

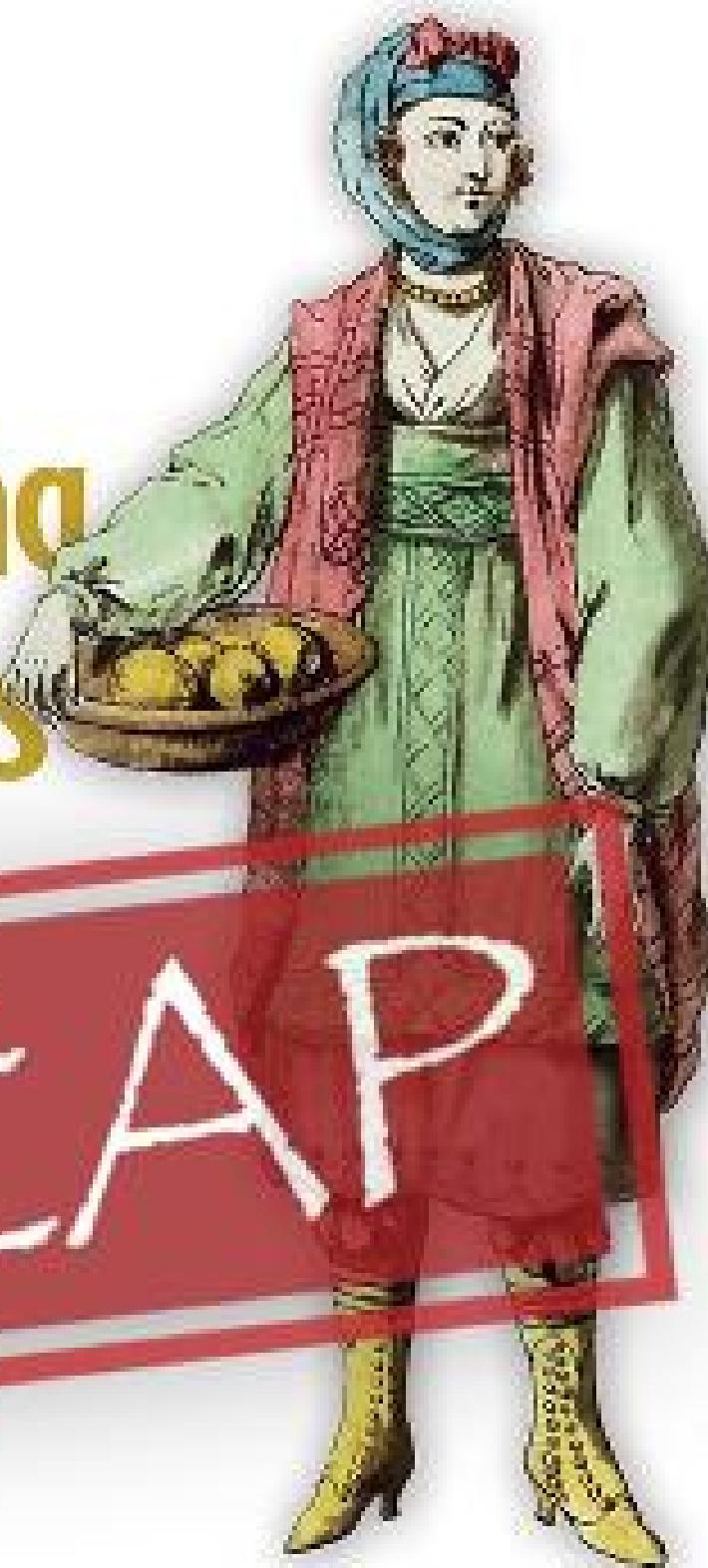


HANNING

Platform Engineering Kubernetes^{on}

MEAP

Mauricio Salatino



MANNING

Platform Engineering on Kubernetes

MEAP

Mauricio Salatino



Platform Engineering on Kubernetes MEAP V09

1. [Copyright 2022 Manning Publications](#)
2. [welcome](#)
3. [1_\(The_rise_of\)Platforms_on_top_of_Kubernetes](#)
4. [2_Cloud-native_applications_challenges](#)
5. [3_Service_pipelines:_Building_cloud-native_applications](#)
6. [4_Environment_pipelines:_Deploying_cloud-native_applications](#)
7. [5_Multi-cloud_\(app\)_infrastructure](#)
8. [6_Let's_build_a_platform_on_top_of_Kubernetes](#)
9. [7_Platform_capabilities_I:_Enabling_developers_to_experiment](#)
10. [8_Platform_capabilities_II:_Shared_application_concerns](#)
11. [9_Measuring_your_platforms](#)



MEAP Edition

Manning Early Access Program

Platform Engineering on Kubernetes

Version 9

Copyright 2022 Manning Publications

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes.

