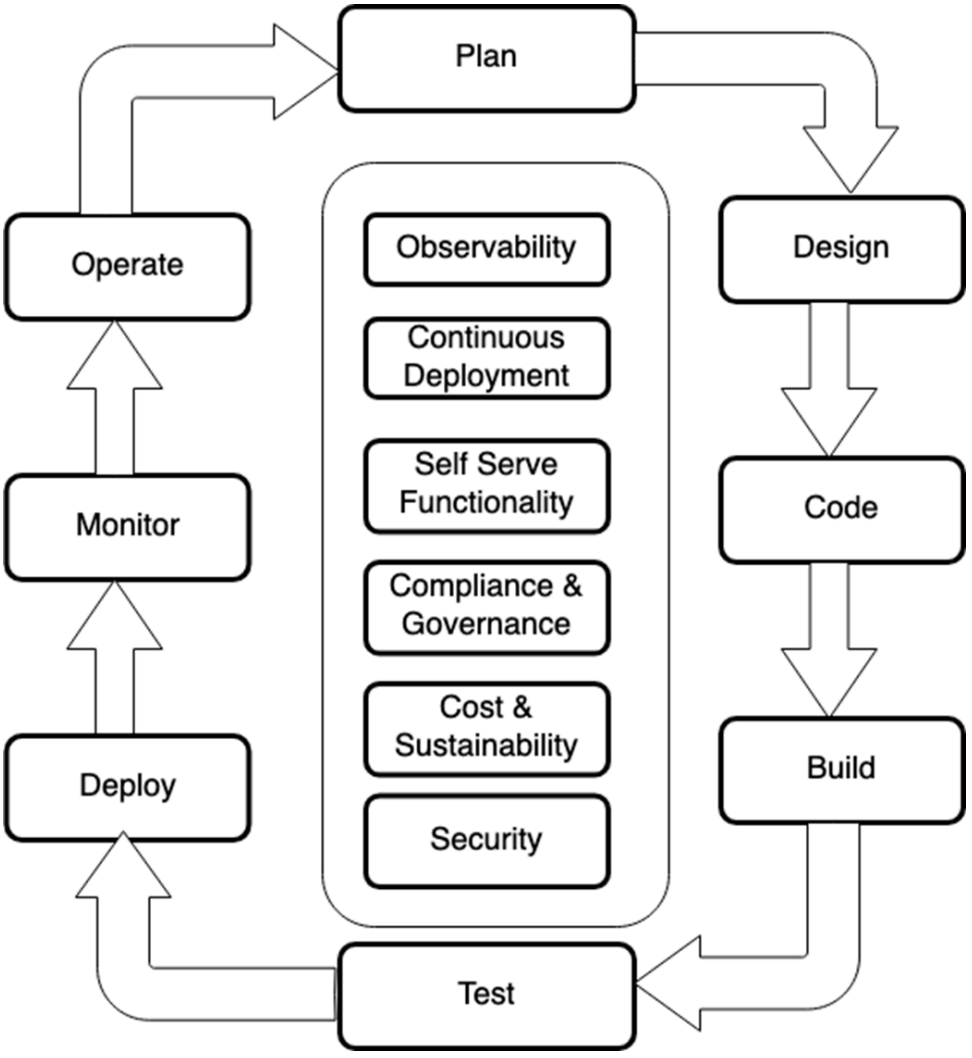

Figure 3.1 A high-level view of the SDLC in the context of platform principles. This figure indicates the fact that each of the platform principles is applicable for the steps in the SDLC.

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)

Plan	Design	Code	Build
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)

Test	Deploy	Monitor	Operate
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.

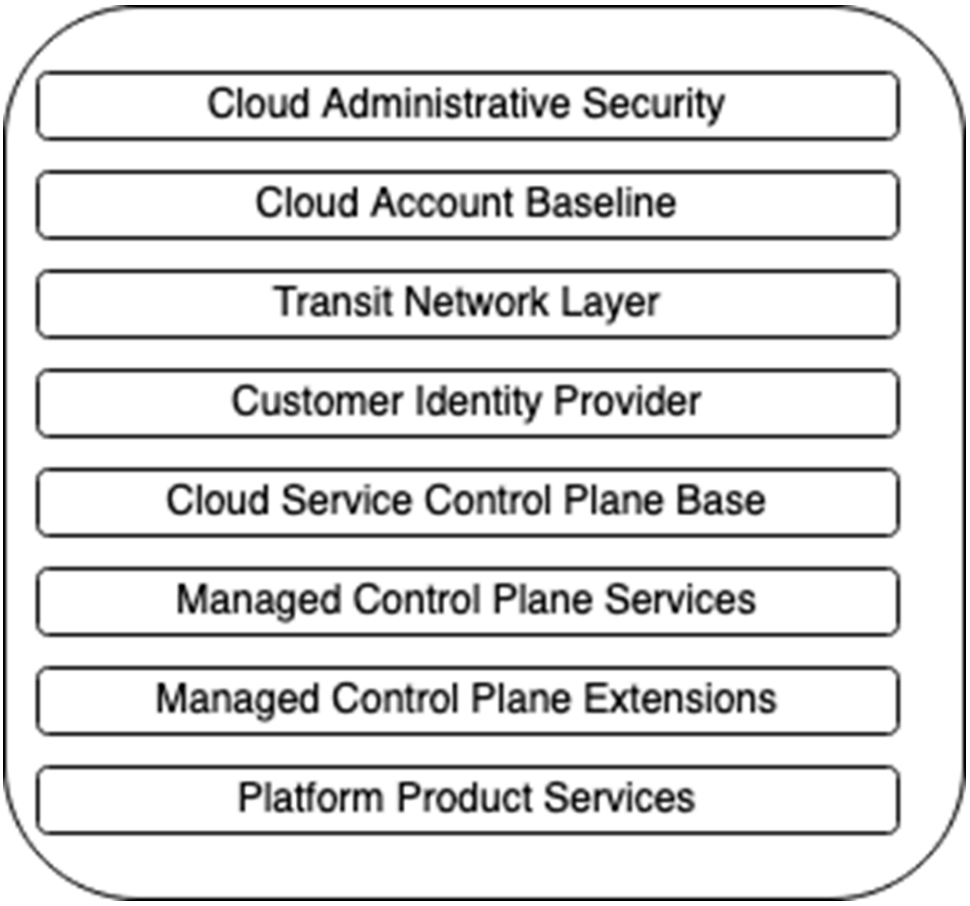

Figure 3.2 Engineering platform product domains. First introduced in Chapter 1 is reproduced here for demonstrating what should be measured for each of these domains

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.

