The Platform Engineering Playbook

THE
PLATFORM
ENGINEERING
PLAYBOOK
Standardize the way you design, manage, and deploy systems for everyone
MICHAEL LEVAN

About The Author

Michael Levan

Michael Levan is a seasoned engineer and consultant in the Kubernetes space who spends his time working with startups and enterprises around the globe on Kubernetes consulting, training, and content creation. He is a trainer, 3x published author, podcast host, international public speaker, CNCF Ambassador, and was part of the Kubernetes v1.28 Release Team.

About The Book

Oh, look... ANOTHER title... "Platform Engineering".

Yeah, it's a new title, but this title actually makes sense.

The goal of The Platform Engineering Playbook is to help you understand:

1. Why Platform Engineering is a discipline that’s actively emerging in the space.

2. What it actually means.
3. Why it’s important.
4. How to implement it.

This book will give you everything from the theory behind Platform Engineering to how it works with teams across your organization to how to create the stack needed for proper Platform Engineering.

By the end of this book, you’ll have not only the understanding of how to implement Platform Engineering, but a great starting point to implement it from a hands-on perspective as the last two chapters focus on creating a stack and creating self-service capabilities.

Book Outline

This book, although not specifically split up into three parts, will more or less be split up into three parts… that sounds confusing, but let me explain.

Chapter 1 is all about the “why” behind Platform Engineering. The entire evolution, including the key differences between DevOps, SRE, and Platform Engineering. We can call this “part 1”.

Chapters 2-4 will be more theory, or rather, planning for Platform Engineering. It’ll include the thought process, design decisions, and most importantly, team decisions… because remember, your customer as a Platform Engineer are the internal engineers and developers, so ensuring that your team and the other teams in the organization are on board with what you’re doing is crucial. We’ll call this “part 2”.

Chapters 5-7 are the hands-on implementation chapters. You’ll be doing everything from architecting a solution to implementing the solution to building self-service and Internal Developer Platforms (IDP) for the engineering teams and development teams using the systems. We’ll call this “part 3”.

Chapter 1: The Evolution of Platform Engineering

Ah, yes… a new title, a new job function, new capabilities, new engineering methods… and more importantly, new ways to confuse everyone.

There are a lot of titles, job functions, and methods that although are set out to help engineers make their lives easier, it just ends up making everyone's job harder. The more complex things get in the tech space, or the IT world, or the cloud-native world, or whatever else you want to call it, the harder it is to grasp what we're supposed to be doing.

I've been in tech for well over a decade at this point, and the amount of change that's occurred is absolutely wild. Everything from focusing on Active Directory and Exchange being a hot topic to focusing on Virtualization to focusing on Cloud to everything in-between. The evolution of how we deal with platforms, systems, and implementations feel like they're drastically changing all of the time.

With that change comes the changing of titles, job functions, and just about any other buzzy thing you want to say or think. Again, although it's for good purpose, it simply ends up confusing everyone that's not only been in tech for years, but especially the new up and coming engineers.

Platform Engineering is the one new and emerging implementation that I've personally seen in years that is actually going to help for a few reasons:

1. It's been around for a long time; it just hasn't had a title or been a focus point.
2. There's actually good need for it as it helps separate concerns between teams.
3. The engineering behind it is needed in a world where everything is always changing.

This chapter will focus on the why, how, when, and where for Platform Engineering. The goal at the end of this chapter is to help you ultimately understand, from a theoretical perspective, Platform Engineering as a whole. It will dive into:

- How Platform Engineering is viewed.
- Why you should care.
- Platform Engineering vs DevOps.
- Platform Engineering vs SRE.
- When's the right time to implement Platform Engineering.

How Platform Engineering Is Viewed

If you do extensive research on:

- Forums
- Books
- Courses
- YouTube videos
- Social media comments
- Articles

To really nail down and ultimately understand exactly how people are viewing Platform Engineering right now, it turns out that the majority of people aren't 100% sure what it is.

There's still a lot of confusion around it. The majority of the confusion comes from wondering what the differences are between DevOps, SRE, and the other avenues of engineering out there. This is pretty common with any new emerging technology or method. As something is emerging, everyone is more or less trying to figure out what it actually means. That's why we end up with multiple different definition.

