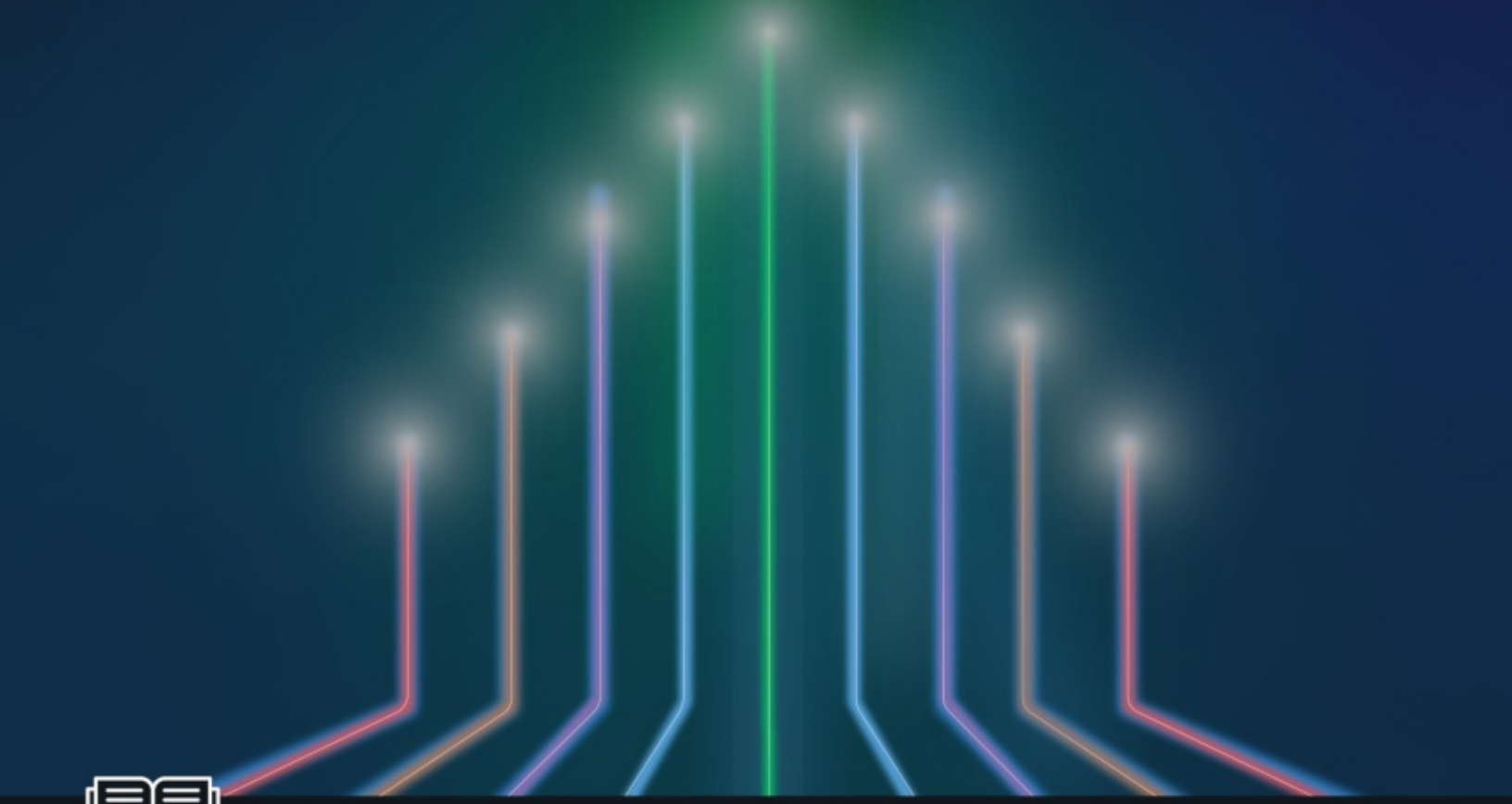
DevEx as a Service

MATTHEW CASPERSON



First edition

Octopus Deploy

Version: 1.0.44

Table of contents

ISBN-13 (paperback): 979-8866462209

Conventions

This book shares many real life anecdotes to help illustrate a point or provide additional context.

These anecdotes are indicated by the following icons, and written in italics:

This is a real life story or example that relates to the main text.

Feedback

This book is developed as an open source project.

The book's "source code" is available in a public GitHub repository (https://oc.to/KkIBCC).

We also have an open discussion for feedback (https://oc.to/HneyQx).

Executive Summary

Perhaps the best summary of DevOps is a quote from Werner Vogels, Amazon's CTO, who said in 2006 "You build it, you run it." In 2006 AWS comprised 3 services: EC2, S3, and SQS, and it was possible to hold the entirety of AWS in your head. Today, AWS has hundreds of services, and is just one of many cloud platforms. We've long passed the point where any individual or team can be expected to hold such complexity in their heads. And yet "you build it, you run it" remains synonymous with DevOps.

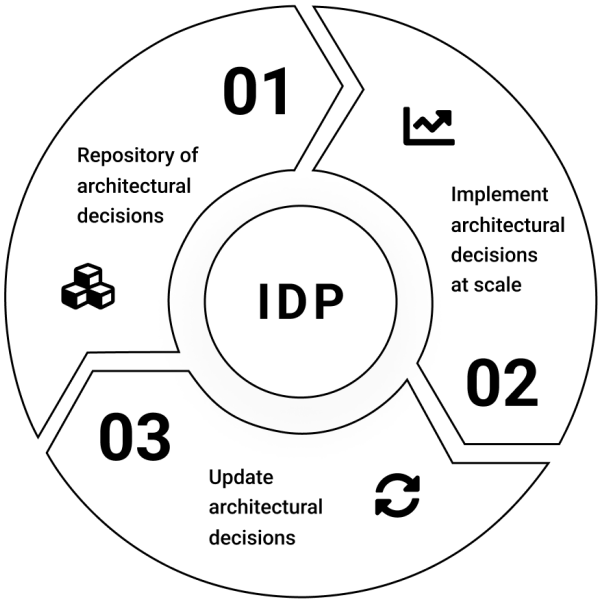
Meanwhile, DevOps has grown to encompass much more than simply building and running technical solutions. DevOps teams are responsible for meeting performance metrics defined by research organizations like DevOps Research and Assessment (DORA) while also supporting high performing team structures such as those defined by Westrum's organizational culture. The concept of DevOps has proven so popular and grown to encompass such a wide range of disciplines, tools, and processes that it is no longer possible to define where DevOps ends.

Since DevOps doesn't place any limits on a DevOps team's responsibilities, DevOps teams must take responsibility for defining a set of architectural decisions to guide the implementation of their projects and processes and divide work amongst individual specialists.

Platform engineering is a strategy for DevOps teams to define, share, and implement architectural decisions designed to provide common solutions to common problems. This is done via an Internal Developer Platform (IDP). An IDP has 3 requirements:

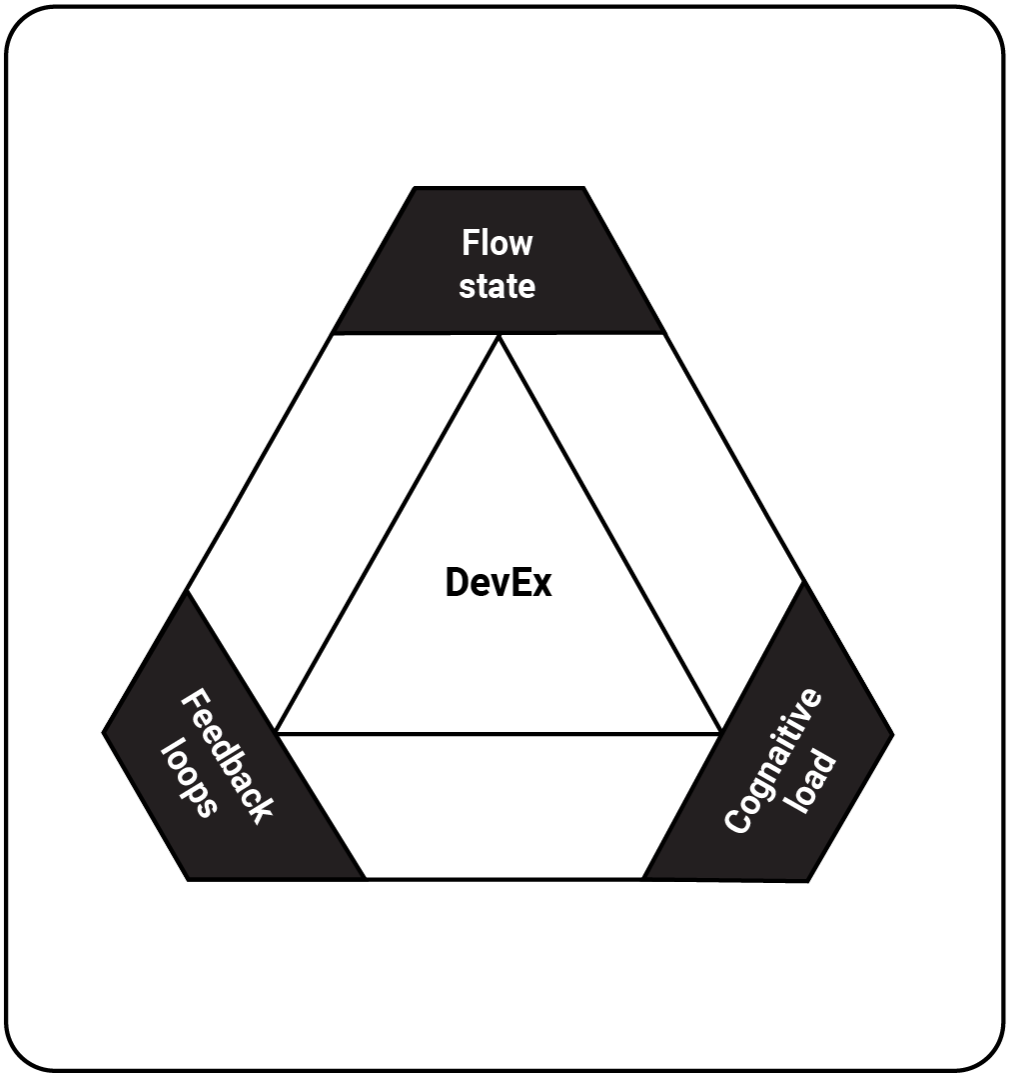
- Provide a central repository of architectural decisions made by DevOps teams
- Provide the ability to implement architectural decisions throughout DevOps teams at scale
- Implement feedback processes that allow architectural decisions to be improved over time

Lifecycle of architectural decisions



Ultimately, the goal of platform engineering teams is to improve the productivity and well-being of DevOps team members. This is commonly referred to as Developer Experience, or DevEx. More specifically, DevEx is measured by three dimensions, as defined by the paper "DevEx: What Actually Drives Productivity":

- Flow state
- Cognitive load
- Feedback loops



DevEx as a Service (DEaaS) is the practice of platform engineering with the explicit goal of improving DevEx.

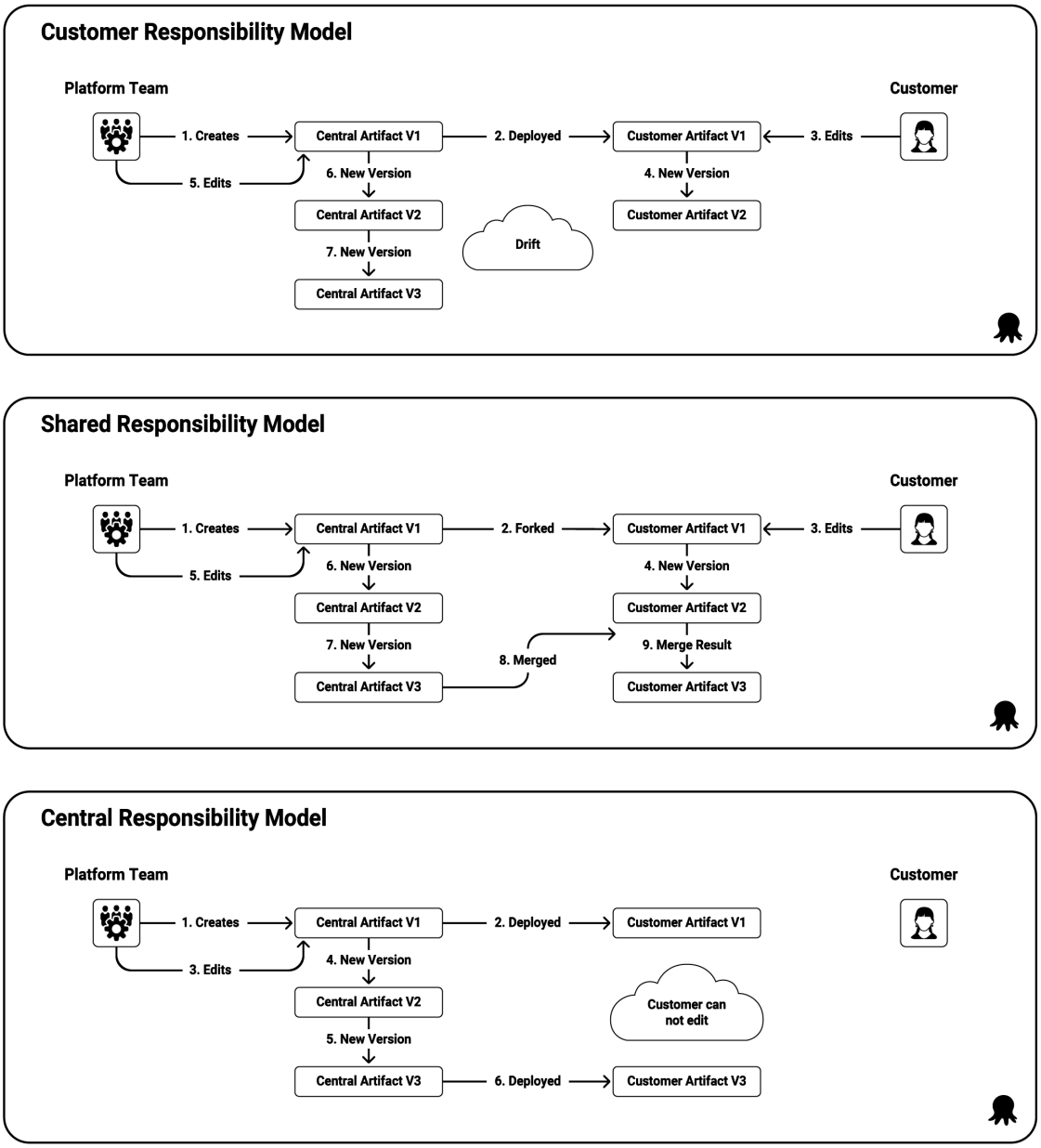
DEaaS is also inspired by other "as-a-service" paradigms such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

One of the defining characteristics of these services is that they clearly articulate what you are responsible for and what you are not. For example, IaaS customers know that they are not responsible for purchasing hardware or racking servers. Likewise, SaaS customers know they are not responsible for patching the software.