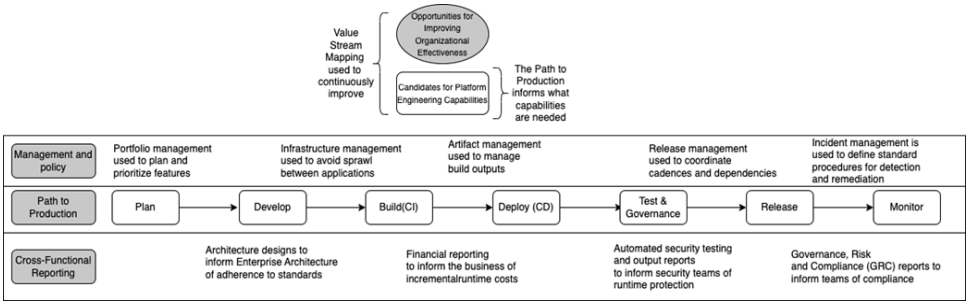

Figure 3.3 shows An Example of Value Stream Mapping (VSM) and path-to-production (P2P). This is used to identify areas that can be improved by optimizing or eliminating the chain of dependent steps in creating and deploying new features.

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.

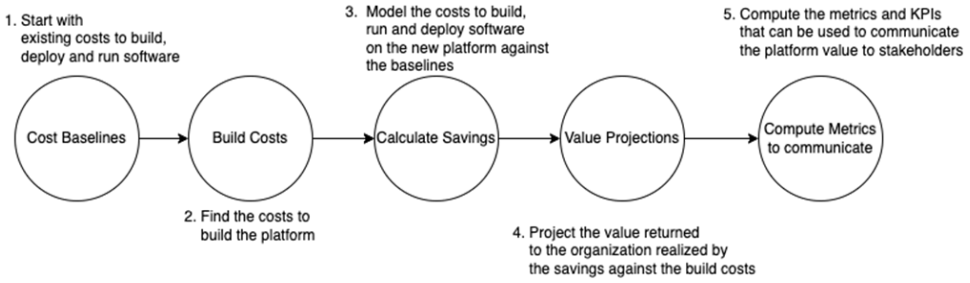

Figure 3.4 shows the value modeling process. This can estimate the value the organization will see from building and using an engineering platform. Actual metrics and KPIs should result from this, which can be communicated to business stakeholders.

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.

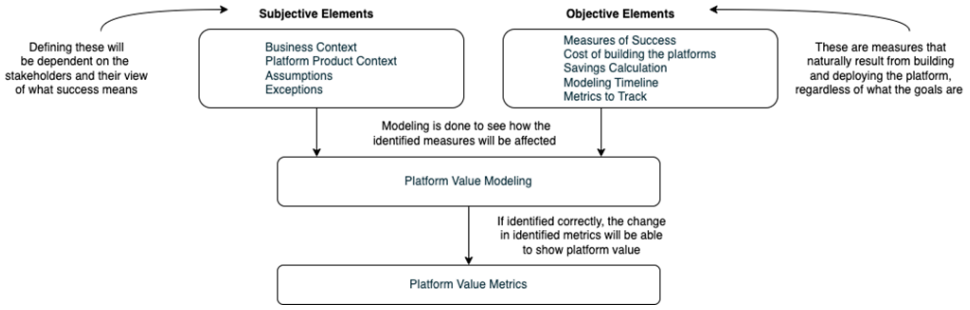

Figure 3.5 Components of Value Modeling Process. These are both subjective and objective measurements that a TPO should identify by working with stakeholders and engineers that will be used to show platform value.

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.

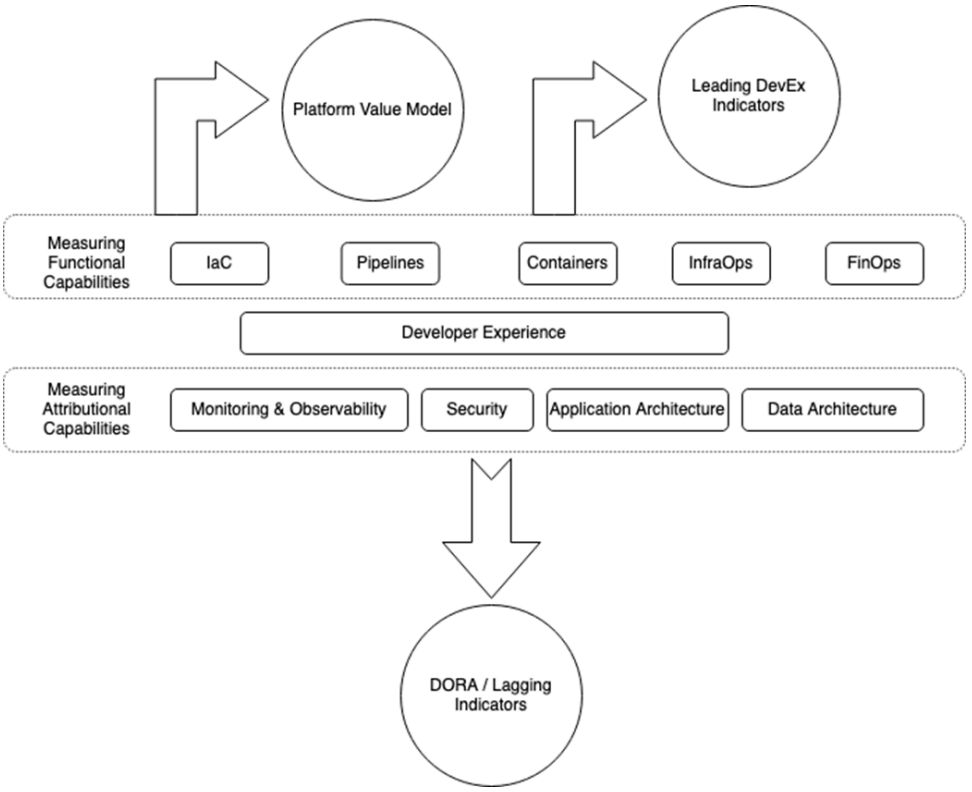

Figure 3.6 Platform engineering capability model. This capability model has 3 parts that talks about functional capabilities, attributional capabilities and developer experience between those two. Developer Experience components are described below

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.

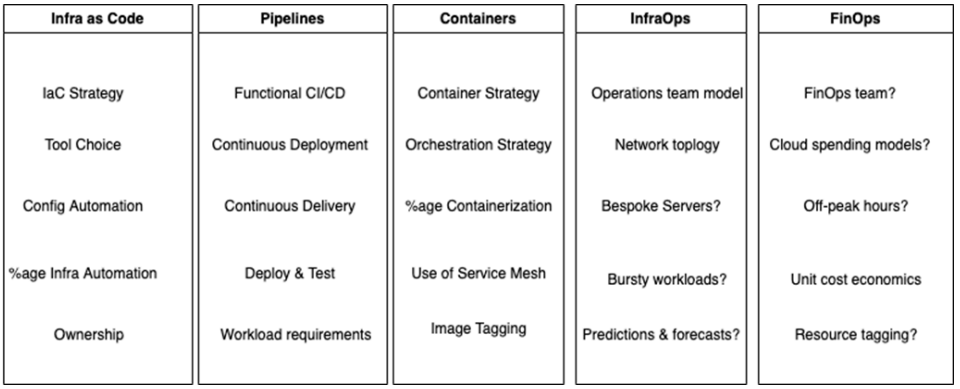

Figure 3.7 Functional capability considerations for each of the five areas of focus we described in the top layer of figure 3.6.