The ironic thing however is that Platform Engineering isn't new. Engineers have been building self-service capabilities for years. It just hasn't been its own team or a focus point. It's more or less been something that a manager or someone from the leadership team would ask of an engineering department as a project to do on the side or when the issue became a major pain. Perhaps it was a development team or another engineering team that needed to use a system or a platform to deploy their code to, but they didn't understand the underlying system, so they needed some sort of automation, script, or "button to click" to spin up the environment so they could run their code.

Because the idea of Platform Engineering has been around for a long time, yet it hasn't really had a title or been a core focus in most organizations, it's oddly in the middle of being emerging, yet at the same time, an old practice.

I believe the best differentiation is the following:

Platform Engineering provides a focus for building internal tools (like self-service tools) to make the lives of internal engineers easier. They have a self-service method (a UI, CLI, or some automation/repeatable process) of creating the exact environment they need without getting bogged down by the details of how it's created.

It’s treating internal engineering as a product within itself.

Why We Should Care

With everything going on in the tech world from AI to DevOps to any other popular and emerging piece of the IT pie, why should you or anyone else care about Platform Engineering?

There are two reasons:

1. This feels like more or less the first time we actually care about the engineers.
2. We want to make the lives of internal engineers better.

First, let’s start off with the “caring” factor. IT and multiple tech/engineering teams have always been looked at as the teams in an organization that were spending a lot of money, asking for a lot of money, but weren’t actually helping to make money (well, they were… but they weren’t look at that way. Afterall, how can you give customers a product if it’s not deployed?). Because of that, I believe it’s pretty well known at this point that IT/tech/engineering teams weren’t exactly “cared for”. Sure, there was pizza and high fives and all of that fun stuff after working 15 hours straight to figure out a problem or to migrate an environment, but that’s about it. Platform Engineering on the other hand is built to literally care about tech teams. It’s all about relieving cognitive load, ensuring proper separation of concern, and treating tech teams as we would a customer.

But how can we do that?

By making engineers lives better, easier, and more efficient. How can we do that? Well, there are a lot of ways. Internal Developer Platforms (IDP), CLI’s, UI’s, portals, and all of that fun stuff, but it really comes down to ensuring that internal engineering/tech/developer teams have what they need without banging their head against a wall. After all, banging your head against the wall isn’t as fun as eating pizza.

The whole idea here with Platform Engineering is to literally hand engineers and developers a system/platform on a silver… no, a GOLD platter. Everything they need is there. The underlying platform, the automation, the security… all they have to do is deploy the app. For the developers and engineers consuming the system/platform that the Platform Engineering teams created, it’s a piece of ~~code~~cake. The entire system/platform is working as expected with all of the needed dependencies, and more importantly, all without the developer/engineering team having to do anything (because it's not their job to, hence the separation of concern).

So, why should you care about Platform Engineering? Because it makes the lives of everyone that doesn't need to care about the underlying platform easier. It's putting the "care" and "let me help you make your life easier" in Platform Engineering.

Platform Engineering vs DevOps

Let's break this down without getting into the whole "DevOps isn't a title" thing.

Two of the biggest questions that always pop up are:

1. What's the difference between Platform Engineering and DevOps?
2. What's the difference between Platform Engineering and SRE?

Let's answer both starting with DevOps.

The short answer is this;

Platform Engineering == Deploy systems/platforms.

DevOps == Deploy apps.

DevOps, as it stands right now, is all about getting applications and application stacks deploy to their desired location in Dev, Staging/UAT, QA, and Prod. Yeah, DevOps teams usually deploy the system/platform as well, but they're deploying it for the running environments that will contain the application they're deploying, which is typically an app/app stack for customers.

Platform Engineering number one doesn't care about the app/app stack. They aren't deploying apps, they're deploying systems, and those systems are for internal engineering/development teams. It's not about deploying systems for paying customers/clients, it's about deploying systems/platforms for internal engineering/development teams, and those internal engineering/development teams are the Platform Engineer's "customers".

Platform Engineering vs SRE

Short answer:

Platform Engineering == Deploying and managing internal systems.

SRE == Maintaining (and sometimes deploying) along with ensuring performance for external systems.