These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://livebook.manning.com/book/platform-engineering-on-kubernetes/discussion>

For more information on this and other Manning titles go to
manning.com

welcome

Thanks for purchasing the MEAP for *Platform Engineering on Kubernetes*. To get the most out of this book, you'll need to feel comfortable with the basics of Kubernetes and Containers. In this book, you will learn how to design and build platforms to enable your application teams to deliver more software efficiently.

To build these platforms, we will look into different open-source projects designed natively for Kubernetes to tackle common challenges you will face when building, architecting, and delivering cloud-native applications. Examples in this book are based on Java and Go, but the book does not focus on any specific programming language, and teaching is transferable between languages.

When I started working with Kubernetes, it was a new container orchestration system, and the entire community was learning the ropes. At the time, there weren't many tools available to build, deploy and monitor cloud-native applications; users needed to do a lot of tasks by hand or create custom scripts.

Nowadays, the ecosystem is vibrant, and choosing from a myriad of tools available is hard and time-consuming - staying up to date is a full-time job. This book shows how some of these tools can be combined to organize the project that works for your organization no matter in which stage of the Kubernetes journey you are.

If you were following along with the MEAP program, you might have noticed that I've changed the book title from "Continuous Delivery on Kubernetes" to "*Platform Engineering on Kubernetes*". There are several reasons behind this decision, and you can read more about why in this blog post: <https://www.salaboy.com/2023/03/31/what-platform-engineering-why-continuous-delivery/> . In short, when I started writing the book, the term Platform Engineering wasn't being used in the Cloud-Native space. Today, however, the term has gained popularity and assists in explaining how to

organize efforts to help teams to get the tools and capabilities that they need to be efficient at producing and delivering software.

The tools covered in this book have been chosen based on their adoption, their relevance in the Kubernetes community, and their way of solving particular challenges that are inherent to the Cloud. This uses Kubernetes as the base layer, which users can rely on, and none of the tools included in the book relies on Cloud-Provider-specific services, which in turn makes this book a good guide for planning and implementing platforms that can span across multiple Cloud Providers.

If at any point you have questions or suggestions, or you want to read more about these topics, I recommend you to reach out via my blog <https://www.salaboy.com> or Twitter @Salaboy as my DMs are open. I've written all the chapters in collaboration with amazing people coming from each project's community, so if you feel passionate about any of these projects and you want to join as an open-source contributor, drop me a message, and I will be more than happy to connect you with the right people to get started.

Please post any questions, comments, or suggestions you have about the text in the book in the [liveBook discussion forum](#). Your feedback is essential in developing the best book possible.

- Mauricio “@Salaboy” Salatino

In this book

[Copyright 2022 Manning Publications welcome](#) [brief contents](#) [1 \(The rise of\) Platforms on top of Kubernetes](#) [2 Cloud-native applications challenges](#) [3 Service pipelines: Building cloud-native applications](#) [4 Environment pipelines: Deploying cloud-native applications](#) [5 Multi-cloud \(app\) infrastructure](#) [6 Let’s build a platform on top of Kubernetes](#) [7 Platform capabilities I: Enabling developers to experiment](#) [8 Platform capabilities II: Shared application concerns](#) [9 Measuring your platforms](#)

1 (The rise of)Platforms on top of Kubernetes

This chapter covers

- What are Platforms, and why do we need them?
- What does it mean to build a platform on top of Kubernetes
- Introduces a “walking skeleton” application that we will use throughout the book

Platform engineering is not a new term in the tech industry. But it is quite new in the Cloud-Native space and in the context of Kubernetes. We were not using the term in the Cloud Native communities when I started writing this book back in 2020. However, platform engineering is the new hot topic in Cloud-Native and Kubernetes communities today. This book aims to go on a journey to explore what platforms are and why you would use Kubernetes, and more specifically, the Kubernetes APIs at the core, to build a platform and enable your internal teams to deliver software more efficiently.