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.

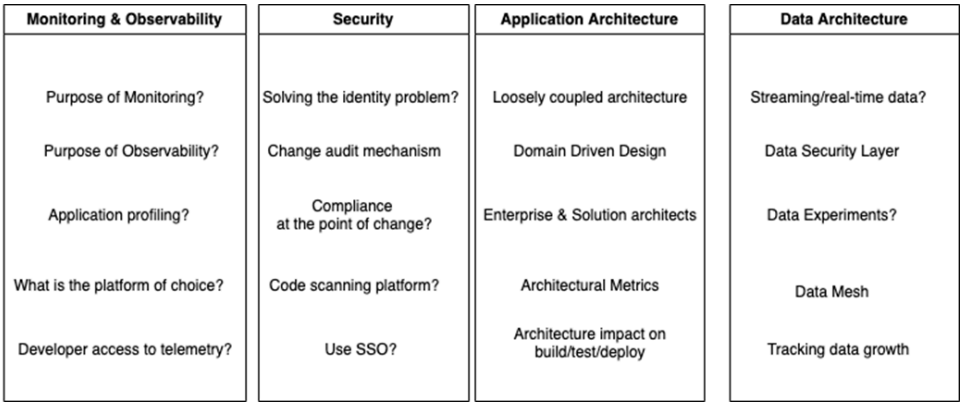

Figure 3.8 Attributional capability considerations for each of the four areas of focus we described in the bottom layer of figure 3.6

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.

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

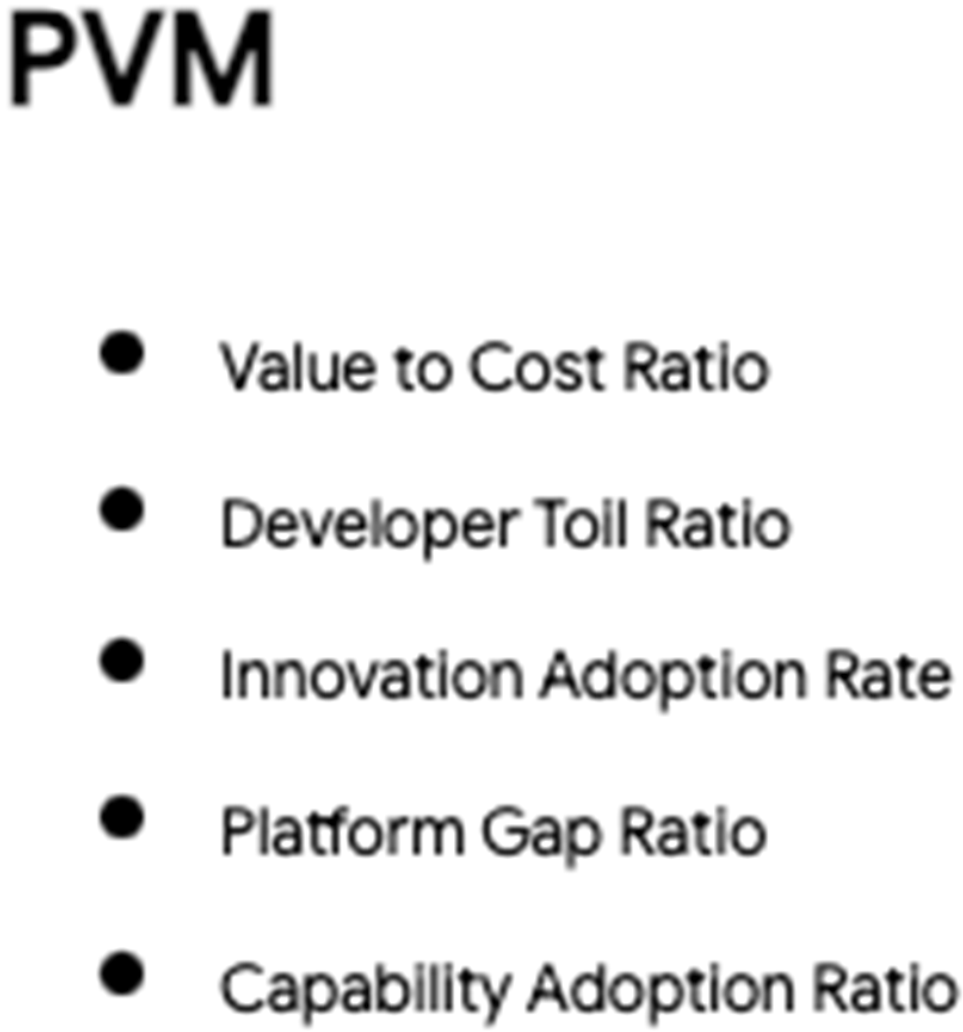

Figure 3.10 Platform Value Metrics (PVM). An organization can strategically determine several PVM considerations. This example shows five of the key ones we believe every organization should use

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

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.

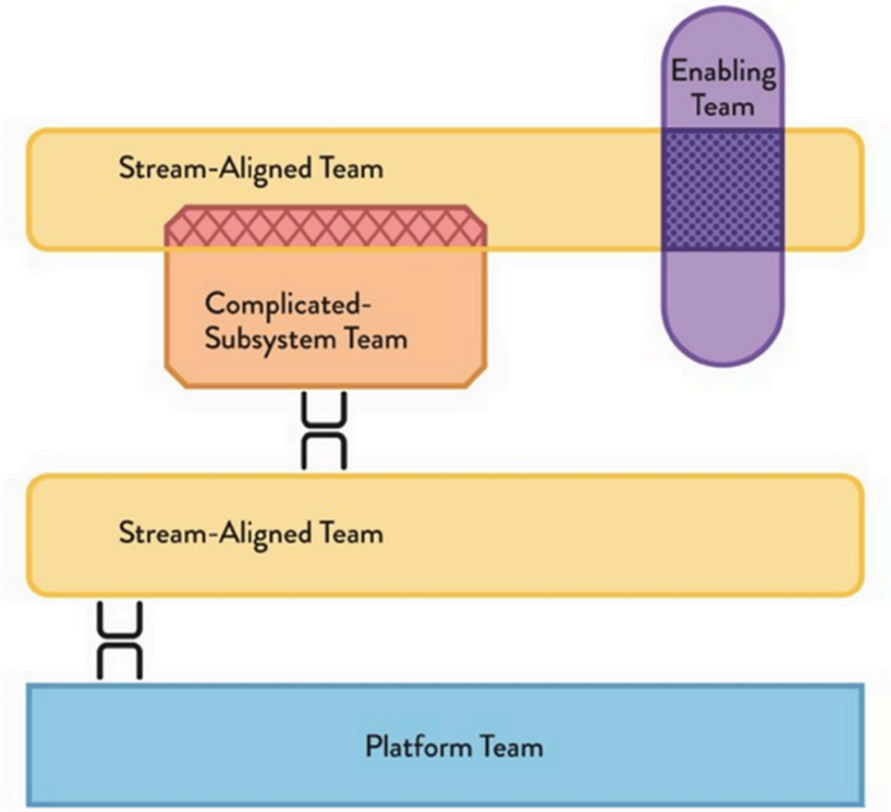

Figure 3.11 Primary Interaction Modes for the Four Fundamental Team Topologies.(Image taken from the book *Team Topologies* by Matthew Skelton and Manuel Pais, 2019. Used with permission.)