SRE’s, as it stands right now, are really focused on performance and maintenance of an existing system that’s typically being touched by an external customer/client. It’s all about ensuring that the system and the app running on the system is working as expected, or hopefully better from a performance perspective.

Platform Engineering on the other hand doesn’t care about external systems and apps. It cares about internal systems, including the performance and maintenance of those systems.

When Should You Implement Platform Engineering

As with all teams, titles, and workflows, there could be a time when you would and wouldn’t implement Platform Engineering. As with all specialties, systems, and platforms, there’s a time and a place.

There’s a funny “tug in each direction” feeling here - the answer is “you should implement it from the start”, but there’s also a “no you shouldn’t” answer, and I believe timing is the key here. If your team is so small that the product is just being developed, chances are you won’t have the budget, ability, or buy-in from the leadership team to start a Platform Engineering team.

This is the same with a product/platform that you may be implementing.

For example, I’ve worked with startups that were containerizing their environment, but they simply weren’t big enough or ready to implement Kubernetes because of the undertaking of deploying and managing Kubernetes.

The same goes for Platform Engineering.

Once the internal engineering/development teams are getting swamped with multiple requests that are pulling them in different directions and the “separation of concerns”, or rather, “should I actually be responsible for this?” questions come into play for systems/platforms is when an organization should begin to think about implementing a Platform Engineering team.

Another time to think about implementing a Platform Engineering team is when you want the ability to have a dedicated engineering department to create self-service capabilities.

Final Definition

Wrapping up Chapter 1, which hopefully gave you a good understanding of the “how” and “why” behind Platform Engineering, let’s officially define it:

Platform Engineering is the creation and management of integrated capabilities that are implemented according to what the platform user needs (the platform user is the internal engineer/developer using the platform) via self-service capabilities.

Chapter 2: Phase 1 - Understanding Business Values

When engineers first start their career, regardless of what path they take (Dev, Sysadmin, Help Desk, etc…), their primary goal is to enhance their technical skills. This makes perfect sense at the beginning of your career because without the proper tech skills, or at least interest in obtaining more tech skills, you’ll peak in whatever job role you’re in.

As you continue in your career, at some point, you’ll hit a technical ceiling. Not because there isn’t more to learn, but because after a while the tech skills only get you so far. You’ll need to start thinking about the business values.

This doesn’t mean you have to go into management or leadership, it means you have to understand the “why” behind your implementation.

Why do you want to implement Kubernetes?

Why will the cloud help your business?

The “why” behind your decision becomes more critical than the tech itself.

In this chapter, you will learn about:

- Why engineers need to care about business value.
- Business value at the leadership level.
- Business value at the team level.
- Thinking about the value vs the tech.

We’re Engineers… Why Should We Care?

There are two main reasons why engineers should care about business values:

1. Tools don’t solve problems
2. Engineers are part of the business discussion.

Tools

There are a lot of tools and platforms. The CNCF landscape in itself has over 1,500 tools, and that’s just the CNCF. There are A LOT of vendors out there that have nothing to do or are not in conjunction with the CNCF. As a whole, you’re looking at thousands and thousands of tools.

And guess what?

A lot of those tools are doing very similar things to the other tools in the same category. When you see the thousands of tools, that doesn’t mean thousands of different ways to do one thing. Think about it - how many tools, both paid and open-source, are in the monitoring and observability space?

Because of the sheer number of tools, you’ll never get a solid outcome just by thinking about the tech. You have to think about the solution. Tools don’t solve tech problems. Planning, architecture, and design solves tech problems. You just bring in a tool to accomplish the plan.

Business Discussion

Technology is evolving. There was a time in this space where there was essentially one or two vendors that did “a thing” and tech leadership chose which one to go with because, well, there weren’t a lot of options. They would read some report from an analyst firm and choose which tech to go with based on which vendor scored the highest.

At this point in time, it’s no longer that simple. As technology gets more complicated and more vendors crowd the space, a CTO can’t just go, read a report, and make a decision. These decisions have to be made at the engineering level because the engineers know how to solve the problem, so they know which tool makes the most sense.

Engineers don’t cut the checks, but they do stop the checks from being cut.

Business Value - The Leadership Level

Breaking down the previous section, let’s talk about business value at the leadership level.