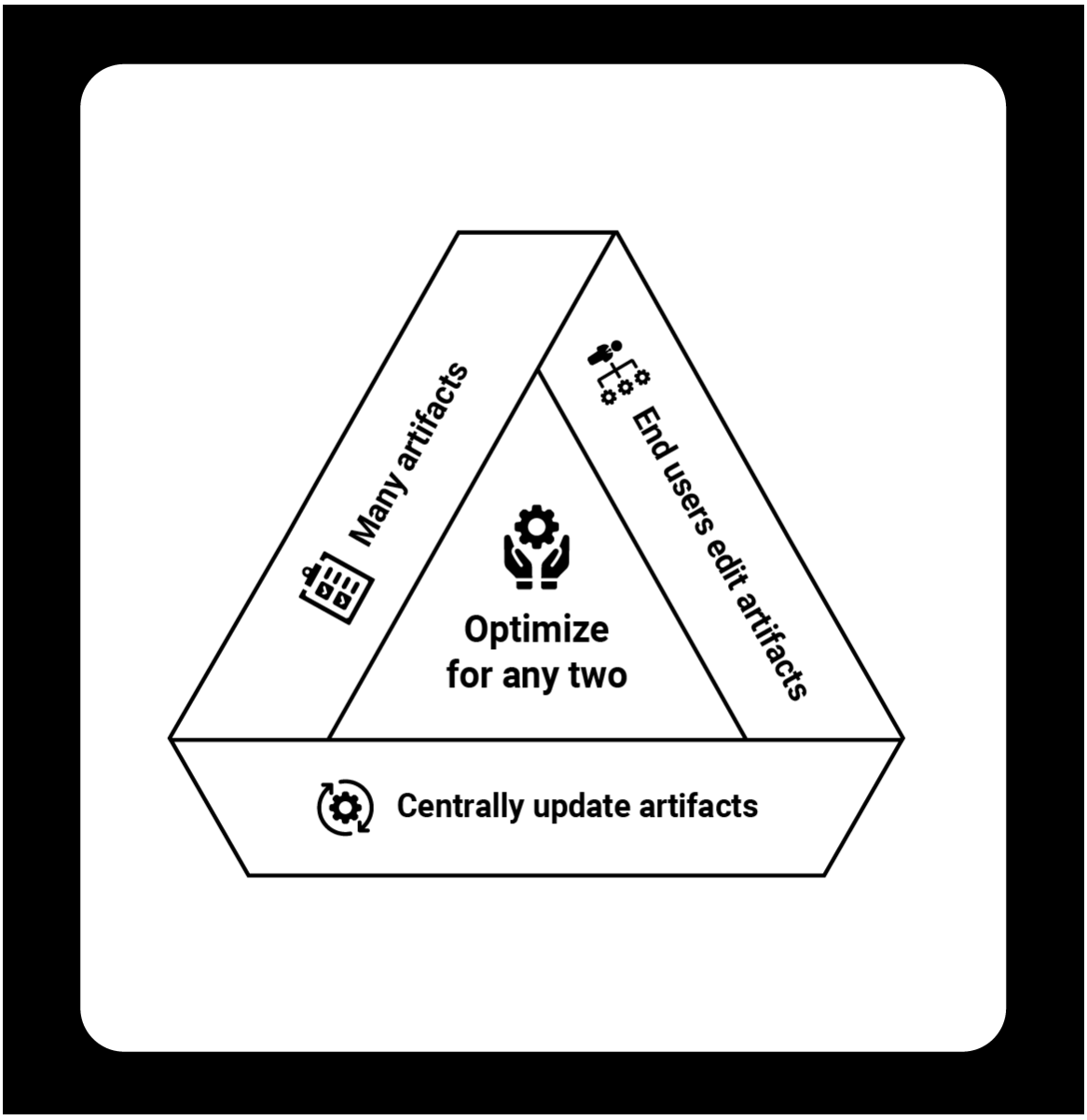
In the same way, DEaaS explicitly defines the responsibilities associated with the various artifacts generated by the IDP. There are 3 responsibility models:

- Customer responsibility, or eventual inconsistency, where artifacts are provided to a customer, who then takes ownership of the artifacts and modifies them for their own needs

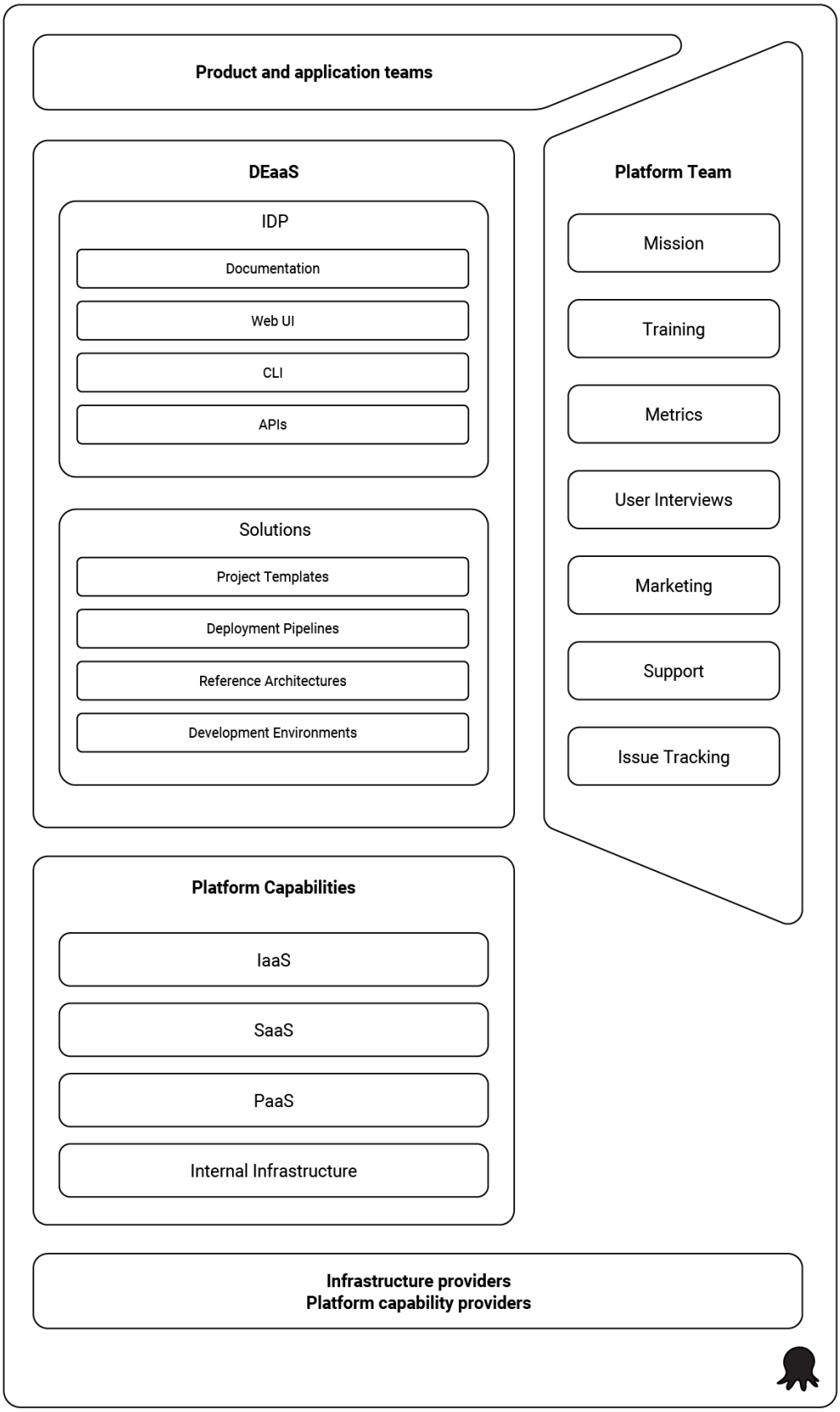
- Shared responsibility, or eventual consistency, where artifacts are shared between the DEaaS team and their customers, with any valuable modifications eventually shared between both parties
- Centralized responsibility, or enforced consistency, where the DEaaS team owns the artifacts generated by the IDP and customers have a limited ability to customize them



DEaaS teams must be mindful of the constraints they face when distributing many artifacts, allowing customers to customize those artifacts, and retaining the ability to update artifacts over time. These three concerns form the responsibility triad, and DEaaS teams can optimize for any two concerns.



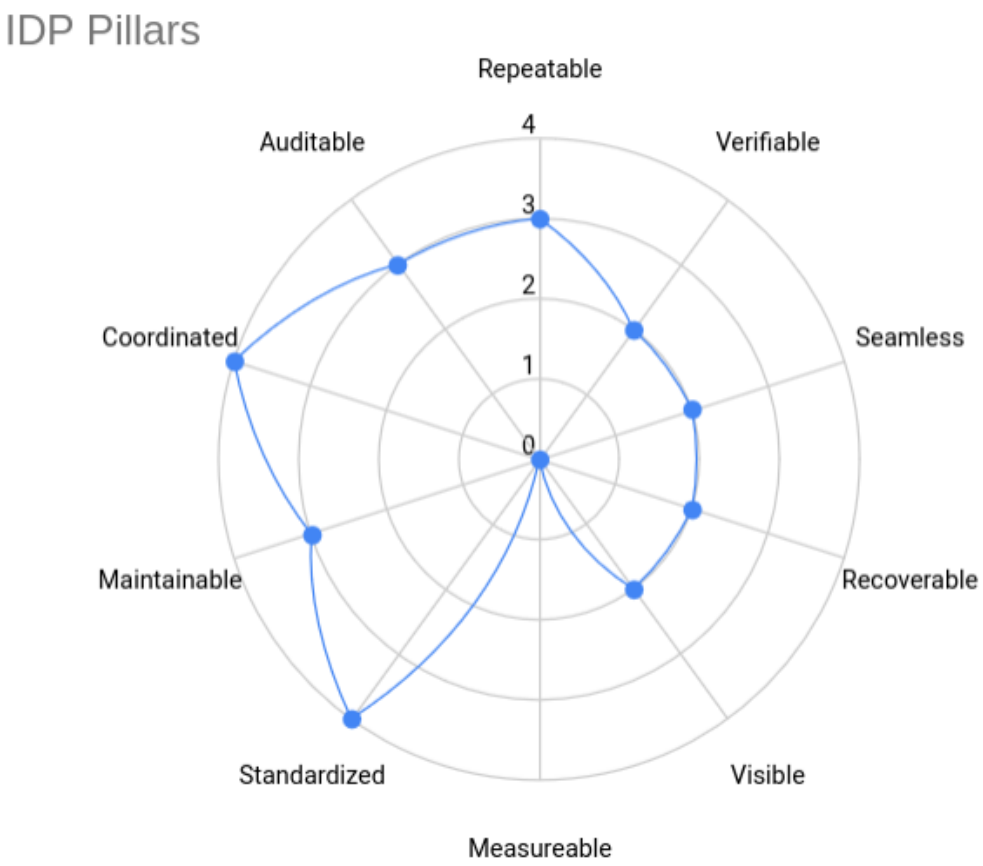
DEaaS is offered as an internal service, and like any service, requires supporting business processes such as documentation, support, issue tracking, monitoring, marketing, and training.


Product and application teams
DEaaS
IDP
Documentation
Web UI
CLI
APIs
Solutions
Project Templates
Deployment Pipelines
Reference Architectures
Development Environments
Platform Team
Mission
Training
Metrics
User Interviews
Marketing
Support
Issue Tracking
Platform Capabilities
IaaS
SaaS
PaaS
Internal Infrastructure
Infrastructure providers
Platform capability providers

Architectural decisions maintained by the DEaaS team can be broken down into functional and non-functional requirements. Functional requirements describe what a system shall do, while non-functional requirements describe how a system shall be.

A common type of artifact maintained by DEaaS teams are golden pipelines describing how software is built and deployed. 10 common non-functional requirements related to the deployment of software are:

- Repeatable deployments
- Verifiable deployments
- Seamless deployments
- Recoverable deployments
- Visible deployments
- Measurable deployments
- Auditable deployments
- Standardized deployments
- Maintainable deployments
- Coordinated deployments



In addition to measuring the impact on DevEx, DEaaS teams can also measure:

- Metrics meaningful to customers, such as the impact on DORA metrics

- Usage metrics, such as how many deployment pipelines, cloud resources, or development environments were created via the IDP
- North star metrics, such as the time it takes for a new employee to commit their 10th PR

Ultimately the value of a DEaaS team must be measured by business impact. The three areas that any business initiative can impact are:

- Revenue
- Risk
- Cost

DEaaS allows you to scale your DevOps teams by providing them with high quality and supported architectural decisions on which to build valuable solutions to unique problems. The responsibility models used by DEaaS clearly identify what DevOps teams and individuals are, and are not, responsible for. Structuring DEaaS as an internal business service ensures common business requirements such as documentation, training, support, and metrics are provisioned from the outset. DEaaS measures its impact on its customers through the improvement of DevEx, the impact on customer specific outcomes such as DORA metrics, how widespread the use of IDP artifacts are, or other north-star metrics. DEaaS also measures its business impact on revenue, risk, or cost.

Introduction

We ask a lot of DevOps teams.

The days of running software on relatively unchanging platforms are over. Cloud-native technologies and cloud providers iterate and expand their offerings at a breathtaking pace. The Cloud Native Computing Foundation (CNCF) landscape looks like the periodic table if it had over a thousand elements. Meanwhile, cloud providers host annual conventions proudly announcing dozens of new headline features. It would be a dull convention if they didn't.

New paradigms like GitOps, DevSecOps, and DevTestOps have entered our lexicon. They often have nebulous definitions but enough gravitas that they're cited and funded as part of corporate strategies.

Metrics derived from large-scale research programs like DevOps Research and Assessment (DORA) (https://oc.to/q6q1RO) measure the performance of DevOps teams. These metrics show that the hallmarks of high-performing teams are:

- Short release cycles
- Fast release cadences
- Low failure rates
- Quick recovery from failure

The shift-left mindset accepts that discovering errors after those responsible for the change have moved onto other tasks is costly. It encourages DevOps teams to identify and fix these errors earlier in their lifecycle.

DevOps teams also face a hostile environment. The Department of Homeland Security noted in its 2022 year review (https://oc.to/NtHgfi) that "the threat of cyberattacks from adverse nation states and criminal actors has only increased." These threats exploit vulnerabilities. High-profile examples, like those discovered in Log4J, left many teams scrambling to identify if they were affected and to deploy fixes, often at a high cost.

Individuals can expect a long and prosperous career gaining expertise in any one of these areas. So it's no wonder that asking DevOps teams to tackle all of these scenarios as part of their regular duties can yield unsatisfactory results.

DevOps teams understand that ad-hoc solutions to these challenges do not scale over the long term. The inability to scale proven solutions has given rise to Platform Engineering. The goal of Platform Engineering is to develop scalable solutions to improve the efficiency and well-being, or Developer Experience (DevEx), of DevOps teams.

Platform teams are responsible for creating and maintaining an Internal Developer Platform (IDP). An IDP encapsulates business goals, best practices, and hard-won practical experience in a scalable internal product. It is the interface providing DevOps teams with a self-service platform to implement trusted and opinionated solutions.

Cloud providers offer self-service low-level infrastructure such as virtual machines and networking as Infrastructure as a Service (IaaS). They also offer more specific services, like web application or container hosting as Platform as a Service (PaaS), and high-level applications as Software as a Service (SaaS). In the same way, the result of

platform teams that emphasise the improvement of DevEx through the practice of platform engineering is DevEx as a Service (DEaaS).