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.

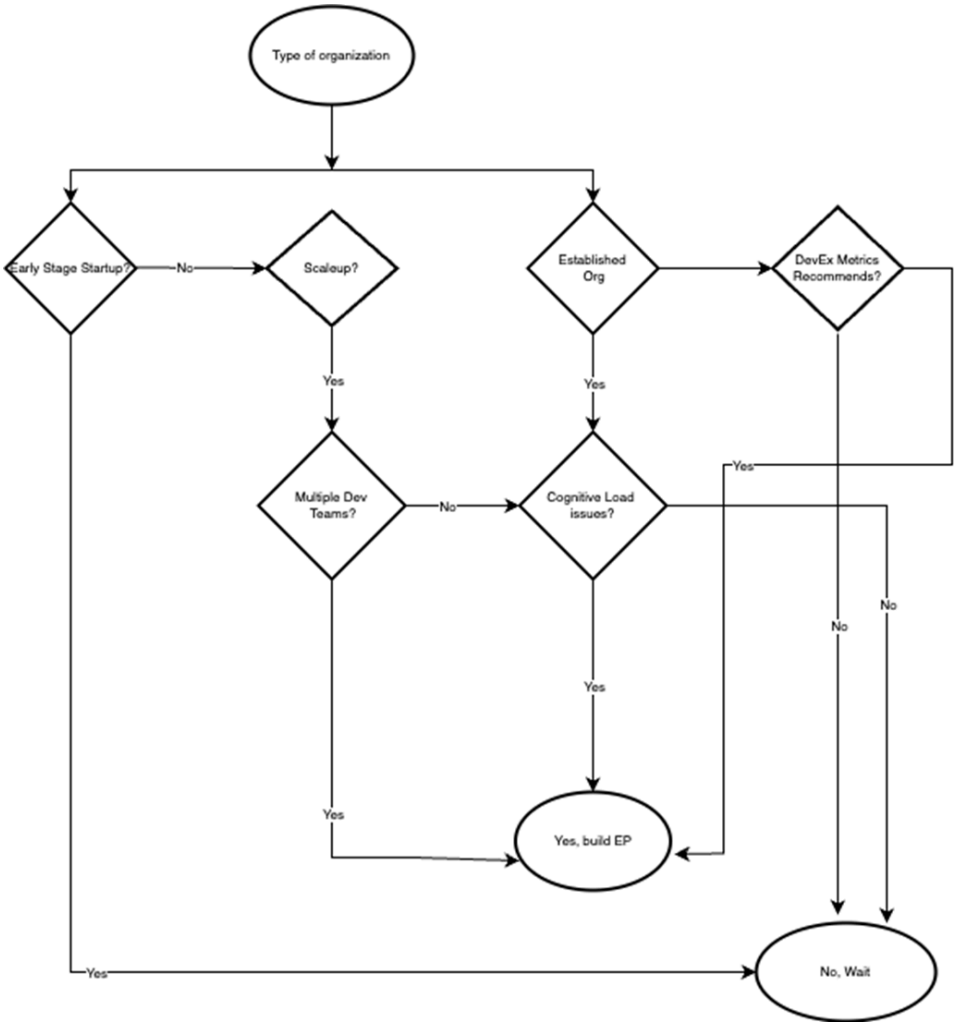

Figure 3.12 A Decision Tree for Engineering Platform Buildout. This is a question that is often asked whether an organization is ready for an engineering platform adoption. This simple to use decision tree gives you clarity in making that decision.

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

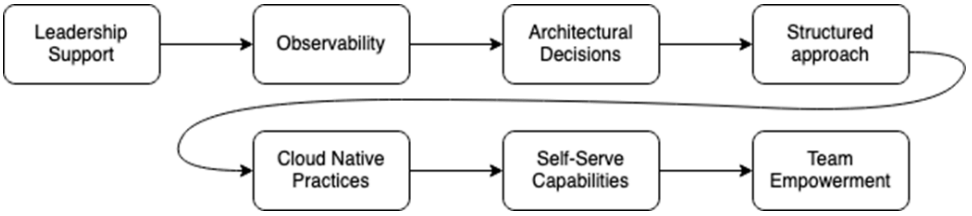

Figure 3.13 Key prerequisites for scaling platforms. Each of the seven prerequisites indicated here are all equally important in deciding to scale the platforms.

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-serve 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-serve 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