To understand why Platform Engineering became a trend in the industry, you first need to understand the Cloud Native and Kubernetes ecosystems. Because this book assumes that you are already familiar with Kubernetes, containers, and Cloud Native applications, we will focus on describing the challenges you will face when architecting, building, and running these applications on top of Kubernetes and Cloud Providers. We will take a developer-focused approach, meaning that most of the topics covered are tackled in a way that relates to developers' day-to-day tasks and how a myriad of tools and frameworks in the cloud native space will impact them.

The ultimate goal for every software development team is to deliver new features and bug/security fixes to their customers. To deliver more software efficiently, development teams need to access the tools to do their work on demand. The main objective of the Platform and Platform Engineering teams is to enable developers to deliver software more efficiently. This requires a

different technological approach and a cultural shift towards treating development teams as internal customers of the platforms we will be building.

We will use a simple application (composed of multiple services) as an example throughout the chapters to build a platform around it using all Open Source and tools in the Cloud Native space.

1.1 What is a Platform, and why do I need one?

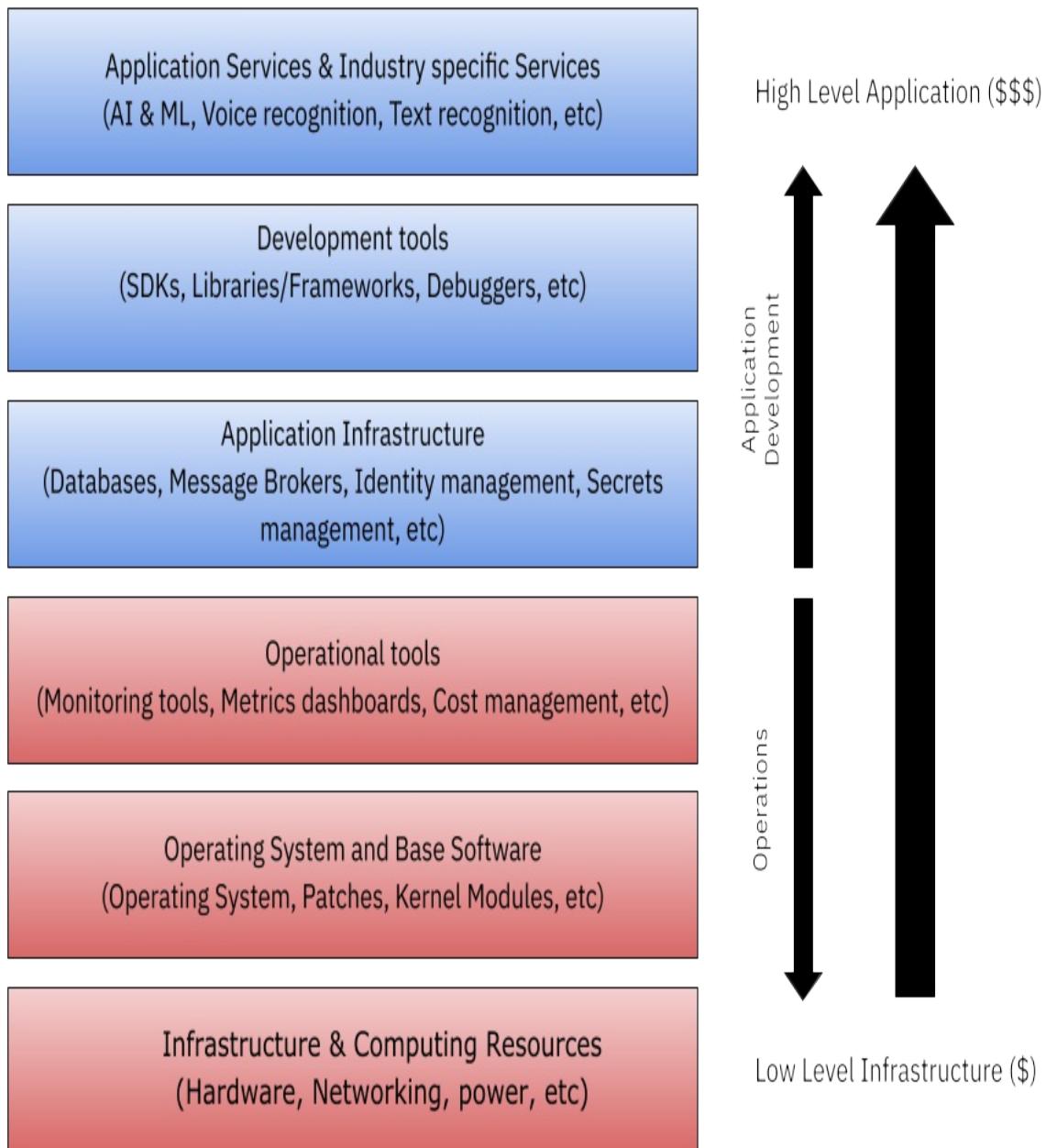
Platforms are not new, and neither are Cloud Platforms. Cloud providers like AWS, Google, Microsoft, Alibaba, and IBM have provided us with platforms for years. These Cloud providers offer teams many tools to build, run and monitor their business-critical applications using a pay-as-you-go model. This allows companies and teams to start fast and create applications that can scale globally without a significant initial investment. If no one uses the applications they are building, their bills will not be large at the end of the month. On the other side of the spectrum, if you are successful and your applications are popular, you need to get ready for a large bill at the end of the month. The more resources (storage, network traffic, services, etc.) you use, the more you pay. Another aspect to consider is that if you rely on the tools provided by your Cloud Provider, it is harder to move away from it as your entire organization gets used to Cloud Provider tools, workflows, and services. It becomes a painful experience to plan and migrate applications across different providers.