DEaaS defines the relationship between DevEx, DevOps, platform engineering, platform teams, and IDPs as:

- Why: Improving the DevEx of DevOps teams
- Who: Platform teams are the service provider, DevOps teams are the service consumer
- What: Platform engineering describing the work done by platform teams
- How: An IDP providing the interface between DevOps teams and platform teams

This book provides advice for the founding members of a platform team during the early stages of implementing a DEaaS strategy. It also provides practical checklists to facilitate discussions and guide decisions. Completing the sample checklists ensures that you have a clear understanding of:

- The value your platform team provides to the business
- The improvements your IDP introduce to DevOps workflows
- How you interact with and support internal customers
- The trade-offs you need to make for the long-term success of your IDP
- Common functional and non-functional requirements supporting the DevOps lifecycle

Platform Engineering versus DevOps

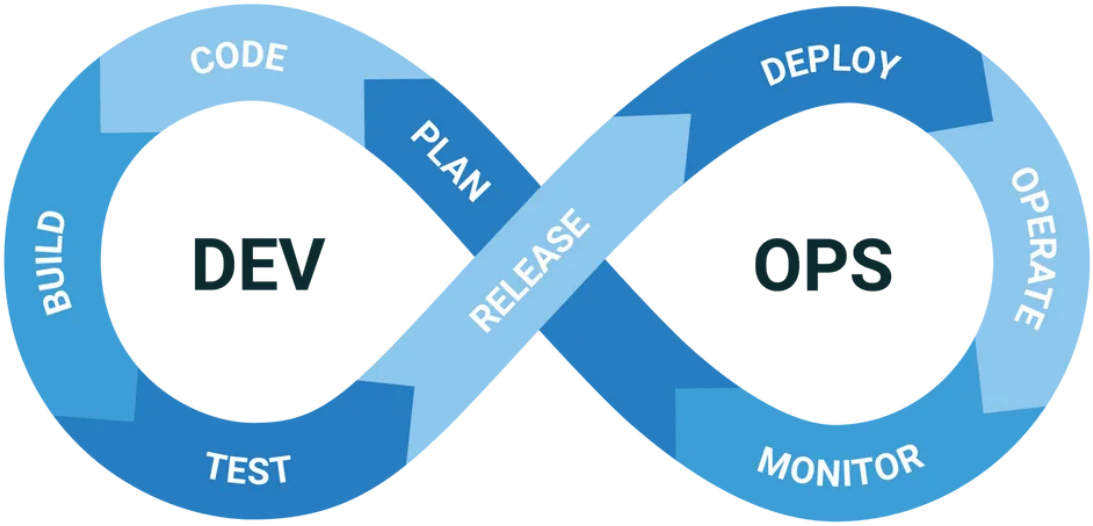
DevOps and platform engineering are similar and complementary concepts. In fact, many DevOps teams will have implemented aspects of platform engineering without necessarily thinking about it in those terms. But as DevOps teams look to provide DEaaS to help them scale, it is helpful to understand the relationship between platform engineering, DEaaS, and DevOps.

This first requires answering the seemingly innocuous question "What is DevOps?"

You can find many resources dedicated to DevOps, yet the concept can be very frustrating for those who've had to apply it meaningfully. To understand why DevOps has become so difficult to define, we'll instead seek to understand it by exploring the counterfactual "What is DevOps not?"

What is DevOps not?

The image below depicts a common interpretation of the DevOps lifecycle:



This image shows that DevOps encompasses planning, writing, testing, deploying, monitoring, and supporting software. The DevOps lifecycle is perhaps best summed up by a quote from Werner Vogels, Amazon's CTO, who said in 2006, "You build it, you run it."

We then have measurements that gauge the performance of DevOps teams. The DORA Accelerate State of DevOps report (https://oc.to/u7njwG) defines 5 key metrics:

1. Deployment frequency
2. Lead time for changes
3. Change failure rate
4. Time to restore service
5. Reliability

We referred to these as the DORA metrics. The report also links organizational performance to the characteristics described by Westrum's organizational culture (https://oc.to/r2TWX0). This is where generative cultures high in trust and low in blame exhibit higher performance.

This means:

- The DevOps lifecycle makes DevOps teams responsible for tooling and processes
- Westrum's organizational culture makes them responsible for all human interactions
- The DORA metrics makes them responsible for all measurable outcomes

And these are just 3 of the more popular perspectives on DevOps. One of the great successes of the DevOps movement is that it encourages new perspectives on the metrics, processes, ideals, and cultural underpinnings of the field.

But even with just the DevOps lifecycle, DORA metrics, and Westrum's organizational culture, we can attempt to answer the question “What is DevOps not?”

There is no satisfying answer to that question.

Without a clear internal structure in DevOps teams, the term DevOps can easily become unfalsifiable. This is because of the DevOps movement's goals. It unashamedly seeks to break down silos and absorb responsibilities that previously separated teams creating and delivering technical solutions. Simultaneously it seeks to build a culture of psychological safety to encourage creative problem-solving. The ideal DevOps team is self-sufficient, with the ability, willingness, confidence, and responsibility to tackle any and every problem.

But for every ideal DevOps team thriving with their clearly defined responsibilities and efficient processes, there's another team slowly drowning under the constant need to resolve every brittle process and incorporate every new responsibility encountered in the DevOps lifecycle. The mental burden of this undifferentiated and unsatisfying work has come to define their existence, leading to a downward spiral of dispirited and low-performing teams. Or, put more simply, teams with an unsatisfactory DevEx.

I found myself working as a developer on a codebase whose test suite would fail far more often than it passed. A few manual retries would solve the issue and ensure that pull requests could progress, although retrying the tests could take days.

The suggestion proposed by the engineering leadership at the time was that everyone should stop what they were doing and fix the tests. This is typically sound advice and very much in line with practices like Test Driven Development (TDD). Unfortunately, every developer on the team intrinsically understood that fixing the tests was a dedicated project in its own right. There were no quick wins.

This forced everyone to quietly calculate the tradeoffs between wasting days on test retries and devoting weeks to undertaking the unplanned work of fixing the tests. Needless to say, we ended up wasting a lot of time clicking that retry button.

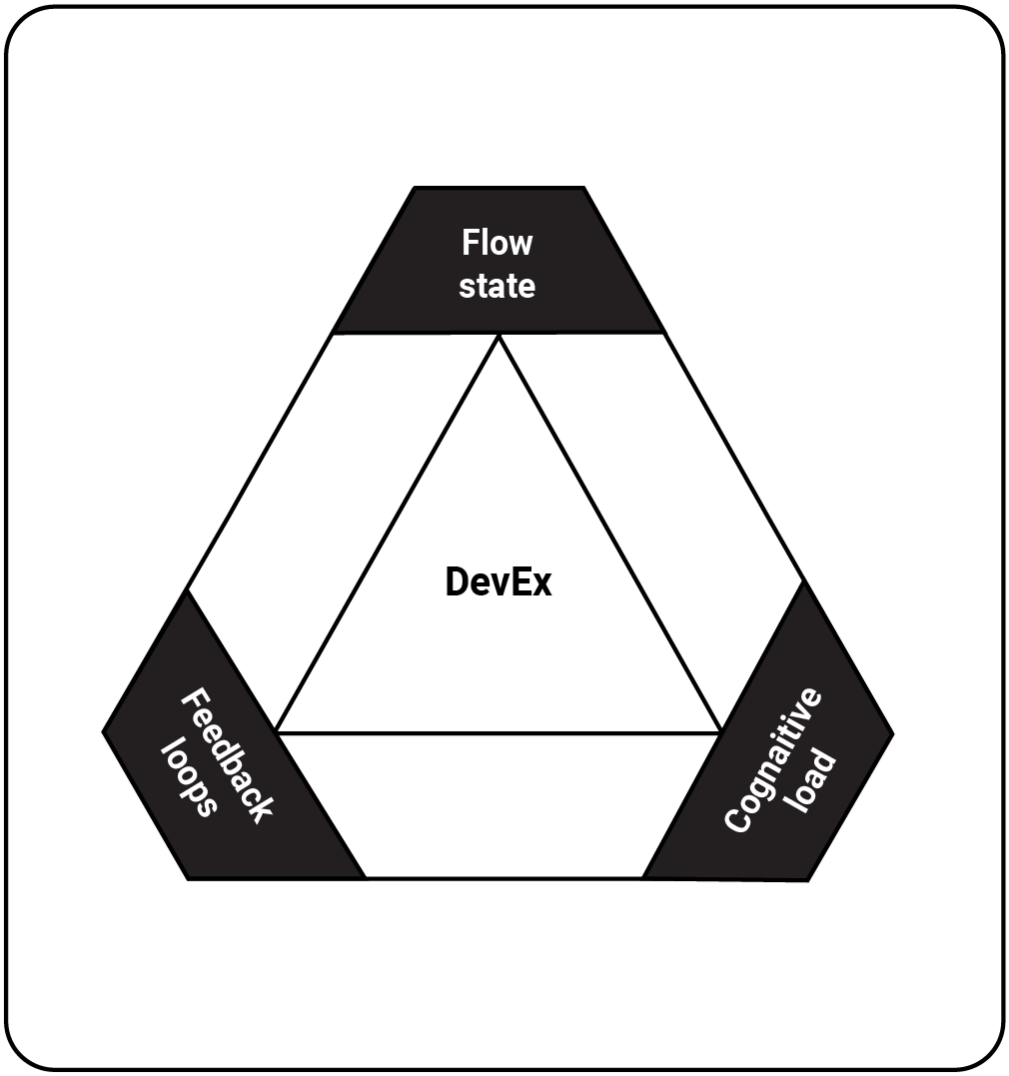
Fixing tests is arguably not the domain of platform engineering, as we'll see in the next section. But it is very much in the domain of DevOps. This example neatly highlights the dilemma faced by DevOps teams expected to take responsibility for every point of friction in their workflow.

It is not the goal of this book to provide another definition of DevOps. There are many other organizations that provide detailed opinions and research into the field of DevOps. For the purpose of this book, we'll simply rely on teams to self-identify with the roles and responsibilities associated with DevOps.

What is DevEx?

The paper titled "DevEx: What Actually Drives Productivity" (https://oc.to/KHlp1Z) provides this definition of DevEx:

> Our framework distills developer experience to its three core dimensions: feedback loops, cognitive load, and flow state. These dimensions emerged from real-world application of our prior research, which identified 25 sociotechnical factors affecting DevEx. These three key dimensions crosscut those 25 factors, providing a practical model for understanding DevEx.



Feedback loops

Software delivery involves many feedback loops:

- Build and test times
- Code review waits
- Manual testing
- Real user feedback

Each of these loops needs to be short to reduce friction. Ideally, all feedback loops should complete while the task is still active. If a task pauses during a feedback loop, it will interrupt the following work when it restarts, disrupting flow state and increasing cognitive load.

"DevEx: What Actually Drives Productivity" notes that:

> Studies have consistently shown that organizations deploying more frequently and maintaining shorter lead times are twice as likely to exceed performance goals as their competitors.

Cognitive load

The tasks associated with the phases of the DevOps lifecycle involve large amounts of mental processing. This natural cognitive load for the task increases when developers have too many tools and technologies.

You can reduce cognitive load if you:

1. Reduce toolchain friction
2. Keep documentation updated
3. Improve the system's architecture
4. Remove delays in the process

Flow state

You can probably remember when you last felt in the flow. You may have had a good chunk of uninterrupted time and could focus on the problem. The problem seemed to resolve itself and you made great progress. You enjoyed the work.

Psychologist Mihaly Csikszentmihalyi studied the concept. He described it as a fully immersed feeling of energized focus.

Flow happens naturally when you have:

1. Control over your work structure
2. Clear goals
3. Engaging work

It's not a case of making flow happen but preventing disruption from interruptions, delays, and context switching.

"DevEx: What Actually Drives Productivity" notes that:

> Frequent experiences of flow state at work lead to higher productivity, innovation, and employee development. Similarly, studies have shown that developers who enjoy their work perform better and produce higher-quality products.

You can make flow more likely by:

- Using asynchronous communication to avoid interruptions
- Clustering meetings to create large blocks of meeting-free time
- Avoiding or reducing the impact of unplanned work
- Batching questions

The impact of broken flow and high cognitive load

DevOps is all about bringing together the people needed to deliver a technical solution to customers. The DevOps lifecycle describes the various disciplines required to deliver technical solutions, but a common mistake made by some DevOps managers is to assume that DevOps means no one should specialize and everyone should be able to implement any phase in the DevOps lifecycle at any time.

There are no hard rules dictating which of the DevOps lifecycle phases any given individual will focus on. However, the reality is DevOps team members will have skills and interests that align with a subset of phases. For example:

- Developers may prefer coding, building, and testing software
- Those from an operations background may prefer building the platforms needed to monitor and operate software
- Product managers will likely focus on planning, testing, and releasing software

Allowing DevOps team members to focus on a subset of the DevOps lifecycle gives them the opportunity to enter a state of flow. It also reduces the cognitive load incurred as a result of context switching.

However, when the implementation of DevOps makes it unfalsifiable, it forces everyone to do everything all the time. This kind of hyper-generalization may be necessary, and even desirable, during the initial stages of a new DevOps team. But when it's the norm in established DevOps teams, team members have reduced opportunities to establish a state of flow. Instead, they spend their time fighting one fire after another, often implementing the fastest and easiest solution before switching focus. This means feedback on any change is given after those responsible for the change have moved on, breaking feedback loops. In addition, the DevOps landscape is now so complex that no one can be reasonably expected to manage the cognitive load incurred by frequent and unplanned jumps to random phases in the DevOps lifecycle. This denies DevOps team members the opportunity to truly engage with their work, resulting in poor DevEx.

What is Platform Engineering and DEaaS?

If DevOps encapsulates literally everything that is required to deliver a technical product to a customer, then Platform Engineering is the catalyst to initiate and facilitate DevOps processes.

More specifically, Platform Engineering is the work undertaken by a platform team to deliver DEaaS. By aligning itself with established paradigms like IaaS, PaaS, and SaaS, DEaaS expresses the outcome of platform

engineering (improved DevEx), its method of delivery (a self-service IDP), and who is responsible (the platform team) by:

- Reducing the cognitive load to participate in the DevOps lifecycle, with tested processes and proven platforms
- Increasing the opportunities to enter a flow state by enabling DevOps team members to focus on the DevOps lifecycle phases that align with their skills and interests
- Improving feedback loops by ensuring DevOps teams use consistent processes, measure their own performance with standardized monitoring, and continually optimize processes with a shift-left mentality to identify problems early

Describing Platform Engineering in terms of DEaaS also lets platform teams identify whether their work is focused on DEaaS or if it's more aligned with the kind of business as usual (BAU) work often performed by DevOps teams. This avoids the unfalsifiability of DevOps. The difference between DEaaS and other tasks is whether the work can be used as the foundation for a new project or team.

Consider that every major Operating System (OS) these days can host Virtual Machines (VMs). Does that mean you've implemented IaaS by giving every engineer a device with a modern OS? Of course not. IaaS requires provisioning a durable, accessible, supported, and monitored VM on demand, or at least with as little friction as possible.

Likewise, if spinning up a new project or team involves recreating processes and infrastructure from memory or reverse engineering settings from existing code bases and established systems, you haven't implemented DEaaS. This is true even if your work has been to improve DevEx by streamlining processes or refining the state of established platforms. This is because the knowledge about what decisions were made and why is only in the heads of a few key members of the DevOps team.

On the other hand, if you can use your IDP to spin up a new project or team, with confidence that best practices and hard-won business knowledge is baked into the foundation your IDP provides, then you have successfully implemented the core product that DEaaS provides to DevOps teams. This one-to-many relationship between an IDP and the processes used by DevOps teams shows that architectural decisions are centrally defined and you can implement them on demand.

In the talk "Compliance standards should be modern development practices" (https://oc.to/zLvWzD), Charity Majors, CTO of honeycomb.io, noted that:

> I feel like in this industry we tend to spend so much more time on individual performance than we do on fine-tuning the sociotechnical system that we operate in, even though the systems that we're operating in have so much more of an impact on how well you can do your job than your own knowledge of data structures and algorithms.

Consider a world where we had no notion of a CI server. DevOps teams still have to build and test their code. This would likely lead to situations where every developer built their code locally, nominated a "builder" with the authority to produce golden artifacts, or maybe teams go a step further and implement a rudimentary centralized solution with cron jobs.

Now imagine adding a second DevOps team. They, too, need a solution for building their code. There are 5 ways common requirements like this will be addressed:

- It won't be addressed at all
- It will be addressed with a novel solution
- It will be addressed with a coincidentally similar solution
- It will be addressed with a solution implemented from memory
- It will be addressed with a common solution

It's easy to see that common solutions to common requirements are the only sustainable path for growing DevOps teams. They reduce the scope of responsibilities that DevOps teams must actively focus on. This directly decreases cognitive load while also increasing the opportunity to enter a state of flow.