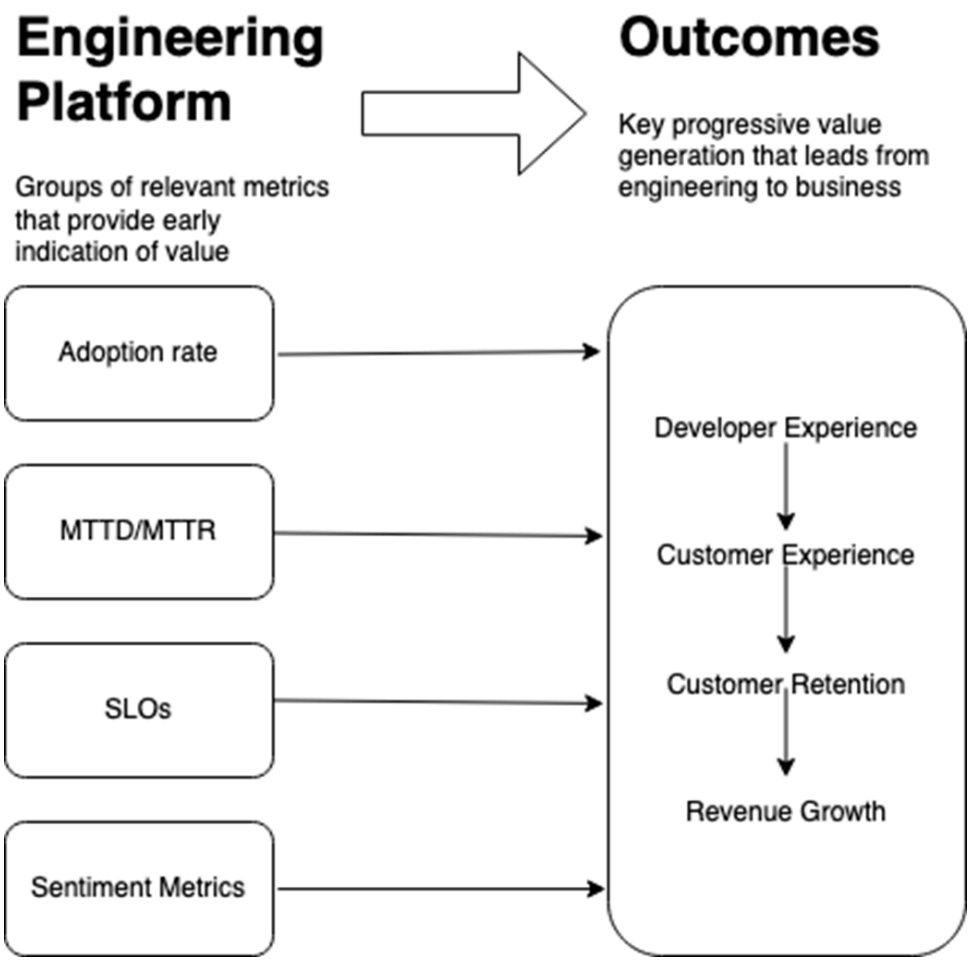

Figure 3.14 A mental model for platform KPIs. The four KPIs indicated on the left side for the engineering platform drives the business outcomes every business should care about.

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 t*otal 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 miniscule 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:

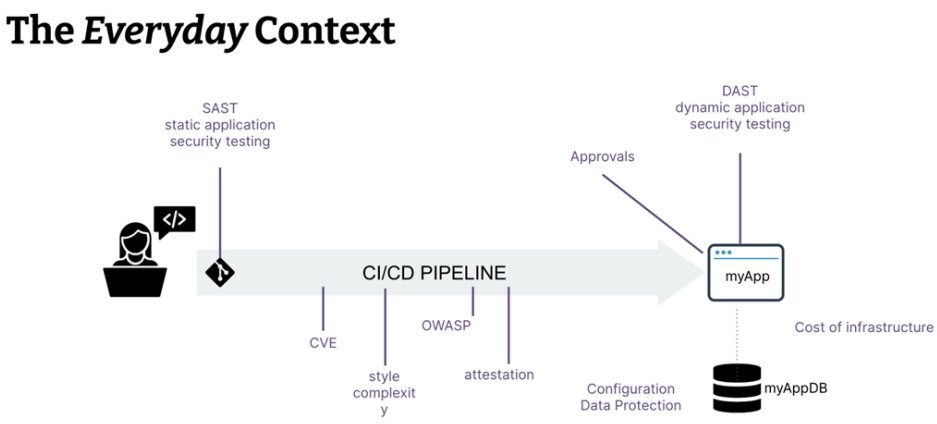

Figure 4.2 The Everyday Pipeline

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:

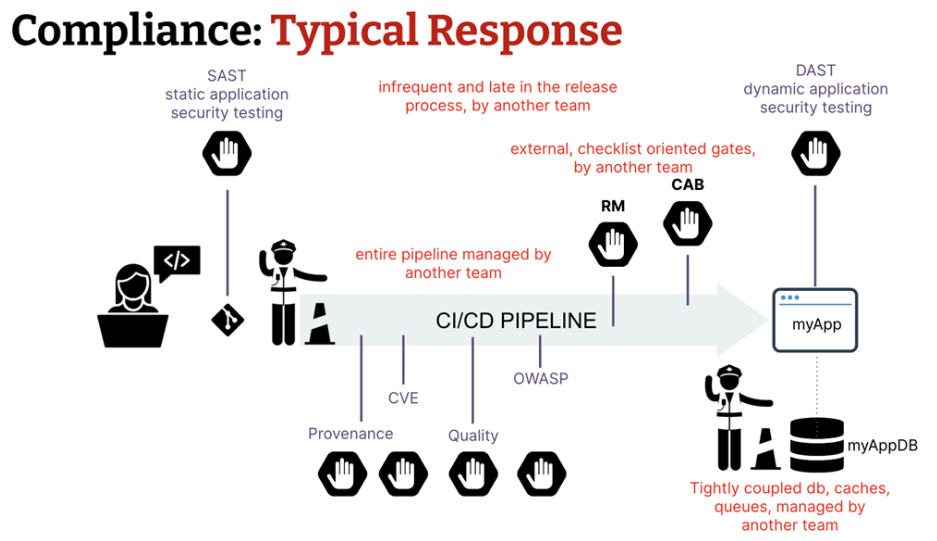

Figure 4.3 The reality of the everyday pipeline

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.

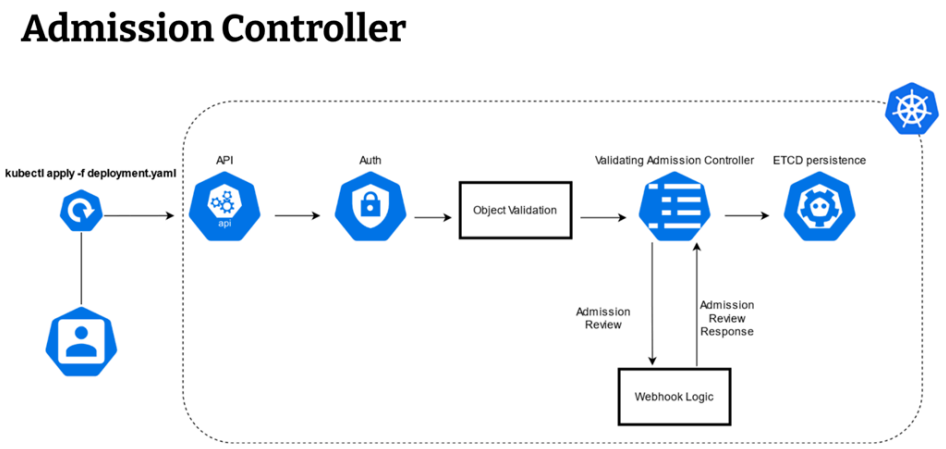

Figure 4.4 Kubernetes Admission Controller

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 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.