At this stage, it’s not going to be about the tool or the platform that you want to use. It’s going to be the “why”. Start thinking about it like this:

1. What problem are you trying to solve?
2. Why does the problem need to be solved?
3. What will happen if the problem isn’t solved?

Managers and leadership teams rarely care about how cool the tech is. As engineers, it’s sometimes difficult to not get caught up in the new and shiny (I literally do it every day, so I get it). Instead, they’re more focused on the outcome, and guess what? They’re looking at the engineers for said outcome.

Engineers have to be able to uniquely identify a problem and understand how to solve that problem in the most efficient and effective way possible.

Leadership teams and management care about the positive outcome and the “why”.

Business Value - The Team Level

There’s no “I” in team as they say, and there’s no “I” in Platform Engineering.

When it comes to Platform Engineering as a whole, your customer/client are the internal developer and engineering teams. Because of that, the majority of your job is literally building for them, so you need to know what they need.

When you’re talking to teams, understand what they need from you to create a proper solution.

Do they want a CLI? An API?

Do they want a UI?

Do they need multiple platforms to deploy to? Just Kubernetes? Just VM’s?

What third-party tools do they need access to?

Figuring out what they need to do to get their job done in an efficient manner is your job to figure out.

Think Value, Not Tool

As mentioned in the previous sections, throwing products at a problem won't help the problem. It'll most likely just create more technical debt. Instead, as you've learned throughout this chapter, figure out what problem you want to solve. What solution are you trying to get to? Why are you trying to get to that solution?

Once you begin thinking about the value you need to create and provide instead of what tool you want to use, the solution (and the tool) will become far more obvious.

Chapter 3: Phase 2 - Teamwork

As you've learned about so far throughout this eBook, the entire job of a Platform Engineer is to create tools for internal use. Platform Engineering isn't a public facing role. It's not a role where you're working with external customers, deploying external apps, or architecting external systems. Everything you do will be for the better of internal development and engineering teams.

Your customers/clients are now the internal teams.

Because of the nature of a role like this, there are two parts; the technical part and the team part. Although you may not be creating tools for the people on your direct team, you're creating tools for people in other teams within your organization, which means they're very-much part of your "team".

In this chapter, you'll learn about:

- Figuring out who's going to use the platform.
- How they're going to use the platform.
- Working with developers and other teams.

Who's Using It?

Working With Developers

Working With Other Teams

Chapter 4: Phase 3 - Standardization

When you're creating any type of system, platform, tool, deployment method, or just about anything else in tech, it requires some sort of standardization. Otherwise, everyone is more or less using different tools than their peers and creating a massive amount of tech debt.

Although standardization may not always feel fun, it's necessary.

> 💡 I love dabbling in new tools as much as the next engineer. In fact, it's a big part of my self-employment. I'm definitely not saying you shouldn't do this. You should be interested in the tech as it's a lot of fun. I'm just saying a business should have particular standards.