In our hypothetical scenario, the obvious solution is to deploy a CI server. But CI servers had to start somewhere. Jenkins, one of the most popular open source CI servers, was initially built by Kohsuke Kawaguchi (https://oc.to/kaK3ec):

> Kohsuke was a developer at Sun and got tired of incurring the wrath of his team every time his code broke the build. He created Jenkins as a way to perform continuous integration – that is, to test his code before he did an actual commit to the repository, to be sure all was well. Once his teammates saw what he was doing, they all wanted to use Jenkins.

Thus, a common solution to a common requirement was born. It's also no coincidence that Jenkins was born to improve Kohsuke's DevEx.

While CI servers are a solved problem (at least for most DevOps teams), environments with multiple DevOps teams working side-by-side will no doubt have many more common requirements. Addressing these common requirements with artifacts generated by your IDP is the goal of a platform team.

What is an IDP?

An IDP is the interface DevOps teams use to implement DEaaS. It may be as simple as a wiki, or as complex as a specialized platform exposing a Command Line Interface (CLI), an Application Programming Interface (API), or web interface.

In the previous section we noted that an IDP should provide common solutions to common requirements. Common solutions to common requirements are described as "architecture" in “Objects, Components, and Frameworks With UML: The Catalysis Approach” by Desmond D'Souza and Alan Wills (https://oc.to/sjdtWf) with this definition:

> The set of design decisions about any system (or smaller component) that keeps its implementors and maintainers from exercising needless creativity.

With this definition of architecture we then define the requirements of an IDP as:

- A central repository of architectural decisions made by DevOps teams
- The ability to implement architectural decisions throughout DevOps teams at scale
- Feedback processes that allow architectural decisions to be improved over time

Common artifacts implementing architectural decisions maintained by an IDP include:

- Software project templates
- CI/CD pipelines templates

- Infrastructure as Code (IaC) templates
- Scripts to configure local development environments
- Document and presentation templates
- Dashboard templates
- Checklists for starting new projects
- Reference architectures

Conclusion

DevOps is arguably the best philosophy we have to build high-performing teams to deliver technical solutions. But while DevOps can be used as the foundation for great teams, the demands placed on them must be matched by their capability. Asking DevOps teams to provide their own unique implementation to every requirement leads to inconsistent solutions at best and eventually creates an unsustainable environment. This in turn leads to a loss of flow, high cognitive load, and slow feedback cycles - all signs of poor DevEx.

This chapter describes platform engineering as the work done by platform teams to build an IDP. The IDP provides the interface through which DevOps teams implement architectural decisions. This allows DevOps teams to focus on meaningful problems rather than being distracted by undifferentiated work. This in turn improves their DevEx, which is the goal of DEaaS.

Functional and Non-Functional Requirements

We now understand the purpose of an IDP is to allow architectural decisions to be implemented at scale throughout DevOps teams. The challenge is to determine the set of architectural decisions that most improve the DevEx of DevOps teams.

Platform teams must articulate architectural decisions to two very different audiences: the business and DevOps teams.

The business is interested in a high level description of a problem and its solution as well as a quantifiable measurement of the solution's value. DevOps teams are more interested in a technical description of a problem and the solution's implementation details.

Describing an IDPs functionality in terms of functional and non-functional requirements allows platform teams to speak to both audiences. Non-functional requirements describe what a system will do for the business. Functional requirements describe how the system is implemented for the DevOps teams.

This distinction is captured by Wikipedia which provides this definition (https://oc.to/PFMuXO):

> Broadly, functional requirements define what a system is supposed to do and non-functional requirements define how a system is supposed to be.

Wikipedia then goes on to list dozens of examples of non-functional requirements. Given the tight relationship between platform and DevOps teams, the DORA Capability catalog (https://oc.to/GXvCYS) may be more relevant as it provides a collection of non-functional requirements specifically tailored to DevOps teams. The chapter "Platform Engineering Non-Functional Requirements" provides a list of ten common non-functional requirements, or pillars, relating to software deployment and operations processes.

Functional requirements are tightly coupled to an IDP and the systems it integrates with. While we won't be going into the details of functional requirements in this book, the chapter "Platform Engineering responsibility models" describes the 3 common patterns that architectural decisions must adopt when they are implemented at scale in DevOps teams.

Platform Engineering Non-Functional Requirements

The shift-left mentality is a valuable guide when determining which steps in the DevOps lifecycle to codify and scale with your IDP. Identifying problems late in the lifecycle and implementing common strategies to resolve them earlier reduces cost and increases velocity. Or, following the motto of Extreme Programming:

> If it hurts, do it more often.

Exact numbers around the value of finding bugs earlier are hard to come by. The post “Everyone cites that 'bugs are 100x more expensive to fix in production' research, but the study might not even exist” (https://oc.to/rLmuBw) references research from Hillel Wayne that questions the validity of this commonly cited metric. However, the byline of that article concedes, "It's probably still true, though, says formal methods expert."

While it doesn’t provide meme-worthy quotes, the report “The Economic Impacts of Inadequate Infrastructure for Software Testing” (https://oc.to/Tq1Zlh), prepared for the National Institute of Standards and Technology (NIST), performed a case study in the financial services sector to estimate the economic impact of inadequate infrastructure for software testing, finding:

> The major benefits developers cited from an improved infrastructure were direct cost reduction in the development process and a decrease in post-purchase customer support. An additional benefit that respondents thought would emerge from an improved testing infrastructure is increased confidence in the quality of the product they produce and ship.

Identifying issues early in the DevOps lifecycle means identifying them while planning new features, writing and testing code, or as part of your Continuous Integration and Delivery (CI/CD) pipeline. The remainder of the DevOps lifecycle takes place after you deploy code to production. These are the lifecycle stages that issue identification and rectification are ideally moved out of.

Your CI/CD pipeline is a desirable target for your IDP because this process should be mostly automated, so DevOps teams can apply improvements at scale. But, more importantly, your existing CI and CD platforms have likely already solved many of the requirements that enable automation:

- They provide an execution environment (in the form of agents or workers) to run automated scripts and applications
- They have an established process for maintaining authentication and authorization
- They have existing access to your critical infrastructure
- Existing operations teams support them
- Their pipelines are easily modified to include new steps, or triggers provide hooks into existing workflows
- DevOps teams are already familiar with them

As we'll see in the chapter "Planning your Internal Developer Platform", clearly defining your IDP's mission ensures the platform team is focused on a specific, achievable outcome. This mission statement has 2 parts:

- Non-functional requirements providing a high-level description of the outcome
- Functional requirements that describe how the feature works

While functional requirements align with DevOps teams' specific processes and goals, non-functional requirements represent common high-level scenarios that can be related back to wider business goals.

This chapter describes 10 non-functional requirements, or pillars, related to CI/CD pipelines. These pillars represent desirable traits to prompt discussions with customers or standardize across your customer base.

Pillar 1. Repeatable deployments

Software teams are deploying software to production more frequently than ever. They also make deployments to pre-production environments as part of their deployment pipeline.

To be confident when you change software at high velocity, you need a mix of methods to confirm the release-ability of your software. These methods include test automation, exploratory testing, self-beta testing (drinking your own champagne), and other techniques.

These techniques are only helpful if what you deploy to production is the same as what you deploy to your other environments.

The pillar of repeatable deployments requires you to deploy the same thing in the same way each time you deploy an application version.

General deployment concepts

To understand repeatable deployments, we need to fine-tune our definitions. We must be precise with some terms relating to deployment pipelines and the timing of different stages.

Deployment pipelines

A deployment pipeline starts when you commit code to a source code repository. And it follows the change all the way to the production environment.

Every activity to progress the change is part of your deployment pipeline. Whether manual or automated, this includes:

- Code reviews
- Builds
- Testing
- Release management
- Sign-offs
- Deployments

Continuous Integration

The most common entry point to a deployment pipeline is Continuous Integration (CI). This is the process of committing code, compiling it, running tests, packaging a new application version, and publishing it.

Dave Farley and Jez Humble recommend changes are regularly integrated into the main branch in source control (hence the "continuous" in "Continuous Integration"). Many teams use the term more loosely to define their

automated build process.

Continuous Delivery

While Continuous Integration is the automation of creating a deployable package each time the code changes, Continuous Delivery (CD) extends this through to deployment automation and monitoring.

You might still have some manual stages in your deployment pipeline, such as exploratory testing or an approval process. But you should automate all deployment steps as this helps you achieve repeatable deployments.

Using the same deployment process for all environments means it gets tested as often as the application.

Continuous Deployment

Continuous Deployment takes CI/CD a step further. It removes all manual intervention to create a fully automated commit-to-consumer workflow. The deployment pipeline automatically rejects a bad application version and deploys all good versions to production without manual approvals.

CI/CD

CI/CD refers to the combination of Continuous Integration with Continuous Delivery or Continuous Deployment.

Teams commonly deploy to the development environment without manual intervention but let people control when they deploy to subsequent environments. We've learned from most of our customers that this approach *works well for them.*

In the test environment, quality assurance (QA) staff validate changes, product owners review new functionality, security teams probe for vulnerabilities, etc. When everyone is happy that the changes meet their requirements, they can promote the application version to production.

The production environment is the final destination. It's where end users can use the application.

What is an environment?

Environments represent the boundaries between copies of individual applications or entire application stacks and their supporting infrastructure.

Each environment should reasonably reflect the production environment. This ensures the application will behave consistently and avoids surprises when you go live.

You might frequently deploy to earlier environments, trading stability for faster feedback. Your users expect production environments to be stable.

You progress deployments through environments to increase confidence that you can deliver a working solution to the end user.

We call the canonical set of environments development, test, and production. The table below describes the characteristics of these environments:

Environment	Description	Deployment Frequency	Stability / Confidence
Development	Used by developers to test individual changes as they're implemented.	High	Low
Test	Used by developers, QA, and non-technical staff to validate that changes meet requirements.	Medium	Medium
Production	Accessed by end users to consume the publicly available instance of the applications.	Low	High

Although you can have any number of environments with different names, we use this set of environments in this book.

What is a deployment?

We talked about deploying "applications" to environments, but to appreciate how you achieve repeatable deployments, we must be more specific about what we deploy.

A deployment should include a snapshot of:

1. The application version
2. The deployment process
3. The variables used to configure the application for an environment
4. Inline scripts that support the deployment and configuration of the application and infrastructure

These are combined into a *release*, which captures the current deployment state. The release snapshot ensures you use the same snapshot for the release, even if you change the process, variables, or scripts during environment progression.

Without a release snapshot, you could deploy a tested application version using an untested deployment process, resulting in a problem in your production environment.

Pillar 2. Verifiable deployments

The repeatable deployments pillar describes how promoting releases through environments increases confidence in the application version and deployment process.

The pillar of verifiable deployments describes the techniques you can use to verify a deployment when it reaches a new environment.

General testing concepts

Testing is a nebulous term with often ill-defined subcategories. We won't attempt to provide authoritative definitions of testing categories here. We aim to offer a high-level description of common testing practices and highlight those you can use when you deploy.

What don't we test during deployments?

We consider unit tests part of the build pipeline. They're closely tied to the code and must pass for the build server to publish the application package.

You might also run integration tests during the build process to verify components interact as expected. Sometimes, you might use a test double instead of a real component to improve reliability. Otherwise, you might spin up the dependencies as part of the test.

The tests run by the build server should result in you publishing a high-quality application package. Bad application versions should get prevented from progressing.

What can we test during deployment?

Tests that need a live application or application stack to be accessible are ideal candidates to run as part of a deployment process.

Smoke tests are quick tests designed to ensure you deploy applications and services correctly. Smoke tests implement the minimum interaction to ensure services respond correctly. Some examples include:

- An HTTP request of a web application or service to check for a successful response.
- A database login to ensure the database is available.
- Checking that a directory has been populated with files.
- Querying the infrastructure layer to ensure the expected resources were created.

Integration tests can run as part of your build or deployment processes. Integration tests validate that multiple components are interacting as you expect. You may include test doubles with the deployment to stand in for dependencies, or the tests may verify 2 or more running component instances.

Examples include:

- Signing into a web application to verify that it can interact with an authentication provider.
- Querying an API for results from a database to ensure that the database is accessible via a service.

End-to-end tests provide an automated way of interacting with a system like a user would. These can be long-running tests following paths through the application that require most or all application stack components to work correctly. Examples include:

- Automating the interaction with an online store to browse a catalog, view an item, add it to a cart, complete the checkout, and review the account order history.
- Completing a series of API calls to a weather service to find a city's latitude and longitude, getting the current weather for the returned location, and returning the forecast for the rest of the week.

Chaos testing involves deliberately removing or interfering with the components that make up an application. This validates that the system is resilient enough to withstand such failures. You can combine chaos testing with other tests to verify the stability of a degraded system.

Usability and acceptance testing often involve a human using the application to verify that it meets the requirements. The requirements can be subjective, for example, determining if the application is visually appealing. Or the testers may be non-technical and don't have the option of automating tests. The manual and subjective

nature of these tests makes them difficult, if not impossible, to automate. This means you must deploy a working copy of the application or application stack and make it accessible to testers.

Pillar 3. Seamless deployments

When you deploy your application, you need to switch users from the old application version to the new one. The pillar of seamless deployments discusses approaches to reducing the user impact when you update the application.

One of the easiest ways to deploy an application without impacting users is to use planned maintenance windows. You can schedule these to occur outside of the users' regular working hours to minimize disruption to their work.

This approach is less practical if you have users in many time zones, need to minimize downtime, or want to deploy more often.

In these cases, you can use some common deployment strategies to deploy new application versions seamlessly.

Seamless database deployments

No discussion on seamless deployments can begin without first addressing the issue of database updates.

A fundamental aspect of most seamless deployment strategies involves running 2 versions of your application side-by-side, if only for a short time. If 2 versions of the application access the same database, all updates to the database schema and data must be compatible with both versions. We refer to this as backward and forward compatibility.

However, backward and forward compatibility is not trivial to implement. In the presentation Update your Database Schema with Zero Downtime Migrations (https://oc.to/BzKUy3), based on chapter 3 of the book Migrating to Microservice Databases (https://oc.to/nLvrV4), Edison Yanaga walks through the process of renaming a single column in a database. It involves 6 incremental updates to the database and application code, and you must deploy each in sequence.

Seamless deployments involving a database require:

- Careful planning
- Many small steps to roll out the changes
- Tight coordination between the database and application code

You can find more database techniques in Refactoring Databases (https://oc.to/huM7G3) by Scott W. Ambler and Pramod Saalage.

Deployment strategies

There are several ways to manage a cutover between an existing and a new application version. The right deployment strategy can help you achieve seamless deployments.

Recreate

The recreate strategy doesn't provide a seamless deployment. It's included here as the default option for most deployment processes. This strategy involves either:

- Removing the existing deployment before deploying the new version
- Deploying the new version over the top of the current one

Both options result in downtime between the existing version being stopped or removed and the new version starting. However, because you don't run the current and new versions concurrently, your database upgrade won't need to satisfy backward and forward compatibility requirements.

Rolling updates

The rolling update strategy involves incrementally updating instances of the current deployment with the new deployment. This strategy ensures at least one application instance is always available during the rollout.

Your load balancer will send users to instances with its usual balancing pattern. As the rollout progresses, more users will move onto the new application version. You can also take each instance out of balance before you update it, so requests aren't dropped during the deployment.

With rolling updates, your database will need to maintain backward and forward compatibility, as during the rollout it will receive connections from both the old and new application versions.

Canary deployments

The canary deployment strategy is similar to the rolling update strategy. Both incrementally expose more end users to the new deployment over time.

With canary deployments, you create a small group of users who get new versions before everyone else. This group may contain users who opt-in to get early access, represent a segment of your business (such as free-tier users), or be an algorithmic sample.

The first step of a canary deployment is to update the application instance used by your canary users. After you have collected information from this group, you can decide whether to continue the rollout. This may be a human decision, or you might automate it based on monitoring data or log files.

Canary deployments let you halt the rollout and revert to the previous application version if you find a problem in the sample group.

Blue/green deployments

The blue/green strategy involves deploying the new version (the green version) alongside the current version (the blue version) without exposing the green version to any traffic. After you deploy and verify the green version, traffic is cutover from the blue to the green version. When the green version handles all traffic, you can remove the blue version.