One thing is clear, Platforms are a collection of services that help companies to get their software running in front of their customers (internal or external). Platforms aim to be a one-stop shop for teams to have all the tools that they need to be productive and continuously deliver business value—with the rise in popularity and with the growing demand to improve development cycles, platforms that once used to provide us only with computing resources had leveled up the stack to provide more and more services.

We can divide Cloud Services into different layers, something that we need to do to understand where the industry is today and where it is heading. The following diagram shows a set of categories of the services provided by

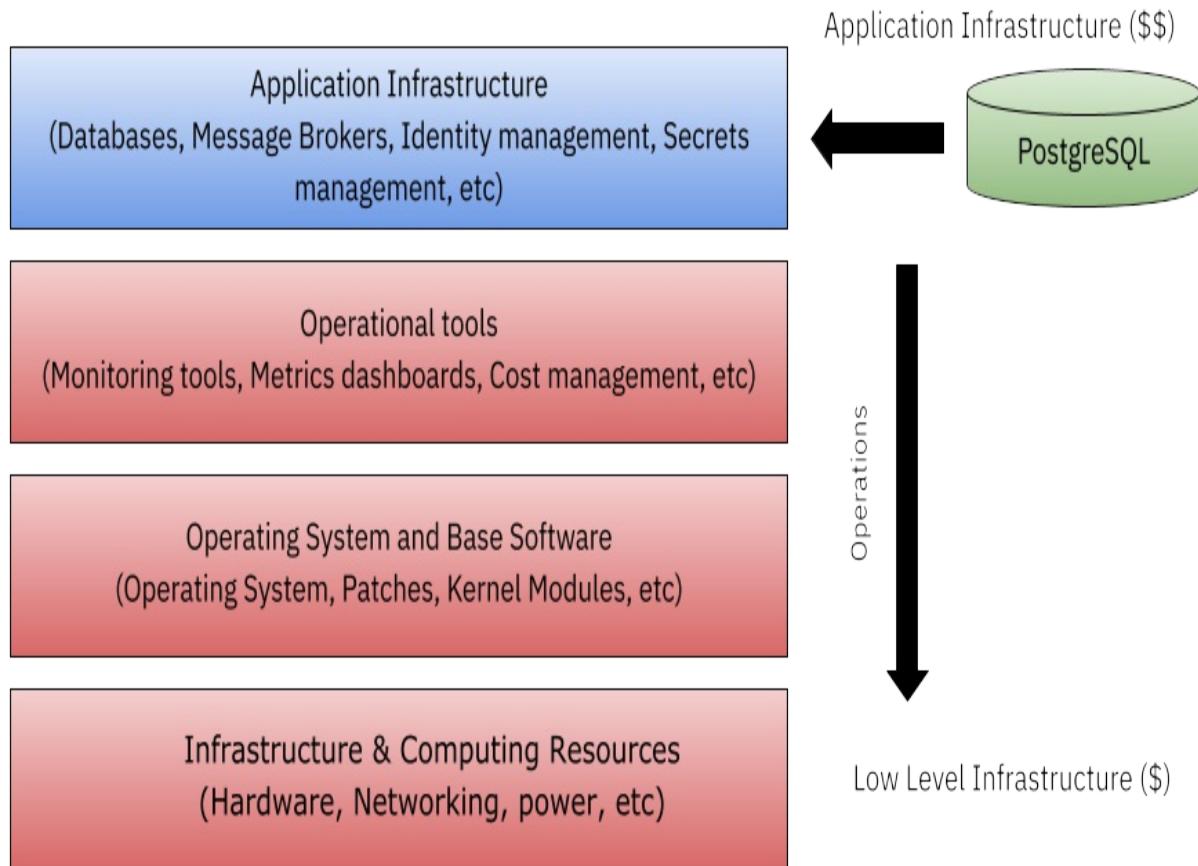
Cloud Providers, starting from low-level infrastructure services, such as provisioning hardware on demand to high-level application services, where developers can interact with Machine Learning models without worrying where these models are running.