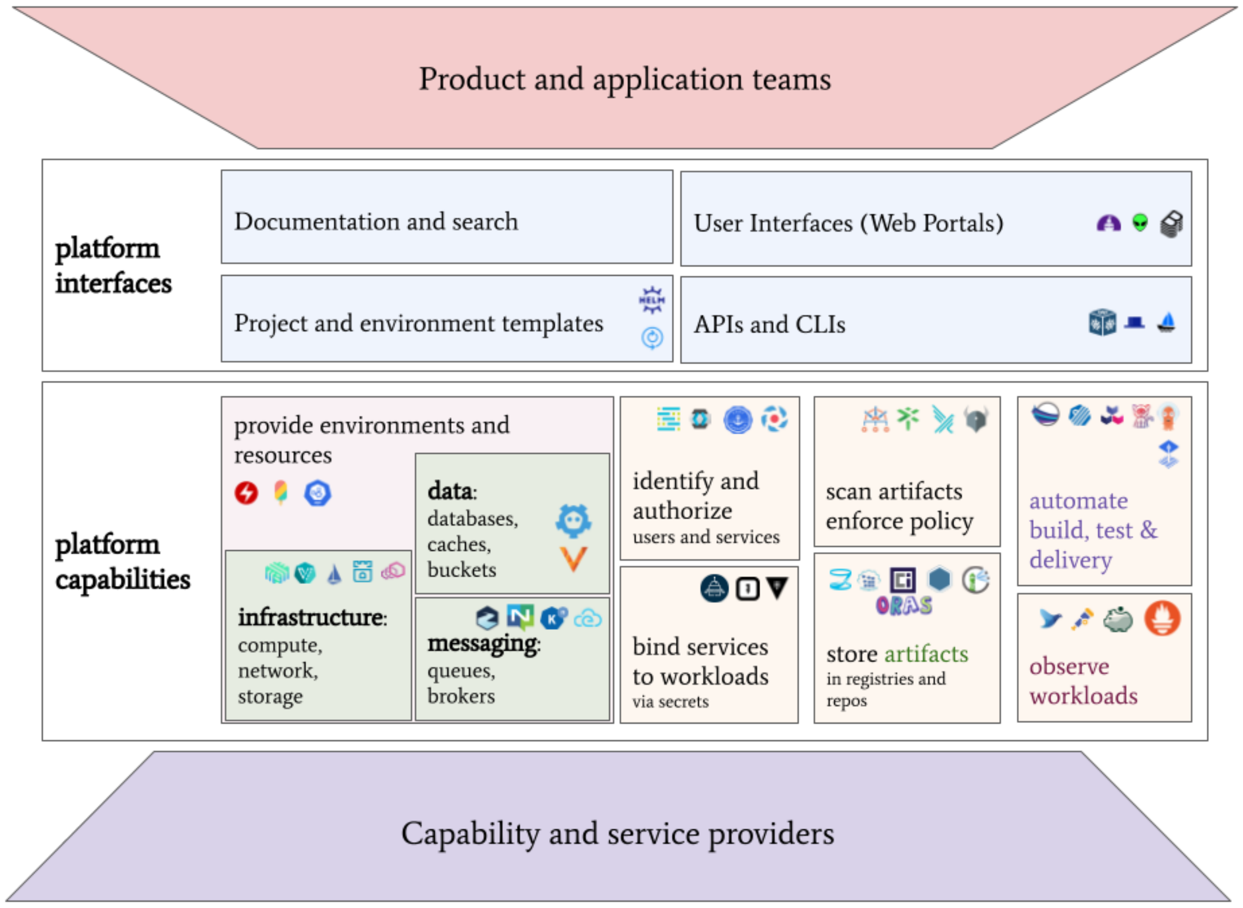
Standards come in all shapes and sizes. For example, how you build a container image or how you deploy Kubernetes. If you have three engineers and they're all using three different methods of deploying a Kubernetes cluster in production, there's bound to be a massive amount of confusion, frustration, technical debt, and downtime. It's why engineers have created standards around tools like CICD.

With Platform Engineering, it's no different. You're going to want to create a standardization around the stack your organization is using, but more importantly, what's going into the stack.

One major benefit that standards give us is the ability to build and use tools in an efficient manner. The more tools you throw into the mix, the more cumbersome the environment gets. Engineers are confused, developers are confused, and no one knows what to use. In an organization that's deploying production-level workloads, the number one way to ensure that the deployments don't go as expected is by not introducing a standard.

This chapter will go over the following:

- Choosing the underlying platform that works best for you.
- Platform capabilities.
- Platform interaction.


Product and application teams
platform interfaces
Documentation and search
User Interfaces (Web Portals)
Project and environment templates
APIs and CLIs
platform capabilities
provide environments and resources
data: databases, caches, buckets
infrastructure: compute, network, storage
messaging: queues, brokers
identify and authorize users and services
bind services to workloads via secrets
scan artifacts enforce policy
store artifacts in registries and repos
automate build, test & delivery
observe workloads
Capability and service providers

Source: https://tag-app-delivery.cncf.io/whitepapers/platforms/

Who’s Doing What?

There are multiple people from various parts of the organization involved in the Platform Engineering process, but let’s break it down into two groups:

1. The Platform Engineers themselves.

2. The engineers/developers using what was built by the Platform Engineers.

This example will be about developers/engineers using a self-service portal.