Any database changes deployed by the green version must maintain backward and forward compatibility. Even when the green version is not serving traffic, the blue version gets exposed to database changes.

This deployment strategy requires additional infrastructure during the deployment process. But you can create and destroy this infrastructure automatically, or use it as a cold standby, depending on your needs and cost constraints.

Session-draining

You can use the session-draining strategy when applications maintain states tied to a particular application version.

This strategy is similar to the blue/green strategy as it also means you deploy the new version alongside the current version, running both side-by-side. Unlike the blue/green strategy, session-draining will direct only new sessions to the new version while the existing one continues to serve traffic for existing sessions.

You need the same infrastructure and clean-up for blue/green and session-draining strategies.

Any database changes must maintain backward and forward compatibility because the old and new application versions run side-by-side.

Feature flags

The feature flag strategy involves building functionality into a new application version and using a toggle or feature flag to control its visibility. This lets you control feature visibility without needing a deployment.

In practice, you deploy a new application version with flaggable features with one of the strategies above, so the feature flag strategy complements those other strategies.

Feature branch

The feature branch strategy lets developers deploy an application version with changes they're working on. It's usually deployed in a non-production environment alongside the main deployment.

Maintaining database backward and forward compatibility may not be necessary with feature branch deployments. Because feature branches are short-lived for testing, it's acceptable that each deployment has access to an isolated test database.

Pillar 4. Recoverable deployments

Despite your best efforts with repeatable and verifiable deployments, you'll always need to handle production bugs. When this happens, quickly and safely recovering is crucial.

Rolling back or forward

Recovering from an undesirable deployment means choosing whether to:

- Roll back to a previous good application version
- Roll forward to a new version that returns the environment to a desirable state

Either solution works with stateless applications, but you must treat rollbacks with care when there's a database involved.

This is the advice from the FlyWay project (https://oc.to/lVr2Zn):

> While the idea of undo migrations is nice, unfortunately it sometimes breaks down in practice. As soon as you have destructive changes (drop, delete, truncate, …), you start getting into trouble. And even if you don't, you end up creating home-made alternatives for restoring backups, which need to be properly tested as well.

Redgate offers this advice for database rollbacks (https://oc.to/sbc7WW):

> Rather than investing time and energy into rollback planning, an alternative is to follow an approach that keeps you moving forward.

The blog post “Pitfalls with SQL rollbacks and automated database deployments” (https://oc.to/WRvLE4) has this advice:

> More often than not, the effort to successfully rollback a deployment far exceeds the effort it would take to push a fix to production.

When deployments involve database changes, we recommend rolling forward to recover from an undesirable deployment.

Rolling back

If you achieved the pillar of *repeatable deployments*, you can roll back by re-running a previous deployment. This is possible because the package versions, scripts, and variables are all captured by a repeatable deployment.

Rollbacks are also an explicit feature of several seamless deployment strategies:

- Canary deployments implement rollbacks by redirecting all traffic from the new deployment to the current deployment.
- Blue/green deployments can roll back a deployment by cutting traffic back to the blue stack.
- Session-draining deployments can redirect new sessions to the current deployment and optionally kill any sessions in the new deployment.

Rollbacks have the following benefits:

- You can fix a deployment issue, without writing code, by rolling back to a previous deployment.
- A rollback leaves the system in a known, verified state.
- You can measure the time to complete a rollback in non-production environments.

Rollbacks have the following disadvantages:

- Rollbacks are all-or-nothing operations. You can't roll back individual features, only entire deployments.
- You need to test rollbacks as part of the deployment process to ensure they work as expected. This increases the complexity and time of the deployment process.
- If a rollback fails, it's likely you'll need to resolve the issue by rolling forward.
- Database rollbacks require special consideration to ensure data is not lost.

Rolling forward

Rolling forward is another way to describe performing a new deployment. In this case, the new deployment will only contain the fixes required to restore an environment.

Rolling forward has the following benefits:

- All deployment strategies, with or without a database, inherently support rolling forward.
- Teams gain experience in rolling forward with every deployment.
- You can choose the scope of a change or fix when rolling forward.
- You can deploy multiple times in succession while rolling forward to resolve an undesirable deployment.

Rolling forward has the following disadvantages:

- Rolling forward typically requires a developer to implement a fix to include in the next deployment.
- You must have a short lead time for changes. Otherwise, it's tempting to skip environments to expedite the fix.
- The production environment will be left in an undesirable state for as long as it takes to develop and deploy the next version.

Pillar 5. Visible deployments

It can be challenging to track which application version you deployed to each environment. You shouldn't have to review the files on the disk or the structure and data in the database. This is like trying to work out what mix of colors produced a tin of paint.

Having a view of environments and application versions is crucial to understanding:

- What features you provided to your customers
- What features are being tested
- What issues you fixed
- The history of any changes

Listed below are the details required to gain complete visibility into the state of your deployments.

Commit messages

Commit messages capture the intention of source code edits, describing the changes made and who made them. These messages are invaluable when trying to understand at a low level what changes made it into a particular version of a package.

Issue tracking

Often, you make source code commits to resolve an issue documented in a dedicated issue tracker. These issues provide a space for you to describe, discuss, and track bugs. A unique identifier references each issue.

Capturing the issue IDs related to changes in a package version and any deployment that includes that package version provides insight into the issues resolved in any deployment.

Build logs

A typical CI/CD pipeline will have a build server that builds, tests, and packages an application. The log files for these builds contain a wealth of information, such as:

- Tests passed
- Tests the team ignored
- Dependencies included
- Packages created

You can quickly review these log files if you have a link to the build information from the deployment.

Library dependencies

Almost every application deployed today combines custom code with third-party, often open-source, libraries. These external libraries provide useful features but can be a source of bugs or security vulnerabilities. Understanding all the application's dependencies is essential for security, auditing, and debugging.

Legislation such as the Cyber Resilience Act (CRA) and the requirements of the NIST “Software Supply Chain Security Guidance Under Executive Order (EO) 14028” (https://oc.to/T3b11u) may compel you to produce Software Bill of Materials (SBOMs) to accompany your application, making the management of library dependencies a requirement of your CI/CD pipeline.

Deployment versions

A release version captures a snapshot of the above information, along with the deployment process, package versions, variable values, and scripts. You deploy this release version to each environment.

Displaying which release versions you deployed to each environment provides a high-level view of the state of your deployments. With this information, anyone can see what's deployed where. By drilling into the details of a release, you can see the commit messages, issues, dependencies, and links to the CI builds.

Pillar 6. Measurable deployments

The deployment pipeline's primary goal is getting your software into your customers' hands. To understand how well your deployment pipeline is performing, you need to measure the metrics that define success for you.

You first need to define useful metrics, then reliably collect them, and surface them in an easy-to-understand format.

Some metrics you can use to track deployments are:

- Deployment frequency: How frequently do you deploy to production?
- Lead time for changes: How long does it take for a code change to be deployed to production?
- Time to recover deployment: How long does it take to recover from a failed deployment?
- Deployment fail rate: What is the ratio of failed to successful deployments?
- Change fail rate: What is the ratio between hotfix and regular deployments?
- Deployment duration: How long does each deployment take?

Pillar 7. Auditable deployments

If the visible deployments pillar is about surfacing the current state of your environments and the changes made as part of the release, then auditing is about tracking people's involvement in the deployment process over time.

Auditing lets teams see a history of all deployment activity, like:

- Deployments to environments
- Changes to the deployment process
- Changes to environments
- Who approved a deployment
- The state of an environment at some point in the past

For audit events to be helpful, they must be searchable, filterable, and exportable to support reporting and analysis.

Pillar 8. Standardized deployments

Just as repeatable deployments build confidence as you promote a release across environments, standardizing deployment processes across different projects allows teams to confidently use proven solutions.

Your IDP is an ideal solution for standardizing deployment processes across DevOps teams. Much of what we have discussed in this book directly applies to planning, creating, and maintaining standardized deployment processes.

As noted in the chapter introduction, CI/CD pipelines are a logical place to embed the automated and scalable functionality that platform teams are responsible for delivering. In this case, your IDP is responsible for providing the pillar of standardized deployment pipelines, and the pipelines will then embed other appropriate pillars.

Pillar 9. Maintainable deployments

Getting your deployments to the production environment is just the beginning. There are day-to-day operations tasks that keep your applications running and your customers happy. These include:

- Diagnosing issues
- Collecting logs
- Performing backups
- Restarting services
- Rotating keys
- Testing connections

While you could SSH or RDP into a server and start poking around, each change causes your environment configuration to drift, making it harder to implement repeatable deployments. It's also difficult to track changes, verify that they worked, and audit who changed what.

Just like deployments, maintenance tasks should be:

- Repeatable
- Verifiable
- Visible
- Measurable
- Auditable

- Standardized
- Coordinated

Maintenance tasks represent the business knowledge needed to keep your deployments running and should meet the same standards as your deployment processes.

Pillar 10. Coordinated deployments

Deploying a package to an environment is one small part of the deployment process. Often, you need to coordinate deployments with other business processes to ensure:

- The right people have given their approval
- Interested parties get notified of the success or failure of a deployment
- Deployments proceed in the correct order
- Deployments can only occur during specific times
- High-priority deployments take precedence over low-priority ones
- Deployments get scheduled to take place at a predetermined time
- External events can trigger deployments
- Deployments can trigger external events

A deployment process may be a single component in the broader ecosystem of business process management tools. Orchestrating deployments from third-party platforms and reporting results back lets teams manage complex deployments as part of a broader business process.

Example checklist: Capturing functional and non-functional requirements

This example checklist provides a table for the platform team to note how to implement the pillars with functional requirements.

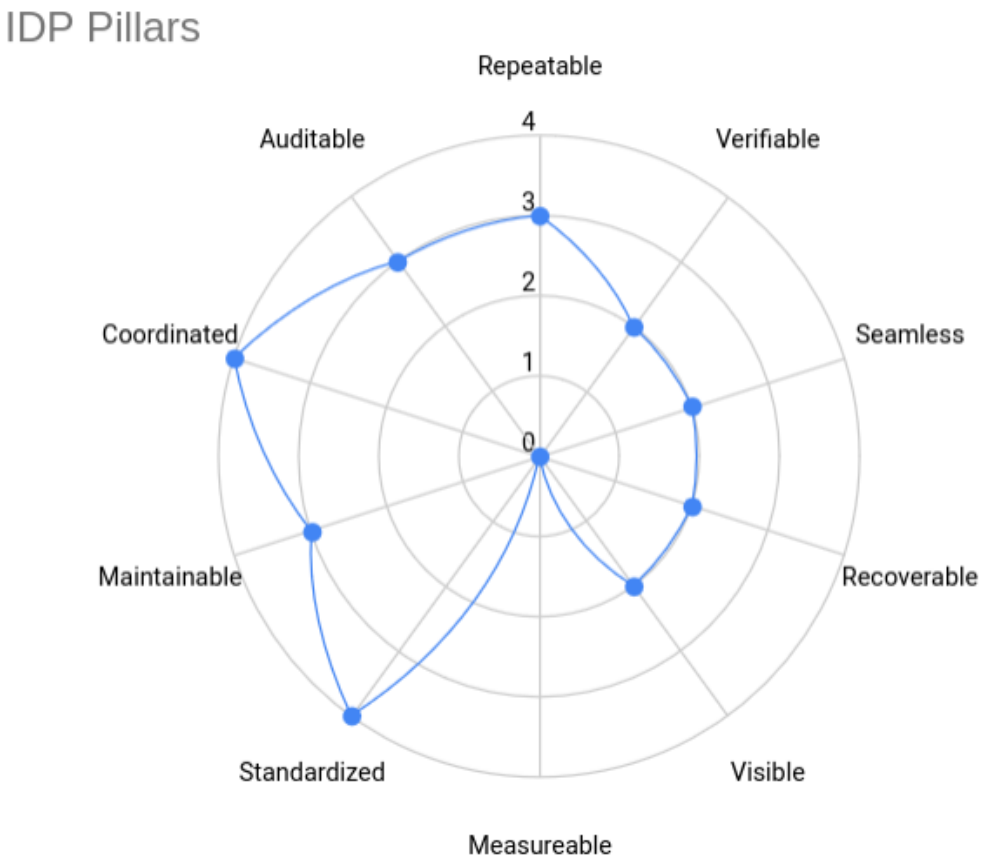
The values assigned to the pillars are subjective but capture their relative importance.

You then plot these values on a spider (or radar) chart. This visualization identifies both essential and deprioritized pillars.

The decision not to implement a feature may be as important as the decision to implement one. The combination of the checklist and spider chart surfaces both decision types.

Pillar	Importance	Functional requirements
Repeatable	3	Need to allow all engineering leads to promote applications to production with a single click.
Verifiable	2	Browser-based tests must be run in the staging environment and optionally in production as a debugging tool.
Seamless	2	Blue/green deployments in production.
Recoverable	2	Rollbacks are performed by reverting traffic to the blue stack.
Visible	2	Need to quickly find production versions of applications.

Measurable	0	Not a priority.
Standardized	4	Need golden paths for Kubernetes, Lambda, and VM deployments.
Maintainable	3	One-off scripts for common DB operations like regenerating indexes. Smoke test script to hit API, web page, and database login.
Coordinated	4	Production deployments are scheduled after hours. Email report of deployment success or failure.
Auditable	3	Must track changes to the deployment process and the deployments to support ISO 27001.



Conclusion

Many IT departments have fallen into the trap of being order-takers, dutifully asking how high when asked to jump. As a platform team member, you have a unique opportunity to move beyond order taking. You can lead discussions regarding best practices and the future vision of your DevOps teams.

The 10 pillars in this chapter provide a set of non-functional requirements relating to CI/CD. These pillars are helpful in discussions with your customers and inspire them as you move towards a more holistic approach to software delivery.

The sample checklist reinforces this holistic approach by presenting the pillars as a radar chart, clearly identifying which pillars you're considering and which you're not. Capturing all the choices that influence your processes helps to keep your focus on your mission as a platform team.

Platform Engineering responsibility models

A primary goal of any platform team is to never block internal customers from achieving their goals while providing solutions that remove the need to reimplement solved problems.

This quote from Peter Gillard-Moss of Thougtworks in the blog post “Platform Tech Strategy: The Three Layers“ (https://oc.to/Ka3d7I) sums it up nicely:

> Platforms are a means of centralizing expertise while decentralizing innovation to the customer or user.

Achieving this balance requires deliberate decisions about who has responsibility for artifacts generated by your IDP. This chapter explores the 3 common ways in which responsibility is assumed by, or shared with, the platform team and your customers:

- Customer responsibility, where customers own IDP artifacts once they're generated
- Shared responsibility, where the platform team and customers collaborate on IDP artifacts over time
- Centralized responsibility, where IDP artifacts are mostly read-only

Customer responsibility

The easiest way to think about customers taking responsibility for the artifacts generated by your IDP is to think of copying a template document from a read-only shared drive into the customer’s home directory.

You must make every attempt to ensure the template is fit for purpose and of high quality. But, after a customer makes a copy, they assume responsibility for using, editing, and updating that copy. Existing copies do not receive updates to the source template document.

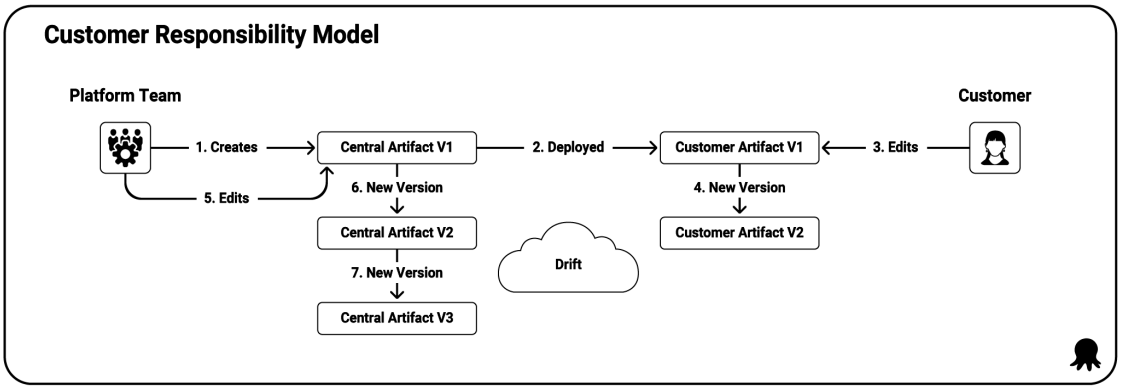
In the same way, your IDP does not provide updates for artifacts like sample projects or scripts, deployment pipelines, or example dashboards. The platform team only supports them in a limited fashion once a customer has taken ownership under a customer responsibility model.