Figure 1.1 Cloud Provider's services categories



The higher the category, the more you will need to pay for the service, as these services usually take care of all the underlying layers and operational costs for you. For example, suppose you provision a new Highly Available PostgreSQL database in a managed service offered by a Cloud Provider.

Figure 1.2 Provisioning a PostgreSQL database instance in the cloud



In that case, the service cost includes the cost and management of the database software needed, the Operating System where the database runs, and the hardware needed to run it. Because you might want to monitor and get metrics on how the database performs when your application is under heavy load, the Cloud Provider also wire up all the monitoring tools available for the service. Then it is up to you to do the math: is it worth paying a Cloud Provider to make all these decisions for us, or can you build an internal team with enough knowledge to run and operate all these software and hardware on-premises?

Keeping up to date with all the provided services, libraries, frameworks, and tools is a full-time job. Operating and maintaining the wide range of software and hardware that companies need to run their applications requires you to have the right teams in place, and at the end of the day, if you are not a large and mature organization on software delivery practices, adopting a Cloud Provider is usually the right way to go.

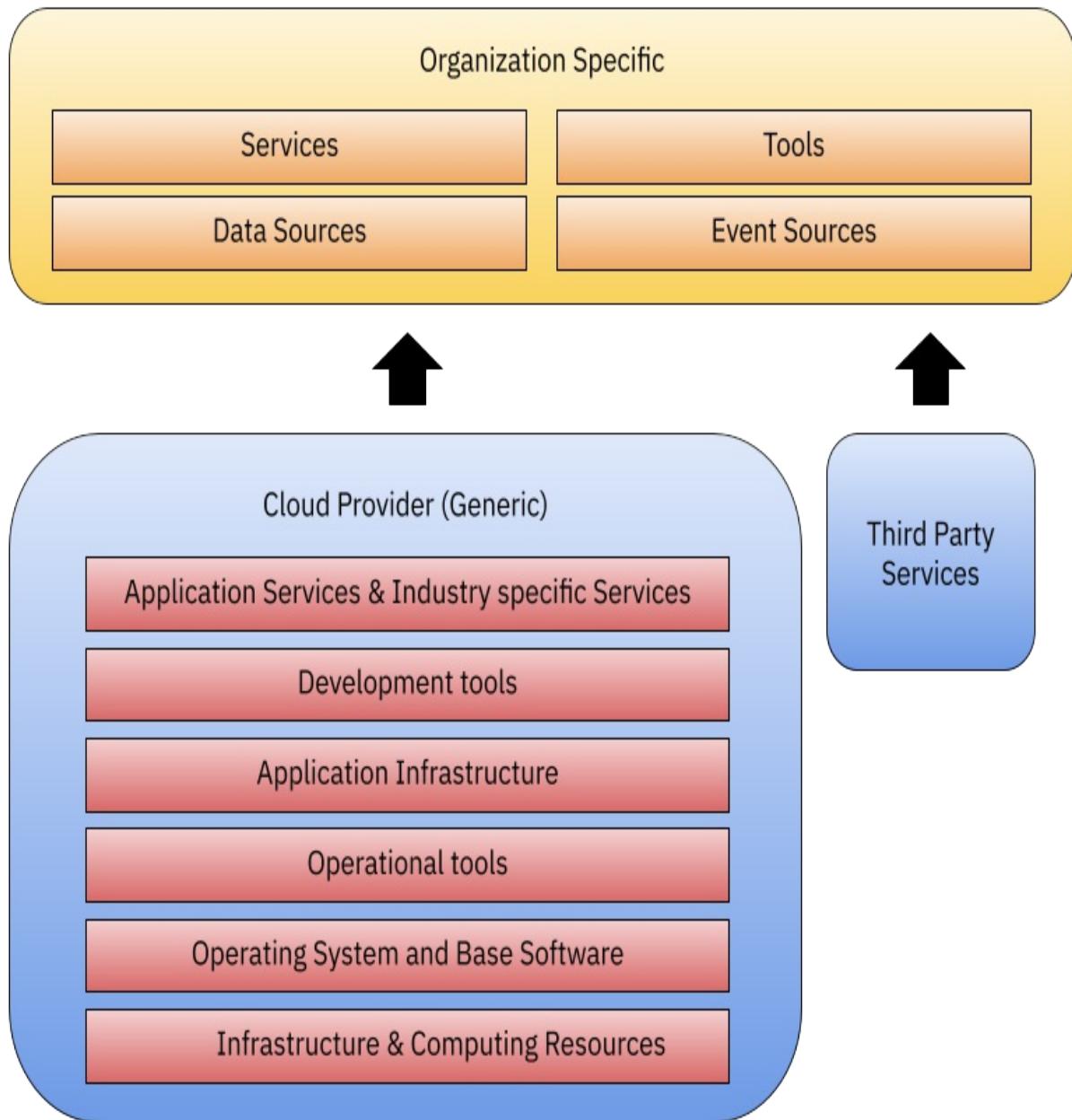
It is still the job of each company and developer to look at the available services and choose what they are going to use, and how they are going to mix and match these services to build new features. It is common to find Cloud Architects (experts on a specific Cloud Provider) in each organization defining how and which services will be used to build core applications. It is also common to engage with the Cloud Provider consulting services to get advice and guidance on specific use cases and best practices.