The Platform Engineers are who build the platform and all it encompasses. For example, let’s say you’re building a self-service portal that has an option to create an environment which consists of Resource Groups, a Kubernetes cluster, security permissions, and Key Vault in Azure. You’re responsible for everything from the self-service portal to what the self-service portal is using to create the environment. It could be a bunch of scripts making API calls or a CICD pipeline building and running code to create the environment.

The engineers/developers using what was built by the Platform Engineers are simply using the self-service portal to get the environment up and running. They don’t know about or care about the CICD pipeline or the code. All they know is they log into a portal, they choose some options, and poof, a new environment is prepared.

Regardless of if it’s a portal, UI, CLI, API, or whatever else, that’s the way the engineer/developer sees it. The platform is prepared. They’re just consuming it.

The standard around “who’s doing what” is this; Platform Engineers build and maintain. Developers/engineers outside of the Platform Engineering team use/consume.

As a Platform Engineer, you must be well-versed in everything from the infrastructure to the services to the code to the API’s and everything in-between.

The Underlying Platform

The first step is to understand what underlying platform you want to use.

Kubernetes?

VM's?

Cloud-based services?

Regardless of what the underlying platform is, you need to know what it’s going to be prior to building any capabilities. The reason why is because the capabilities that you’re providing will differ based on the underlying architecture. For example, let’s say you decide that one of the capabilities will be Grafana. With Grafana, you can use something called Kube-Prometheus, which in short, is a bunch of Kubernetes-centric dashboards out of the box for Grafana. This is almost a standard if you’re using Grafana and Kubernetes. However, if your underlying platform is cloud-based VM’s in Azure or EC2 instances in AWS, you wouldn’t need to offer Kube-Prometheus as a solution.

The underlying platform is one of the major components that the Platform Engineer will be very privy to, but the user (engineer/developer) of the platform will not. It’s very much like a cloud-based environment. We use Azure services, AWS services, etc…, which are running on servers, we don’t know anything about those servers. Are they Dell? HP? How many? How are they stacked? We don’t know because we don’t care and we don’t have to care because it’s out of our control. That’s how a user (engineer/developer) of the platform you’re building should feel. They shouldn’t care what it is because it doesn’t matter to them.

Two of the most important things when you’re building the underlying platform and making a standard around it is:

1. Do the Platform Engineers know and understand it?
2. Is it the platform that the consumers need?

If your Platform Engineers know VM’s, but they don’t know Kubernetes, chances are building a platform out of Kubernetes won’t go so well. Before the build process occurs, the Platform Engineers will need to study, lab, learn, and deploy Kubernetes so they can understand it inside and out. Afterall, you can’t build a platform on a platform that you don’t know.

On the flipside, what if the engineers/developers using the platform aren’t deploying containerized apps? Well, in that case, chances are you shouldn’t be deploying Kubernetes as the underlying platform then. Remember, it all comes down to what the engineer/developer needs from the platform.

Platform Capabilities

Once the platform itself is decided upon, the next thing to think about is what capabilities the platform needs. This is where meeting with the teams that will use the platform makes sense. You have to get feedback from them to see what they need.

Do they need monitoring?

Do they need logging?

Do they need to install third-party tools like ArgoCD, Crossplane, or something else specifically for what they’re deploying?

The capabilities of the platform are what you will make public to the developer/engineer using the platform. What exactly can they install and use on the platform?

Creating these platform capabilities will also come down to a standard that's being set. Just as an example, if everyone is using one GitOps Controller and that's what the team has decided on, but there's one developer/engineer that wants to use another GitOps Controller, a conversation needs to be had. Why? Because unless there's a compelling reason, you don't want to muddy up the waters by giving too many tools to use. If you do that, it's just going to be as bad as it currently is - tool madness and technical debt all over the place.

Capabilities are what the platform needs the ability to do for a specific purpose based on developer/engineer needs.

💡 Regardless of the capabilities, keep this in mind - your job as a Platform Engineer is to make the capability a developer/engineer needs to use more efficient, easier, and straightforward. You implement the automation and practices to get this done while they use the finished product.

The long and short of it is the platform capabilities are what the developers/engineers need to get their job done, and they need it delivered to them in an automated fashion. Nothing more, nothing less.