This model makes it less likely the platform team will block their customers. If the artifacts are insufficient or require change, customers have the freedom and responsibility to do it themselves.

The downside to this model is that it doesn't allow for automatic updates to artifacts generated by the IDP. The disconnected nature of artifacts distributed under the customer responsibility model limits the ability of the platform team to propagate improvements.

Heavily modifiable artifacts are well suited to this model. For example, a "hello world" style software project template that embeds standard settings, like dependency repositories, common build scripts, and a skeletal folder structure, is an ideal candidate for the customer responsibility model, as customers will drastically alter this project template almost immediately.

You can think of the customer responsibility model as an "eventual inconsistency" model where teams start with the same foundation but diverge as they customize their solution.



Shared responsibility

The shared responsibility model lets the customer and the platform team modify an IDP artifact after creation. An example of this model is the process of forking a Git repository. Because both Git repositories share the same history, you can merge changes between them while keeping any local changes.

This model may involve the platform team pushing changes, the customer pulling changes, or both.

Git is an excellent example of the shared responsibility model and one of the most practical ways to implement it. The merging strategies implemented by Git do an excellent job of tracking changes (as opposed to simply finding differences between files). By having your IDP generate Git repositories as artifacts, it's possible to merge the downstream (or customer controlled) and upstream (or IDP controlled) repositories.

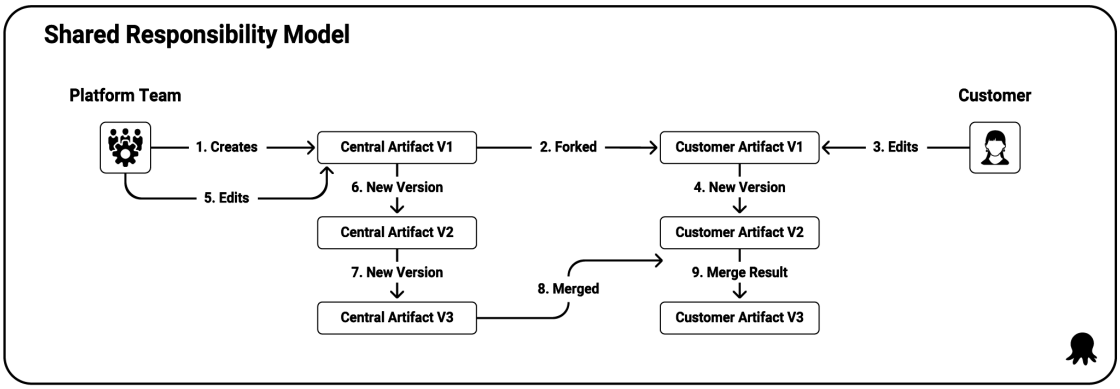
Git is not a requirement to implement the shared repository model, though. A regular diff between two plain text files may be enough for customers and the platform team to consolidate changes made by each other. As a fallback, copy and paste may suffice to synchronize changes.

Like the customer responsibility model, the shared responsibility model helps prevent the platform team from being a blocker. Customers have the ability and responsibility to update their artifacts to resolve any issues.

The shared responsibility model can propagate changes from artifacts maintained by the IDP to those maintained by the customer.

However, be aware that it may not be reasonable, or even possible, to propagate changes to heavily modified customer artifacts. In these situations, you need an agreement on who does the work of merging changes. Customers may be happy to pull in changes as needed, or the platform team may need to do the work to push changes out.

The shared responsibility model represents an "eventual consistency" model where the platform team and their customers work in parallel and eventually merge any useful changes.



Centralized responsibility

The centralized responsibility model gives customers read-only or locked-down copies of any generated artifacts. While customers are free to use these artifacts, they have little or no ability to customize them.

An analogy is a virus scanner deployed to your workstation by the security team. You can run the virus scanner, but you can't configure it. A central team manages any updates to the virus scanner or its configuration.

Likewise, customers can use a deployment pipeline distributed under the centralized responsibility model to deploy their software, but they can't edit the pipeline.

The centralized responsibility model lets the platform team push out changes quickly. Because the artifacts are always in a known state or the platform team can overwrite any changes, it becomes much easier to update artifacts at scale. This model is easy to implement with common Infrastructure as Code (IaC) tools like Terraform, CloudFormation, and Azure ARM Templates.

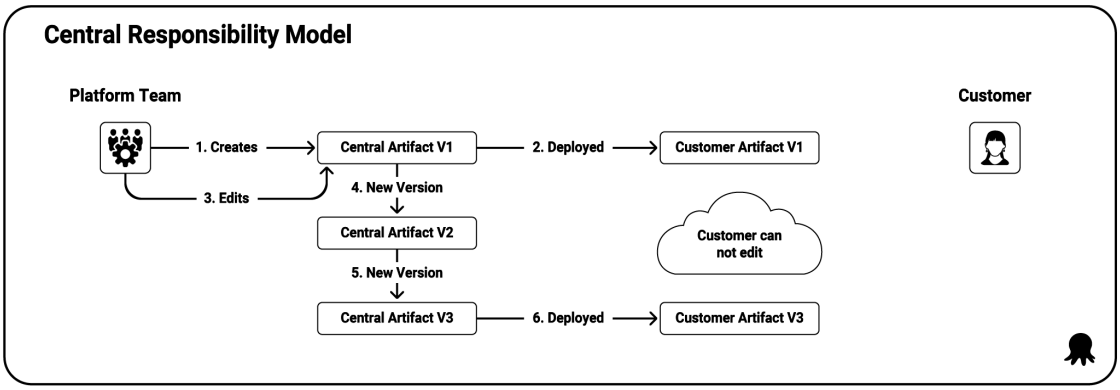
The downside to this model is that you risk blocking your customers. If the artifacts you provide have faults or limitations, it's entirely up to you as a platform team member to push out updates to unblock your customers.

This model suits simple and extensively tested artifacts, such as scripts to perform backups or restart services. Unfortunately, this model tends to fail when distributing more complex artifacts. As Gregor Hohpe put it in his presentation "The Magic of Platforms" (https://oc.to/bWjWFK):

> There are two ways to build a platform:
>
> 1. Be smarter than everyone else and anticipate all their needs
> 2. Evolve the platform based on user needs, which can be sensed from platform usage
>
> One is more likely than the other.

As your IDP gains traction, your customer's needs will test the bounds of complex artifacts distributed under the centralized responsibility model.

The centralized responsibility model is an "enforced consistency" model where the platform team retains most of the control of any artifacts generated by the IDP.



Customization as feedback

In the section "What is DevEx?", we noted that feedback loops are a core dimension of DevEx.

Metrics like NPS are appealing because they can be automated and asynchronous but may be of limited value with small sample sizes. And while scheduling feedback sessions with your customers is relatively easy, since your customers are typically also your colleagues, this process is time-consuming.

Another feedback option is to observe the changes made by customers to the artifacts generated by your IDP. This asynchronous feedback provides high-quality insights into how your customers want to use your platform.

The idea is similar to desire paths. The ABC News article “What desire paths can tell us about how to design safer, better public spaces” (https://oc.to/tVhtZ4) provides this definition:

> A desire line, or desire path, is an unplanned trail that forms as a result of traffic, either by humans or other animals, and often veers away from conventional paths.

You may have seen desire paths yourself as dirt tracks that cut a more direct line between paved paths or through vegetation.

The same idea applies to the artifacts produced by your IDP. Suppose your customers have edited them to suit their purposes. Those edits must be considered valuable feedback indicating that the source artifacts may need modification to support their intended use case.

The responsibility triad

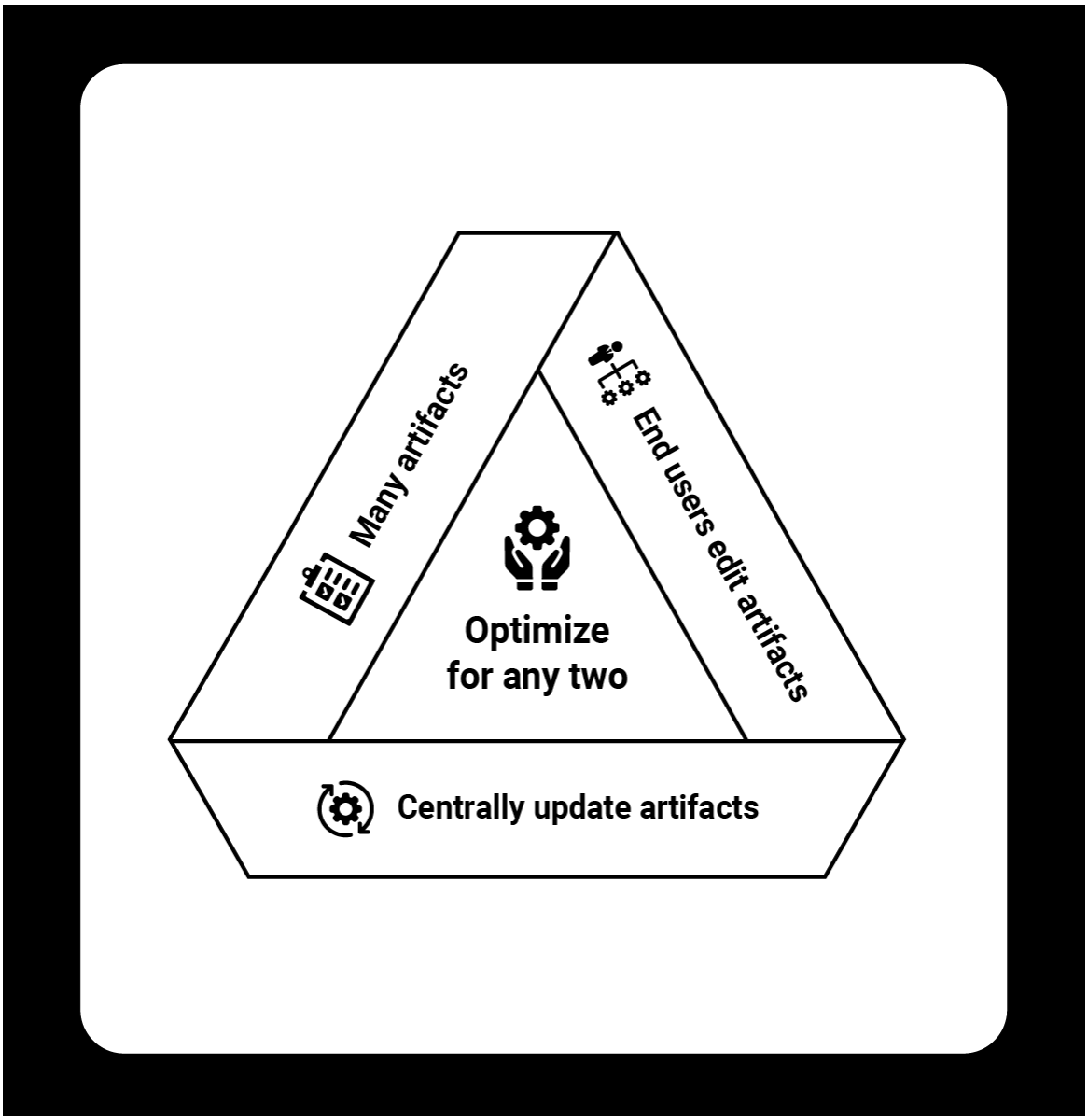
You'll have 3 competing concerns as you develop your IDP, captured in the responsibility triad:

1. To have many artifacts
2. To centrally update existing artifacts
3. To have end-users modify their artifacts

In practice, you can optimize any 2 concerns of the responsibility triad:

1. You can have many artifacts and update those artifacts over time. However, customers will have limited ability to modify those artifacts.

2. You can have many artifacts and let customers modify those artifacts. But you'll have limited ability to update those artifacts centrally.
3. You can centrally update artifacts and have customers modify them. But you'll only be able to support a few artifacts.



Platform teams must plan for the trade-offs imposed by the responsibility triad. Otherwise, teams will unexpectedly paint themselves into one of its corners. There are common strategies for platform teams that ensure the trade-offs are intentional.

Initially, your mission statement must limit the number of created artifacts. Deploying too many artifacts without building the muscle to update those artifacts will leave you with a graveyard of good intentions.

Policies or other restrictions that limit how much customers can edit shared responsibility artifacts reduce the burden of centrally pushing changes. For example, imagine if customers could edit steps 2 and 3 in a deployment

process, while steps 1, 4, and 5 are read-only. The platform team could reliably publish updates to the read-only steps without affecting customer modifications.

Providing read-only processes but supporting customer-edited variables is a specialized policy that allows customers to customize their artifacts while ensuring the platform team has a stable target for centralized updates.

Maintaining multiple versions of your artifacts lets you add new features to later versions while distributing only critical fixes to previous versions. The SemVer versioning scheme, which uses a x.y.z scheme, captures this idea as a major version (or x version) increment. This lets your customers treat artifact updates like software updates and schedule the integration of new major versions when needed.

Preferencing the customer responsibility model over the shared responsibility model limits the burden on the platform team to resolve conflicts in edited artifacts. The centralized responsibility model also reduces the conflict burden but has the downside of potentially making the IDP a blocker for your customers.

Passing the responsibility to merge changes to the customer has the effect of outsourcing the work of resolving conflicts. This strategy relies heavily on your feedback channels, as this is how you make your customers aware of pending changes and the urgency to perform updates.

The one plan that will eventually fail is to try and deploy a vast number of customizable artifacts and then try to centrally maintain them over time. An IDP is only as valuable as its ability to affect change at scale. Keeping the responsibility triad in mind as you develop your IDP ensures its scalability over the long term.

Choosing a responsibility model

Those looking to introduce platform engineering into existing DevOps teams face 3 common realities:

- Your IDP is a greenfield project
- Your platform team must reduce the mental burden of an existing, complex environment
- Your customers are technically adept

The purpose of an IDP is to create opinionated artifacts embedding the accumulated business knowledge and best practices learned by your customers over the years. By design, these artifacts abstract away much of the institutional knowledge needed to build high-quality solutions. They instead let DevOps teams focus on solving novel problems, rather than duplicating the effort required to solve already solved problems.

The challenge when developing such abstractions is to strike the right balance between hiding complexity and enabling flexibility.

Platform teams have the added challenge of being compared to the established platforms their IDP seeks to abstract away. The reality is that these existing platforms were adopted because of the many years, countless hours, and eye-watering sums of money that were spent refining the balance between complexity and flexibility. For example, popular operating systems, hypervisors, cloud platforms, and container orchestration platforms all have billions of dollars worth of development behind them.

Meanwhile, a good number of your customers will be comfortable with these underlying platforms. This means they'll quickly spot, and likely get frustrated by, the limitations of your abstractions. This risks making your IDP the kind of blocker it was supposed to remove.

It's not a question of whether abstractions are valuable or possible to build correctly. Every successful piece of technology is an example of building a useful abstraction. A car abstracts the complexity of an engine and transmission behind a steering wheel, accelerator, and brake. An electrical powerpoint abstracts the continent-sized infrastructure of a power grid. And a phone abstracts a worldwide communication network. The better question, especially for new platform teams, is "Do we have the knowledge to create the correct level of abstraction?"