Cloud Providers might suggest tools and workflows to create applications. Still, each organization needs to go through a learning curve and mature its practices around applying these tools to solve its specific challenges. Staffing Cloud Provider experts is always a good idea, as they bring knowledge from previous experiences, saving time for less experienced teams.

In this book, we will focus on organization-specific platforms, not generic Cloud Platforms that you can buy off the shelf, like those offered by Cloud Providers. We also want to focus on Platforms that can work On-Premises on our organization's hardware. This is important for more regulated industries that cannot run on Public Clouds. This forces us to have a broader view of tools, standards, and workflows that can be applied outside the realm of a Cloud Provider. Consuming services from more than one Cloud Provider is also becoming increasingly popular. This can result from working for a company that acquired or became acquired by another company using a different provider, ending up in a situation where multiple providers must coexist, and there should be a shared strategy for both.

The kind of platforms we will be looking at extends with custom behavior the layers mentioned before to include company-specific services that allow the organization's development teams to build complex systems for the organization and their customers.

Figure 1.3 Organization-specific layers



These extra layers are, most of the time, “glue” between existing services, data, and event sources combined to solve particular challenges the business faces or to implement new features for their customers. It is common to rely on third-party service providers for more business-specific tools, for example, industry-specific or generic CRM (Customer Relationship Management) systems, such as Salesforce.

For the customers, the Platform, the Cloud Provider, and where the services are running are entirely irrelevant. Internally, for development teams, the Platform acts as an enabler for development teams to do their work.

Platforms are not static, and their main objective is to help the organization improve and excel at continuously delivering high-quality software in front of its customers.