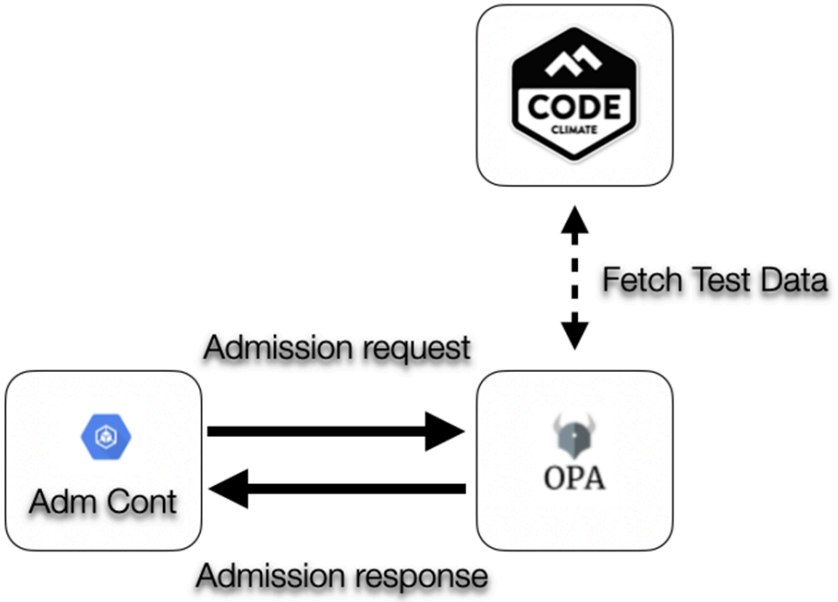

Figure 4.6 Example of a Kubernetes Admission Controller

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 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).

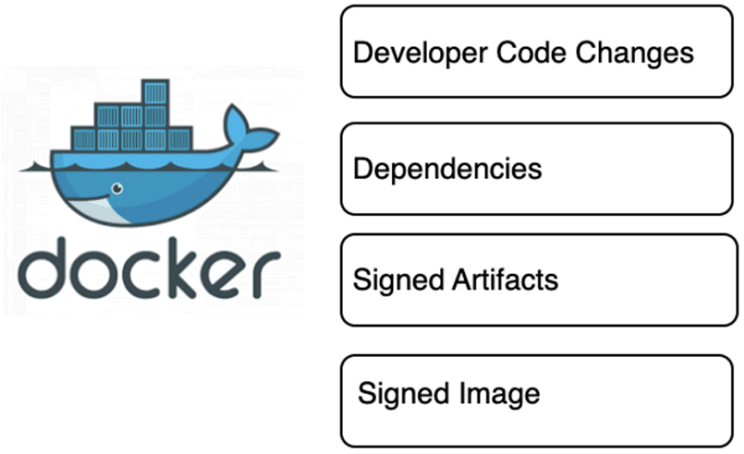

Figure 4.7 Docker Images are made of many layers.

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.

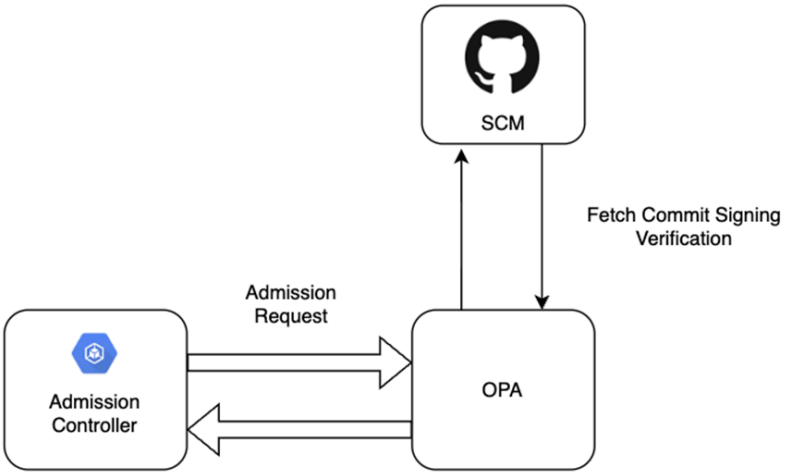

Figure 4.8 Commit Signing is pulled from the SCM

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?

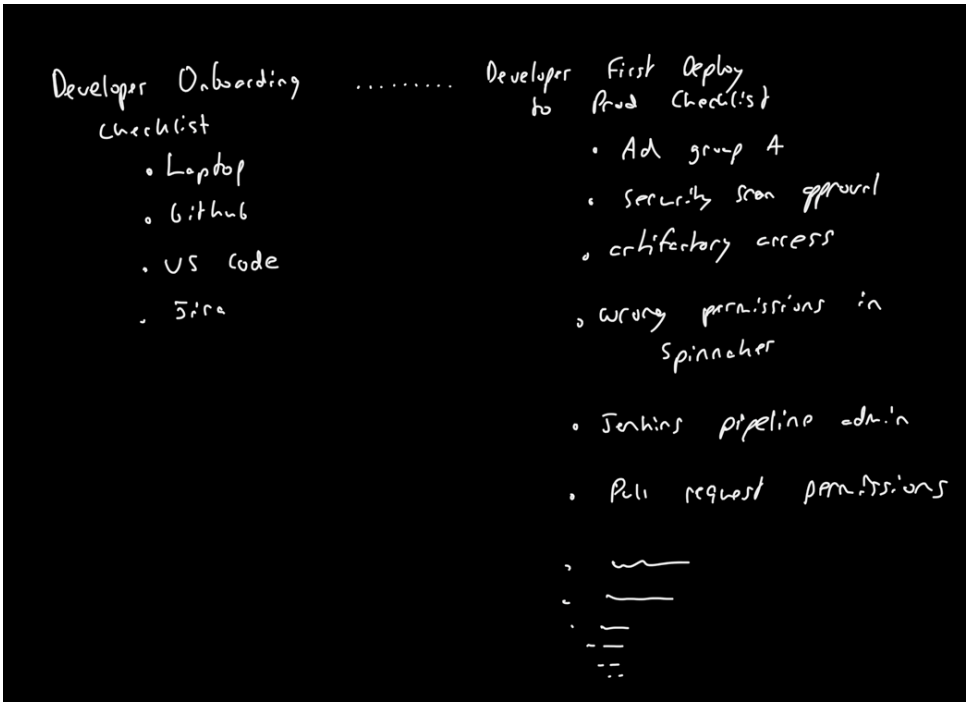

Figure 4.9 "Onboarding" vs Time to First Deployment Reality. The number of steps involved causes this time to be long when manual steps and cross-functional check-ins are required.

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.