The only safe initial answer to this question is "No", with an aim to reach the point where the answer is "Yes, but...". While you'll never have perfect knowledge of your customer's needs, a successful platform team can refine their abstractions over time. But you do need to have some level of early success to earn the time required to refine the artifacts provided by your IDP.

Anyone aiming to provide an opinionated solution faces this same dilemma, but the survivorship bias of every successful abstraction you see in daily life can make it seem like designing abstractions is easy. However, new platform teams must assume designing abstractions is, in fact, quite difficult, and therefore have a plan to support customers with unique needs or opinions of their own. This is the crux of the previously discussed responsibility models:

- Centralized responsibility (or "enforced consistency") artifacts provide highly opinionated solutions exposing a relatively small number of options to customize the end result. The success of the centralized responsibility model is highly dependent on how closely aligned your abstractions are to the needs of your customers.
- Shared responsibility (or "eventual consistency") artifacts can be modified by customers, providing them with the ability to customize their solution while still allowing the platform team to centrally distribute updates. These artifacts are the most flexible and provide a sliding scale between enforcing opinions and exposing flexibility (with the caveat that increased customization will typically decrease the ability to centrally distribute updates). The shared responsibility model assumes abstractions will be imperfect and emphasizes a workflow that lets platform teams and their customers jointly refine the solution over time.
- Customer responsibility (or "eventual inconsistency") artifacts provide customers with a base template that they're expected to own and modify to suit their needs. While the templates can provide an initial set of opinions, they're expected to be heavily modified. The customer responsibility model assumes customers will keep what's useful and discard what's not.

An anti-pattern to avoid when starting a new platform team is focusing too heavily on the centralized responsibility model. This model requires that you build the most accurate abstractions, and starting a new platform team is the time when you have the least amount of information to do so.

Instead, new platform teams should aim to initially offer artifacts under the shared responsibility or customer responsibility models. These models embrace the reality that abstractions evolve over time, and in the case of the shared responsibility model, prioritize processes that observe and incorporate those changes. This approach aligns with Gall's Law, which states:

> A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple system.

To put the difficulty of designing abstractions into perspective, consider that Kubernetes added the Deployment resource in version 1.2 to expand on the capabilities of a ReplicaSet and has been working since 2019 to introduce the Gateway resource to address limitations with Ingresses. These are just 2 examples of the constant refinement platforms like Kubernetes undergo as they find the most effective level of abstraction. And all of this is after the many years that Borg, the predecessor to Kubernetes, was under development internally in Google, as noted in the blog post Borg: The Predecessor to Kubernetes:

> Kubernetes traces its lineage directly from Borg. Many of the developers at Google working on Kubernetes were formerly developers on the Borg project. We've incorporated the best ideas from Borg in Kubernetes, and have tried to address some pain points that users identified with Borg over the years.

In the unlikely event that you managed to get your abstractions just right first time, there's little practical difference between the 3 responsibility models, as the resulting solutions will all remain mostly unchanged from the initial artifact generated by your IDP. However, in the more likely event that your abstractions need refinement, the shared responsibility model lets you work with your customers to evolve your abstractions, while the customer responsibility model lets customers make whatever changes they need.

I was part of a team that designed new steps in Octopus to deploy applications to Kubernetes and AWS ECS. One of the challenges in designing these steps was deciding how to support use cases we hadn't anticipated or might be introduced by the underlying platforms after our steps were released. After spending some time thinking about all the hooks or custom scripting we could expose in these steps, we decided that the best course of action was to offer both highly opinionated steps that supported most common use cases and generic steps that allowed the deployment of raw YAML for either Kubernetes resources or CloudFormation templates. Importantly, the opinionated steps offered a way to export the equivalent YAML they would generate, giving customers a way to jump from the opinionated step (which aligns with the central responsibility model) to the generic step (which aligns with the customer responsibility model).

This kind of "break glass in emergency" process to move from one responsibility model to another meant we could offer a path for any customers blocked by a lack of functionality in the opinionated steps, with the understanding that customers had to take more responsibility for what was now considered bespoke YAML. This is a practical, and now battle-tested, example of the responsibility models in action.

Example checklist: Defining responsibility models

This example checklist lists artifacts produced by the IDP and defines the responsibility model associated with each. Strategies for minimizing the impact of customer-edited artifacts are noted, with policies limiting changes or the process of updating artifacts delegated to the customer.

Artifact	Responsibility model
Database backup scripts	Centralized
Microservice deployment processes	Shared with updates pushed centrally
Microservice bootstrap code projects	Customer
Virtual machine creation and destruction scripts	Centralized

Serverless deployment process	Shared with policy enforcing required approval and reporting steps
Linux app deployment process	Shared with customers pulling in changes

Conclusion

DevOps teams have always been able to duplicate artifacts, as most tools have some form of copy-and-paste functionality. While building high-quality upstream artifacts is an essential function for any platform team, much of the value derived from an IDP is its ability to distribute updates while still allowing customers to edit their artifacts.

The customer, shared, and centralized responsibility models let platform teams be transparent about who is responsible for maintaining long-lived artifacts. While allowing customers to edit their artifacts is an effective strategy against the platform team becoming a blocker, the responsibility triad shows that they come at the cost of being able to update the artifacts centrally or deploy large numbers of them.

The value of Platform Engineering

Despite being relatively new, studies, articles, and industry examples prove the value of Platform Engineering to engineering organizations.

This chapter is most useful for organizations considering establishing a platform team. Such a request requires a compelling argument for funding. It's essential to relate the value of a product team and the IDP they create back to business goals. Supporting those arguments with industry survey results and external examples of success gives you credibility. It provides those who fund this team confidence that Platform Engineering is not a fad.

Platform Engineering by the numbers

The HashiCorp “2023 State of Cloud Strategy Survey” report (https://oc.to/vr6S2Q) notes that:

> The presence of a centralized platform team is a foundational element of a well-rounded multi-cloud strategy.

The report then says:

> Platform teams are a key pillar of operational maturity. These centralized hubs effectively manage people and standardize processes, but organizations need skilled professionals to staff them. Those without platform teams cited a lack of skills/staff (36%) as the reason why they hadn’t put one in place to manage their cloud strategy — a rise of 14 percentage points from 2022’s State Of Cloud Strategy Survey.

The Puppet “State of DevOps Report Platform Engineering Edition 2023” (https://oc.to/7O5fqO) reports that 68% of respondents saw an increase in development velocity after implementing a platform team. In addition, 93% of respondents reported that development speed after the inception of a platform team increased “A great deal” or “somewhat”, with only 5% reporting “very little”.

Platform teams also improved DevEx, with the report showing that:

> Most notably, respondents at firms with platform engineering in place are more likely to identify as “very satisfied.”

The Port 2024 State of internal developer portals report asked respondents how much time was spent on non-core work because of a lack of platform engineering:

> We asked how much time developers spend each day on non-core work because of not having platform engineering in place.
>
> 78% of respondents report three or more hours of developers’ time spent daily on non-core work.

In his presentation “Why did you build an Internal Developer Platform at GitHub?” (https://oc.to/ItKChg), Jason Warner, CTO of GitHub, notes that scaling processes with people was not a viable option:

> When we hit that scale on operations, or the infrastructure team, [who were] responsible for coordinating all those things that were happening and how to release it, it was unmanageable.
>
> You couldn’t scale that with people.

Eugene Kim of Insider (https://oc.to/8V7w5I) reported the efforts of Amazon CEO Andy Jassy to fix a crumbling engineering culture with a new unit called Amazon Software Builder Experience (ASBX). The article notes the engineering team's frustrations:

> In a recent internal survey, seen by Insider, 34% of the engineers said they spent 4 to 8 hours on undifferentiated efforts weekly, or roughly 10% to 20% of their week on things that are not related to building new products. Another survey found that engineers spend 30% of their time on "repetitive tasks."

The article goes on to list the 6 tenets used by the ASBX team:

1. Software builders across Amazon require consistent, interoperable, and extensible tools to construct and operate applications at our peculiar scale; organizations will extend on our solutions for their specialized business needs.
2. Amazon's customers benefit when software builders spend time on novel innovation. Undifferentiated work elimination, automation, and integrated opinionated tooling reserve human interaction for high-judgment situations.
3. Our tools must be available for use even in the worst of times, which happens to be when software builders may most need to use them: we must be available even when others are not.
4. Software builder experience is the summation of tools, processes, and technology owned throughout the company, relentlessly improved through the use of well-understood metrics, actionable insights, and knowledge sharing.
5. Amazon's industry-leading technology and access to top experts in many fields provides opportunities for builders to learn and grow at a rate unparalleled in the industry.
6. As builders we are in a unique position to codify Amazon's values into the technical foundations; we foster a culture of belonging by ensuring our tools, training, and events are inclusive and accessible by design.

While the ASBX team doesn't specifically describe itself as a platform team, a focus on "undifferentiated work elimination, automation, and integrated opinionated tooling", the relentless improvement of "software builder experience", and "codifying" organizational values all speak to the principals of a platform team.

More than the specific highlights above, the most valuable things to take away from this chapter are the reports and industry examples that quantify the value of platform teams and IDPs. These examples are invaluable to have in your back pocket when quantifying the value of a platform team.

Quantifying the business value

There are many ways to show the potential value of a business proposal. But, as we'll discover in the "Planning your Internal Developer Platform" chapter, the initial members of a platform team are likely to be generalists rather than seasoned salespeople. So, we'll need a straightforward methodology to articulate the business value of a platform team.

It's worth acknowledging that quantifying the value of a platform team is not a trivial task. In the podcast "The Journey To Build your Platform Engineering Team" (https://oc.to/a07RuP), Alejandro Sanchez-Giraldo, Head of Quality Engineering & Observability At DevOps1, noted:

> I think the funding model is one of the biggest challenges around trying to build the platform engineering teams, because to my point it's really hard to have a direct correlation to the value that it brings.

Paul Kelcey, VP Engineering at NurtureCloud, then says:

> It is actually very difficult because it isn't always like a direct path.
>
> I find it's good to attack it on 2 fronts. The first one is to talk about what the value is by introducing it. And the second is to talk about the cost of not introducing it.
>
> You can talk about like the rate at which you know features are being released and how that's decreasing. And you can extrapolate - obviously this is not sustainable.

The post “Identified Pain Is About Customer Outcomes” (https://oc.to/iktukN) lists 4 concerns, or pain points, that all business initiatives must consider:

- Revenue – is there a pain impacting their booking revenue? Does your solution solve this pain?
- Cost – is there a pain driving up the cost of doing business? Can your solution solve this pain?
- Risk – is there a new regulation they are in danger of breaching? Does your solution mitigate the pain?
- Shareholder commitments – have they committed to something to their shareholders, which they'll fail to deliver without your solution?

Any new initiative must ultimately impact one or more of these concerns, where risk and cost decrease while revenue increases. It is harder to show independent increases to shareholder value, but it's included here for completeness.

Emma Maslen, the author of the post above, included shareholder value. She did this to capture cases where businesses were working towards efforts like environmental initiatives or diversity, equity, and inclusivity that may not necessarily impact risk, cost, or revenue. In most cases, shareholder value is a consequence of changes to the other 3 pain points.

Fortunately, an IDP has plenty of scope to impact business value positively. By infusing best practices at scale throughout your DevOps teams, it is reasonable to argue that your IDP directly:

- Reduces friction
- Improves employee satisfaction and engagement
- Improves deployment frequency
- Improves compliance

Importantly, these improvements are repeatable and scalable. The final step is to describe these improvements in terms of business concerns like risk, cost, or revenue.

You need to express the business value of a platform team as a number to ensure it is measurable. Defining measurable values involves answering questions like:

- How much money will the organization earn from new features delivered as a result of the increased performance of DevOps teams?
- How much will standardizing on security practices across all projects reduce reputational risk?
- How much money will the organization save by removing the need for DevOps teams to manually respond to a newly published CVE?
- How many fewer staff members will leave DevOps teams in a year if the team removes 20% of undifferentiated work?

Metrics and numbers can be sensitive subjects. It's easy to impose arbitrary values on people who may not appreciate the implications. Pia Nilsson, Lead Platform Developer and Experience Lead at Spotify, noted in her talk "Developer Enablement at Spotify" (https://oc.to/BXqhGK) that co-creating metrics has been a successful approach:

> I think the way I've seen it successful, and it took us a few years to get this right too, is to really go to the organization within your company, approach these leaders with empathy [...], and help them co-create metrics that would matter to them.

Spare-time platform teams

A common symptom of an organically grown Platform Engineering strategy is spare-time platform teams. They're usually made up of a few enthusiastic engineers. These people tackled the technical aspect of an IDP and demonstrated enough value in their immediate orbit to justify maintaining it, if only in an ad-hoc manner.

Spare-time product teams can do fantastic work. As Eric S. Raymond teaches as the first lesson in "The Cathedral and the Bazaar":

> Every good work of software starts by scratching a developer’s itch.

However, such teams are one reorganization away from getting dispersed amongst their broader DevOps ecosystem. This makes them fragile. Clearly articulating the business value of a platform team makes it more resilient to changing business conditions.

Example checklist: Impact of DEaaS on core business needs

The checklist below provides an example describing how DEaaS may positively impact core business concerns.

Business Concern	Impact
Risk	An internal review identified 15 dependencies used in production applications with vulnerabilities rated as critical. These vulnerabilities expose the company to data breaches and denial of service attacks. The current process of discovering vulnerabilities is reactive and ad-hoc, leaving production applications vulnerable for many months. DEaaS providing CI/CD pipelines including standardized security scanning steps lets DevOps teams identify which applications have vulnerable dependencies within one day of vulnerability reports being publicly released.
Cost	Exit surveys indicate that undifferentiated work contributed to 20% of those who quit last year. By reducing undifferentiated work, DEaaS aims to reduce staff turnover and the cost of training new staff by $100,000 annually.
Revenue	New features took, on average, 6 months to be released to customers last year, with 17 lost

opportunities due to missing product functionality.

DevOps teams have identified that time spent waiting for infrastructure provisioning contributed at least 4 weeks to this release time.

By exposing self-service infrastructure provisioning, DEaaS will reduce release times by 10%, allowing up to 3 new features to be released per year, resulting in the completion of 5 additional deals worth $75k per year.

Conclusion

If you're looking to introduce a platform team, you must present a compelling business case for an IDP and the platform team responsible for it. This chapter provides insights into the benefits of Platform Engineering based on industry reports and examples.

We then distilled business value down to 3 pain points to attach quantifiable value to. We also provided an example checklist you can use when pitching a platform team to your organization.

Planning your DEaaS implementation

Everything we have discussed in this book has been leading to this point. We now understand:

- Why we are implementing DEaaS - to improve DevEx
- Who is responsible - the platform team
- How the service is delivered - an IDP
- What is being delivered - the functional and non-functional requirements that most improve the team's DevEx

Implementing DEaaS is not as simple as deploying an IDP and telling everyone to use it. It effectively requires your platform team to run an internal startup. This means you're not providing a piece of software or single solution but a complete service to your internal customers.

Running an internal startup can be daunting. Platform teams must draw from the work typically done by dedicated engineering, support, technical writing, training, developer relations (DevRel), product, and design teams. However, addressing a few key activities early in the inception of a platform team provides a solid foundation for future growth. It also retains the agility to keep pace with high-performing DevOps teams.

Building your platform team

It can seem like platform teams are saddled with the same do-it-all-yourself mentality as their DevOps customers. Indeed, in the early stages, platform teams will attract generalists who enjoy tackling various challenges but with a clearly defined mission to avoid taking on an impossibly broad scope of work.

Over time, platform teams evolve to incorporate the kind of specialized roles found in other product-focused engineering teams. The "State of DevOps Report Platform Engineering Edition 2023" (https://oc.to/7O5fqO) notes that:

> At firms in which a platform team has existed for three or fewer years, 60% of respondents identify a need for a product manager; at firms in which a platform team has existed for more than three years, this number rises to 74%.