Something that'll help you begin to evaluate capabilities is by splitting them into groups. For example, the platform may need:

- Monitoring
- Observability
- SDK access
- The ability to create resources via Kubernetes with something like Crossplane
- Messaging

Within each of those groups/categories are several tools. For example, in monitoring there's Grafana, Datadog, App Dynamics, and several others. Creating groups/categories will make choosing the tool easier.

Platform Interaction

Last but certainly not least is the overall platform interaction. What this comes down to is how the developers/engineers are interacting with the platform.

Do they need a CLI?

Do they need an API?

Do they need a UI?

The question is this: How will the developers/engineers interact with the platform you've built?

The typical answer to this question, as of right now, seems to be Internal Developer Platforms (IDP), which you can think of like a self-service portal. In chapter 6, you're going to be diving into IDP's in terms of building them out, so let's not go too deep into that discussion here.

Platforms for engineers/developers should be D) All of the above. There should be capabilities that provide a CLI, a UI, and an API. Think about it - all products in today's world typically contain all three (unless it's a CLI-based tool), so why should your internal tool (product) be any different? The interaction with a platform you're creating must be capable of what an engineer/developer needs to fully do their job to the best of their ability with the tools (capabilities) they need. It's the whole reason you'd want to create a platform for them to use.

Although IDP's seem to be the way forward right now, I do have some opinions that I'd like to share with you. Not because they're definitely going to happen or not, but it's certainly something raddling through my mind. There was a time where GUI's were far more popular than automation, and then GUI's got a "bad name" because it was all "clicky clicky" and we couldn't move fast that way. It was also incredibly cumbersome to do the same thing over and over again, so repeatability came into play. Now, we're almost… going back to the GUI? And Platform Engineers are building a GUI that others can use. Now, I know IDP's are way different than some GUI that you have to click "next next next" through. I know the purpose is different and I know it's better. However, I am curious to see if we go back to the same mindset of "we have to get out of a GUI". Again, nothing set in stone here and it's not even an opinion… it's just more of a thought.

Who’s Responsible for The Environment

As you’re thinking about the overall standardization within a Platform Engineering environment, this question may come up - who manages the environment once it’s created?

Here’s an example: The Platform Engineering team gives the engineer/developer a method of creating an environment with all of the capabilities they need. There’s now a new environment running. Who manages it?

The answer is a little bit of both. Let’s break it down into groups.

Something goes wrong from a technical perspective:

An upgrade is needed:

An update is needed:

A new environment needs to be built:

The environment needs to be deleted:

Overall maintenance:

Security:

Overall, the responsibility of the environment underneath the hood is the job of the Platform Engineer. Anything that can be created as an automated process that can be done with something like a “button click” is done by the developer/engineer.

Wrapping Up

This chapter was created to specify some ways to think about standards, but chapter 5 and 6 will be used to actually build a stack. This chapter was more a theoretical approach on how to think about Platform Engineering from a standardization perspective.

Chapter 5: Phase 4 - Building the Stack

TBD. Actively being worked on. Once complete, update will be sent containing the content.

Stack 1: Kubernetes

AKS

EKS

Stack 2: Virtual Machines

AWS

Azure

Chapter 6: Phase 5 - Building Self-Service

TBD. Actively being worked on. Once complete, update will be sent containing the content.

Internal Developer Platforms and Self-Service

Chapter 7: Phase 6 - Cognitive Load Reduction

TBD. Actively being worked on. Once complete, update will be sent containing the content.

This chapter will cover:

- The ability to reduce cognitive load
- Reduce complexity
- A brief recap of IDPs

Reducing Overall Complexity

Self-Service Portals and IDP’s

Developer Experience

There has been an emerging talking point very recently called Developer Experience, or DevEx for short. The whole goal is, as it sounds, to provide developers (and I’m assuming other engineers) the best possible experience.

It’s all about making the use of a platform/system as easy as possible.

Now, where the confusion currently is (and why I believe this will be lumped into Platform Engineering) resides in the fact that making the experience as good as possible for working with platforms/systems is exactly what Platform Engineering is trying to accomplish.

Because it’s still emerging, the definition and outcomes may change, but as it sits right now, it very-much just feels like a subset of Platform Engineering.