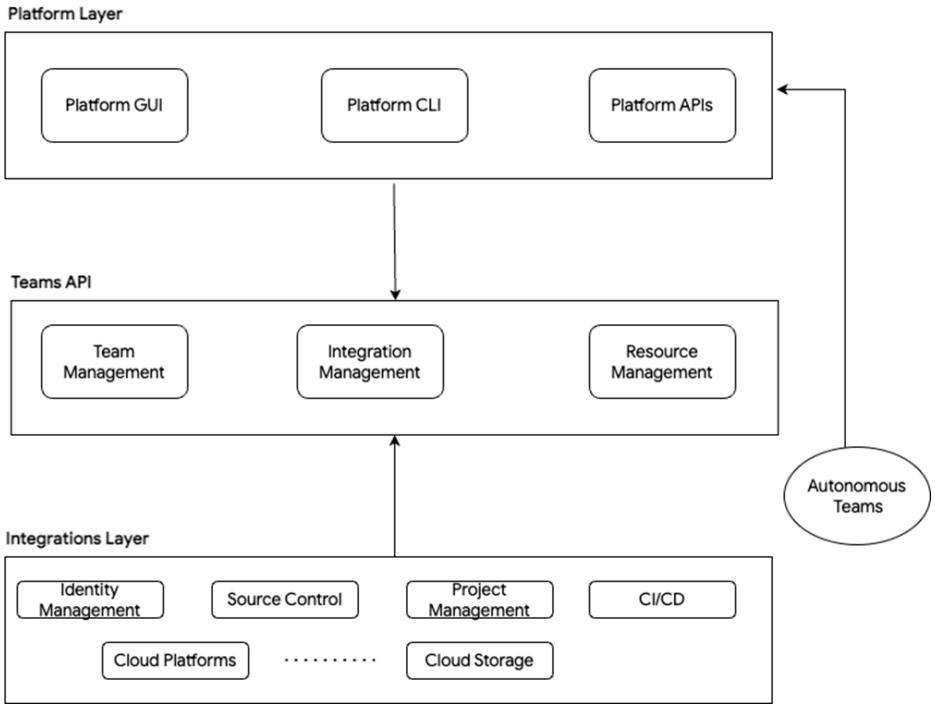

Figure 4.10 Shows the high level concept of Teams API with the platform layer and integrations layer, and the separation of concerns from cloud vendor details.

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.

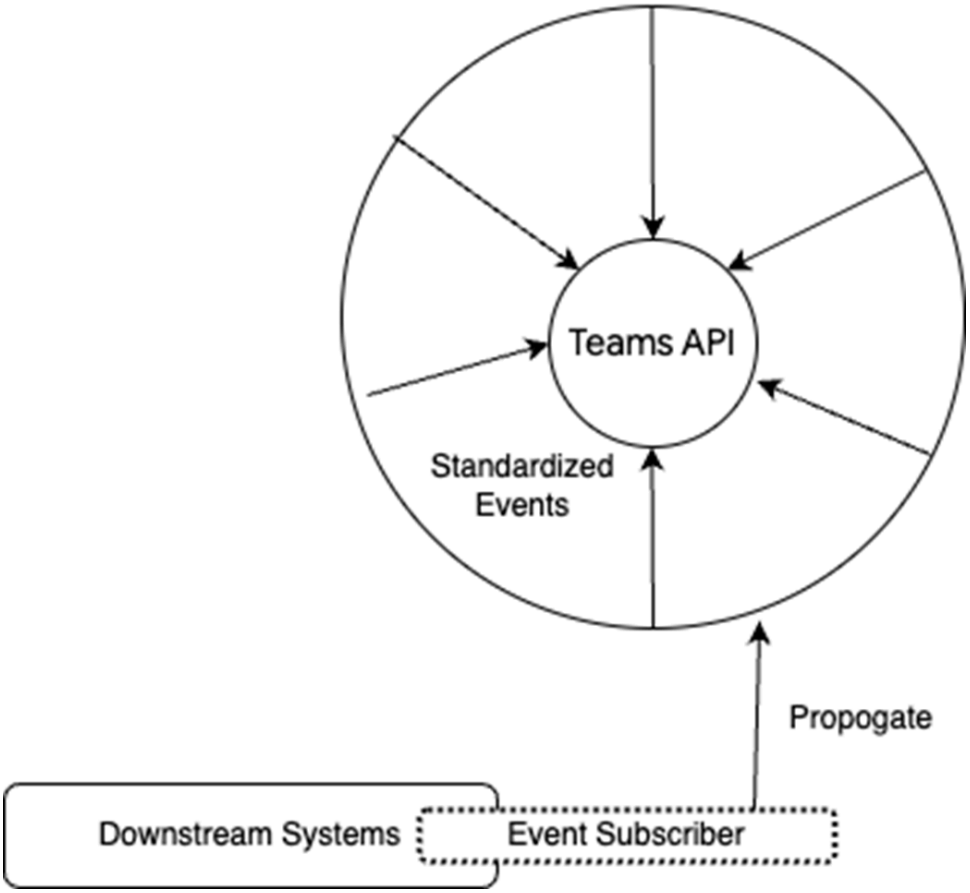

Figure 4.11 shows how the downstream systems can propagate the events through a subscriber mechanism

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:

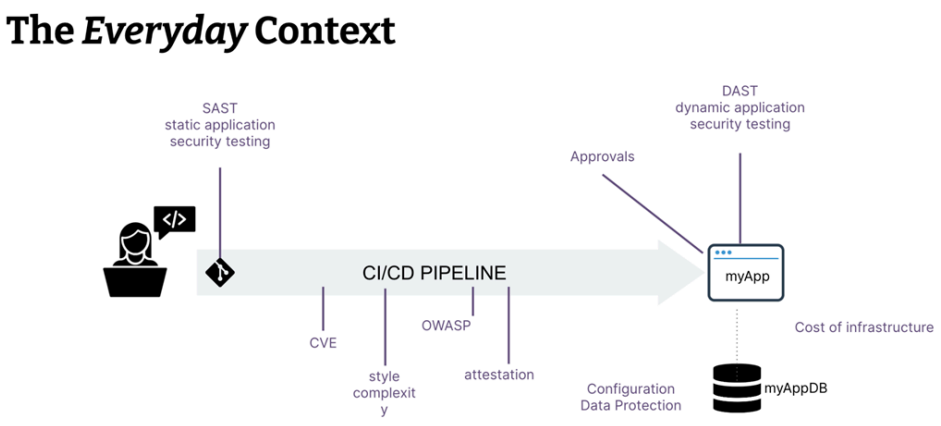

Figure 4.12 The Everyday Pipeline

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?

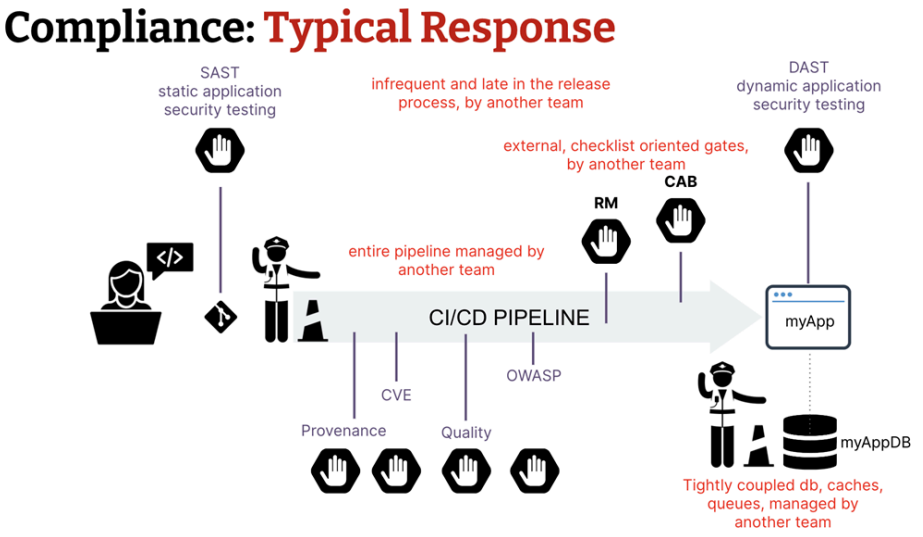

Figure 4.13 The reality of the everyday pipeline

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.