You can expect a successful platform engineering strategy to work closely with, or incorporate, other specialized roles. These include:

- DevRel
- Technical support
- Technical writers
- Security
- Site reliability

That said, though, generalists will serve your initial platform well by being able to draw on experience from various backgrounds.

The Forbes article “Generalist Vs. Specialist Teams: How To Build A Highly Creative Startup” (https://oc.to/jkdl9C) notes that the unpredictable nature of startups does not support specialization:

> In highly-predictable environments, specialization allows you to solve problems very efficiently. In unpredictable environments, however, efficiency isn't as important as creativity. And the environment of innovative startups is anything but predictable.

The flip side of bringing people into a platform team is to consider the option of secondments. This lets you embed members of the platform team back into the wider DevOps ecosystem as a method of sharing knowledge and experience. David Jorm, Head Of Security At x15ventures, noted in the podcast "The Journey To Build your Platform Engineering Team" (https://oc.to/a07RuP) that:

> One of the ways to solve [architectural and cultural issues] is to get secondments from the core platform team into that individual engineering team so that I can access some of the thinking behind what was done [by the platform team].

The activities listed in this chapter support teams in the initial stages of their platform engineering journey. The aim is not to make the platform team experts in product management, marketing, technical writing, and training. Instead, these activities reinforce the need to treat platform engineering like a startup rather than a feature to be delivered in a sprint and provide a solid foundation for operating an internal product.

Mission and focus

The platform team's mission and focus are crucial to ensuring you deliver the highest value solutions to your DevOps teams. Some examples of mission statements are:

- Enforce conformance by provisioning standard deployment pipelines for all microservices.
- Improve security and maintainability by enabling security scanning against all production deployments.
- Manage costs by providing the ability to create standardized cloud resources with self-service workflows.
- Standardize observability by adding monitoring capabilities to all public-facing applications.
- Increase reliability by automating fault detection and rectification for all Kubernetes applications.

The mission statement must capture the platform's functional requirements (what the platform will do). And it must capture non-functional requirements (capabilities that the platform will support, typically "ilities" like observability, scalability, and maintainability). It must also positively influence the business concerns listed in the "Quantifying the business value" section of cost, risk, revenue, and shareholder value.

A simple statement that defines the team's mission will provide context for all their subsequent activities. For example, by focusing your user interviews on pain points and aspirations related to the mission while triaging support and feature requests based on their alignment with the mission.

The mission statement also guards against the platform team becoming a reactive, general-purpose support platform for wider DevOps teams.

User interviews

Talking to your DevOps teams and their management is critical to understanding the opportunities for a platform team to improve DevEx. These conversations will quickly highlight the pain points in your company's current processes, and surface many aspirational goals.

Your IDP will be bought by your peers, not sold by their managers. So the people you interview are the same people you're trying to sign up for your internal startup. You'll gain your customer's trust by taking note of their pains and wishes and repeating these observations as you talk about the goals and features of your IDP.

Courtney Kissler led large-scale reorganizations and technology transformations at Starbucks, Nike, and Zulily. She notes in her presentation “How to build internal platforms for the enterprise” (https://oc.to/V0pDUd) that:

> Just because you build a platform doesn't mean people are going to consume it.
>
> It has to solve a need, it has to be compelling, and you need to treat it like a product.

For those from an engineering background, talking to customers can elicit a feeling of dread. As an engineer, I never wanted to talk to customers because every customer I ever interacted with asked me to fix a bug or solve a problem. This order-taking was tolerable via asynchronous communication like email, but face-to-face conversation felt like an invitation for unbounded personal technical support, which was the last thing I wanted.

I wasn't necessarily wrong (I've had more than a few "I hope it is OK to email you directly, but I was wondering if you could ..." emails after one-on-one conversations). Fortunately, internal user interviews are far more informal and enjoyable. They have proven to be a great way to build camaraderie and surfaced many wishes and requirements for our IDP.

Just as important as listening to your audience is the ability to challenge them with new ideas. Even if you're given the opportunity to build an IDP, user interviews are still as much about selling the benefits of an IDP as about gathering requirements (or, in other words, being an order-taker).

The Forbes article “Why Most Salespeople Are Order Takers And Not Sellers” (https://oc.to/VmnpgR) notes that:

> Selling is about change, and it's salespeople's job to influence that change. Salespeople who can't influence change are simply order-takers.

Gregor Hohpe similarly notes in his book “The Software Architect Elevator” that modern IT departments have moved beyond order taking:

> Positioning IT architecture on par with business architecture highlights, though, that the days when IT was a simple order-taker that provides a commodity resource at the lowest possible cost are (luckily) over. In the digital age, IT is a competitive differentiator and opportunity driver, not a commodity like electricity.

You're uniquely positioned as a platform team member to:

- Understand the different viewpoints held by your DevOps teams
- Share those insights with a broader audience
- Sell a vision for how a DEaaS can influence positive change in an organization

Remember, you're running a startup, not building a better conveyor belt.

For those platform teams tasked with improving their CI/CD pipelines, the chapter "Platform Engineering Non-Functional Requirements" lists typical deployment pipeline non-functional requirements. These provide valuable prompts to surface desirable qualities as you talk with your customers.

Documentation and training

Your IDP is a tool that tool builders use. It integrates many other tools and platforms. It needs to be flexible, configurable, and customizable. In short, it's a bespoke product encapsulating and supporting the specific requirements of your internal teams and organization.

Your customers don't have the luxury of Googling how to use this internal tool. There are no posts on Stack Overflow to point them in the right direction. The lack of external community support means your platform needs robust documentation and training for your internal customers to get the most out of it.

You can measure the success of your IDP by its ability to embed best practices more broadly than would be possible if individuals implemented them in each team or project. This scale is only possible if DevOps teams can understand and implement the solution without needing a personal demonstration every time. Documentation and training are 2 proven solutions for upskilling customers at a time and pace that suits them.

Documentation quality is one of the DevOps capabilities noted by DORA (https://oc.to/6RM3FU):

> Internal documentation is a fundamental part of software development. Analysis by DORA has found a clear link between documentation quality and organizational performance — the organization's ability to meet their performance and profitability goals.

The table below highlights the lift to organizational performance achieved by teams with below-average and above-average documentation:

	Lift to organizational performance	
Technical capability	Teams with below-average documentation quality	Teams with above-average documentation quality
Continuous Delivery	63%	656%
Continuous Integration	34%	750%
Loosely coupled architecture	46%	313%
Site Reliability Engineering (SRE)	79%	343%
Supply chain security	37%	451%
Trunk based development	36%	1525%
Version control	27%	278%

"Any product that needs a manual to work is broken." It's a catchy quote from Elon Musk, and yet the first time I caught a Tesla Model 3, my Uber driver had to open the door for me because I didn't know how the handles worked. Thankfully, and ironically, Tesla's online owner's manual has excellent documentation on how to open the doors. I'll know better next time.

Support

A considerable aspect of the ongoing operation of your IDP is the support you offer your internal customers. It's tempting to think you'll have a captive audience with a mandate to use your platform. But in reality, you're competing with how DevOps teams used to do things. If your IDP doesn't work as expected (which is inevitable at some point, given the scope of Platform Engineering), you must have a responsive support channel. Otherwise, your customers will revert to their existing processes.

Your documentation and training efforts can significantly complement your support processes. Ideally, supporting your customers means pointing them to the documentation or training that answers their questions, addressing a bug, or accepting a feature request.

A responsive support layer is critical to prevent your solution from being the blocker a platform team should remove. Nothing is more detrimental to DevEx than the broken flow caused by a blocking issue and an unresponsive support team.

Note that being responsive is not the same as solving every problem. It's an anti-pattern to move all the pain to the platform team by expecting them to provide site reliability services, general advice, or function as a help desk.

Platform teams must balance many concerns while interacting with their customers:

1. To provide internal customers with opinionated solutions.
2. To support these solutions to prevent customers from being blocked.
3. To encourage customers to implement their own customizations and incorporate those improvements into your IDP.
4. To manage the potential expectation that the platform team will immediately support anyone with any issue.

The best way to maintain this balance is to clearly define levels of responsibility. This ensures the platform team and their customers have a clear understanding of who is responsible for maintaining the artifacts generated by an IDP. The chapter "Platform Engineering responsibilities" discusses different responsibility models.

Issue tracking and feature requests

The ability to track the status of issues discovered in your IDP and to track any progress with feature requests provides your customers confidence that bugs are being fixed and requests are taken seriously.

Your customers will be technical people familiar with issue-tracking platforms like Jira, GitHub, and Bugzilla. You'll likely have an issue tracker already in use by your DevOps teams, and reusing it for your IDP is a natural choice.

Be confident enough to admit you must wait to implement new features. While working in product teams, I was overwhelmed with requests for new features and improvements and felt that confessing that we wouldn't consider even great ideas for at least 6 to 12 months was admitting defeat. Often, I just ignored these requests, but that was a mistake. Having been on the other side of that conversation, I'd prefer an honest answer to no answer.

Marketing/Evangelism

Building an IDP is a long-term commitment. As you fix bugs, add features, or respond to changing requirements, you must be able to publish new and improved solutions to your internal customers.

Typically, your IDP will either push changes out to your customers or let them pull the changes as they're made available. If you opt for a pull-based approach, you'll need a method for notifying your customers of new features.

However, as impressive as your IDP may be, you'll only need such a solution when your customers already have a dozen other platforms they use every day. A challenge of running a startup is respecting your customers' attention and not abusing that relationship with noise.

Most DevOps teams have an internal chat solution, which can be a low-friction method of announcing features or changes. Lunch-and-learn sessions are another excellent way to reach your internal customers. Also, consider recording a demonstration of the new features and sharing it as a monthly webinar.

One of the tips I learned when I moved to a sales team was the value of existing, general-purpose meetings preallocated in a team's calendar. Never underestimate how hard it is to find a time for everyone to meet. Being able to present at a lunch-and-learn is an incredible opportunity for your marketing efforts.

Feedback and metrics

Feedback from your customers is crucial to understand whether you're meeting their expectations. Metrics let you measure the impact of your IDP. Both need the platform team to have clear processes for collecting data. Feedback is generally sourced directly from customers and metrics are directly measured from the systems established by your IDP or the IDP itself.

The data collected falls into 3 categories:

- IDP metrics to directly measure the characteristics of your IDP
- Key customer metrics to measure the impact of your IDP's artifacts
- Feedback from customers to gauge their perceived DevEx

IDP metrics

You can find IDP metrics by measuring the interactions with your IDP or the ongoing operation of resources generated by your IDPs artifacts. Some examples of these metrics include:

- The number of IDP artifacts deployed.
- How frequently the IDP artifacts are used. For example, how many times a IDP generated CI/CD pipeline is run or how frequently a IDP generated dashboard is viewed.
- The percentage of DevOps resources created by your IDP. This measures your IDPs market share. For example, using tags to identify which cloud VMs were created by your IDP and how many were created by hand.

Key customer metrics

Your IDPs artifacts can empower your customers to measure their own performance by exposing the ability to measure key customer metrics.

The DORA metrics provide a well-researched set of measurements that DevOps teams can use to measure their performance.

Metrics that gauge the core DevEx dimensions are used to complement the perceived DevEx. The section "What is DevEx?" noted that 3 core dimensions categorize the friction developers experience:

- Feedback loops
- Cognitive load
- Flow state

Some example metrics include:

- Feedback loops: Build times, code review waits, deployment frequency, lead time for changes
- Cognitive load: The number of manual stages in the deployment pipeline, lead time for technical questions
- Flow: Meeting free blocks, incident frequency, number of unplanned tasks

You may also have other key performance indicators to measure the performance of DevOps teams. A common example is onboarding time. Spotify documented its approach to metrics in the post "How we measure Backstage success at Spotify" (https://oc.to/Di3qdM), noting that the "time-to-10th PR" is their north-star.

Perceived DevEx

The final category of feedback involves surveying your customers to understand their perceived DevEx. Because your customer's perceived DevEx is a subjective measurement, you must ask them directly how they feel about the 3 core DevEx dimensions. Some questions include:

- Feedback loops: How satisfied are you with (build times, test run speed, the time it takes to get a change live)?
- Cognitive load: How complex is (the codebase, debugging the system, understanding the documentation)?
- Flow: Are you able to (focus on work, avoid interruptions, understand your tasks)?

You may also include a Net Promoter Score (NPS) survey to track how your DevOps teams are perceived by their members.

Example checklist: Planning your DEaaS implementation

The following checklist is an example that documents how you might address the activities noted in this chapter. As you can see, this example checklist is not complicated, but it does commit the platform team to:

1. Defining the mission statement
2. Selecting individuals for user interviews
3. Writing introductory documentation
4. Creating onboarding training
5. Selecting support and notifications platforms
6. Committing to regular internal announcements to demonstrate new features
7. Planning follow-up user interviews at regular intervals to gain feedback

Process	Plan
Mission	Enforce conformance by provisioning standard deployment pipelines for all microservices.
User interviews	

	1. Jane Smith 2. Sunita Singh 3. Lin Zhou 4. Mike Jones
Documentation	Getting Started guide in GitHub Wiki
Training	5-part video training series: 1. Creating the first deployment project 2. The features required in every project 3. Customizing the project 4. Updating the project with upstream changes 5. Getting support and requesting features
Support	#platform-engineering in Slack
Issue tracking	GitHub issues: 1. Issues tagged with “bug” 2. Issues blocking DevOps teams or compromising security tagged with "high" 3. Non-blocking issues tagged with "low" 4. High priority issues take precedence over feature work 5. Feature requests tagged with “enhancement”
Marketing / Evangelism	1. A channel announcement in #platform-engineering 2. A summary of new features in the monthly lunch-and-learn session
Feedback	Track market share by measuring the percentage of CI/CD pipelines that were created by the IDP. Track the DORA metrics of any team using CI/CD pipelines generated by the IDP. Survey DevOps teams about their perceived DevEx every quarter. Include an NPS question in the survey to measure how likely DevOps team members would recommend a friend for a job with this company.

Conclusion

This chapter reinforced the importance of treating your Platform Engineering initiative as an internal startup within your company. You're not building a feature. Instead, you're creating an end-to-end solution with all the business processes customers would expect from any business-critical application they adopted for their day-to-day operations.

Platform Engineering is not a trivial undertaking, but for those lucky enough to be involved, it's an exciting opportunity to run a business in a business.

The activities listed in this chapter come before any discussion about implementing an IDP. Prioritizing these activities is quite deliberate and highlights the importance of planning for success before implementing the solution. The example checklist provides an easy way to ensure these requirements are tangible and not afterthoughts.

Epilogue

We ask a lot of DevOps teams. These teams are engines of innovation using some of the most complex tools ever created while building harmonious and efficient environments able to assume responsibility for any new requirements.

Every DevOps team strives to be high performing. But as James Clear noted in his book "Atomic Habits":

> You do not rise to the level of your goals. You fall to the level of your systems.

DEaaS is an approach to maintaining systems, or architectural decisions, that can be implemented at scale throughout DevOps teams using the practices of platform engineering with the explicit purpose of improving DevEx. By adopting DevEx as the goal of platform engineering, DEaaS provides a clear and measurable outcome as the foundation for high performing DevOps teams.

Or, in other words, DEaaS provides systems that high performing DevOps teams can fall back to.

This book covered the activities required for DevOps teams looking to establish DEaaS, including:

- Articulating your IDP's value to high-level stakeholders, who are ultimately responsible for investing in your ideas
- Gathering and inspiring ideas for how your IDP will support DevOps teams to achieve their goals while removing pain points from existing processes
- Establishing the support, feedback, and internal marketing processes to keep your customers happy and informed
- Building feedback loops to ensure your priorities align with your customers
- Clearly articulating the responsibilities of the platform team and your customers to update and maintain the artifacts generated by your IDP
- Understanding the pillars of a holistic DevOps lifecycle and sharing those insights with your customers

By following the processes outlined in this book, organizations looking to establish a platform team benefit from a set of activities and practical checklists that ensure the product you build improves DevEx and the performance of your DevOps teams for many years to come.