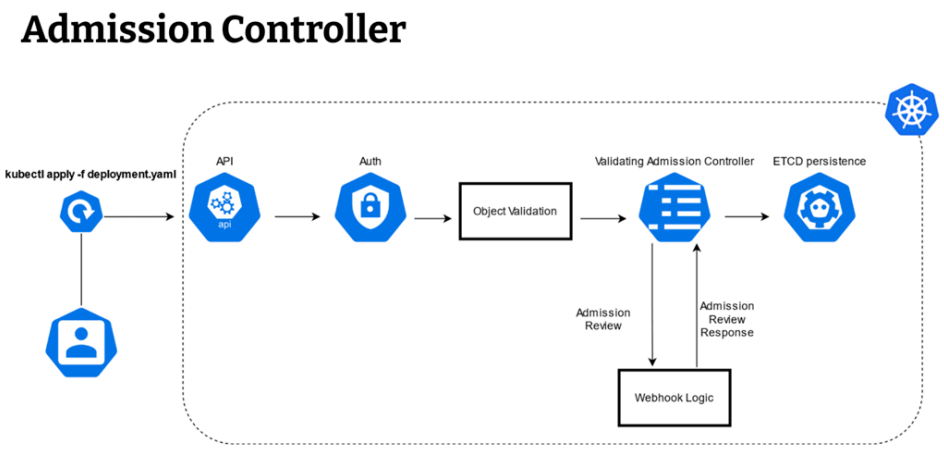

Figure 4.14 Kubernetes Admission Controller

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

```
meshCongif.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, Flanal, 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.

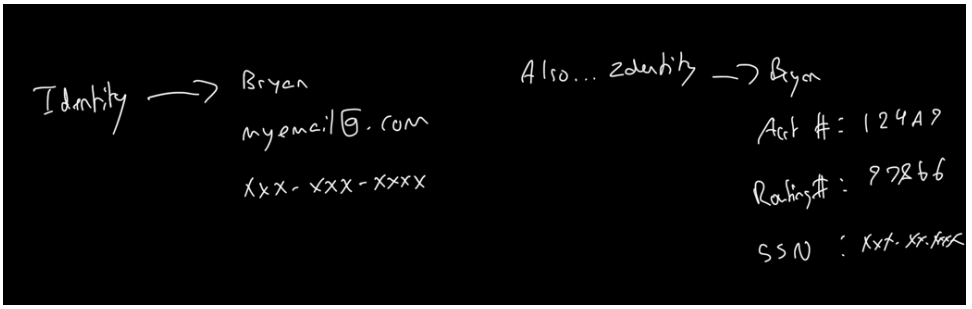

Figure 4.15 An example of an identity decoupling

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.

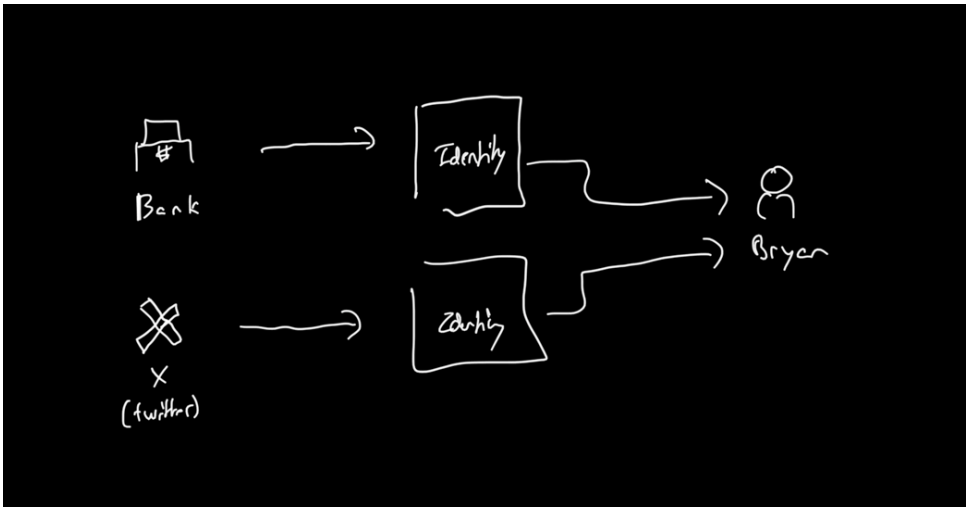

Figure 4.16 An example of an identity evolution

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.

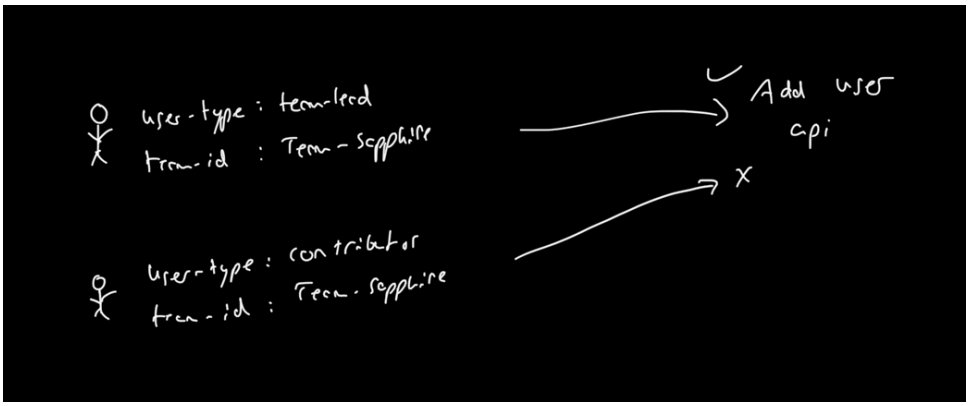

Figure 4.17 User Attribute Policy

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.

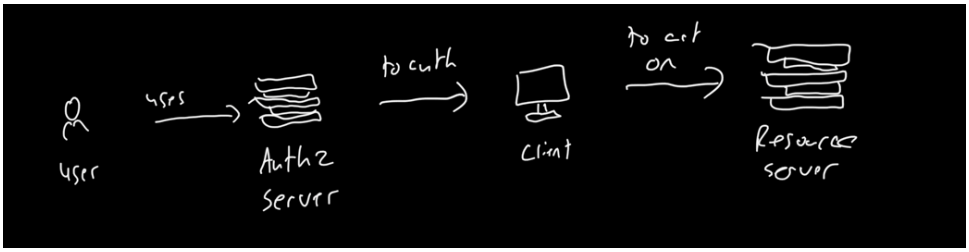

Figure 4.18 A recommended flow of attribute relationships

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.

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.