Technically, platforms are all about system integrations, best practices, and composable services that we can combine to build more complex systems. No matter the industry where your company operates and whether you choose to use a Cloud Provider, your company's combinations of tools and workflows to deliver new features to its customers can be described as your platform. This book will look at standard practices, tools, and behaviors that make platforms successful and how you can build your platforms for the cloud, whether running on one or more Cloud Providers or On-Premises.

We will use Cloud Providers as a reference to compare the services and tools they provide and how we can achieve similar results in a multi-cloud provider and On-premises way by using Open Source tools. But before looking into concrete tools, it is essential to understand what kind of experiences we can get from Cloud Providers.

1.1.1 Three key features of a platform

One common denominator between all Cloud Providers is that they provide services using an API-first approach (API being Application Programming Interface). This means that to access any of their services, an API will be available to the users to request and interact with this service. These APIs expose all the service functionality, such as which resources can be created, with which configuration parameters, where (in which region of the world) we want the resource to run, etc.

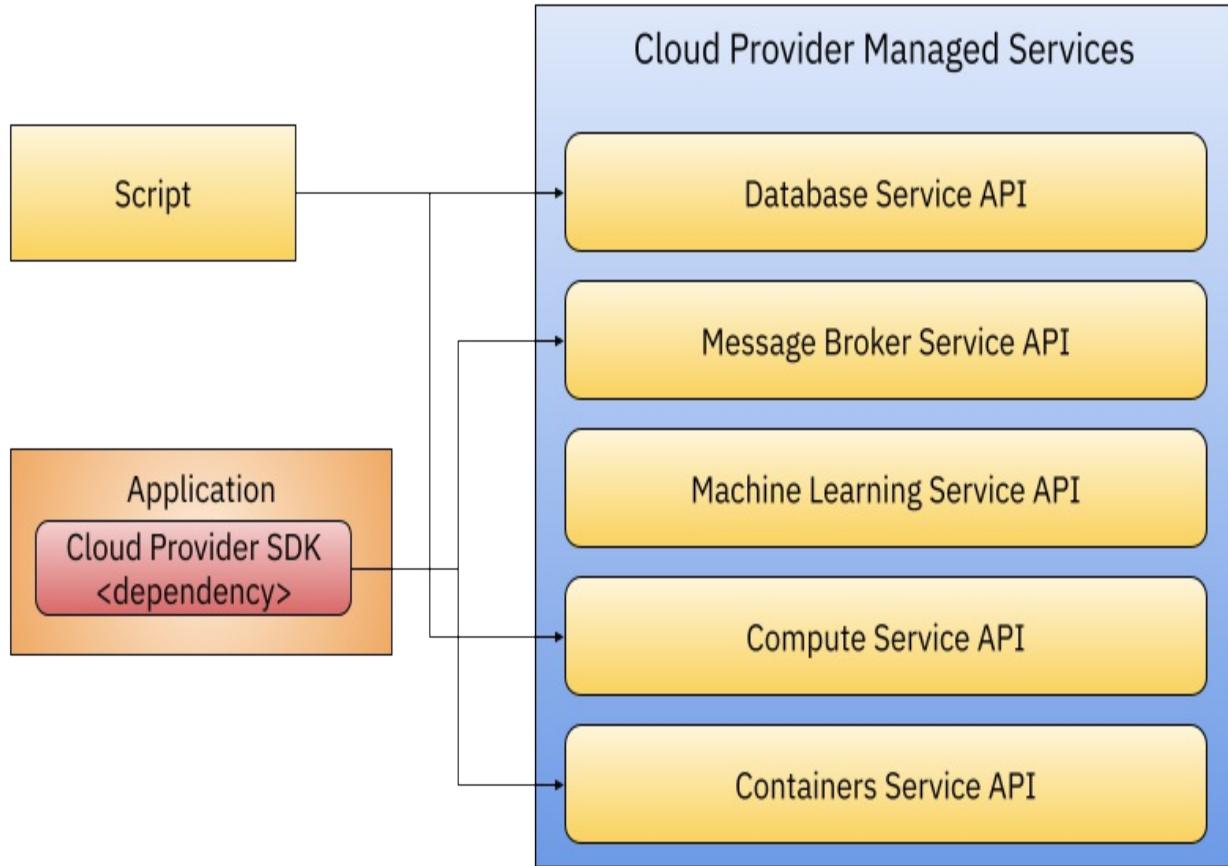
Each Cloud Provider can be analyzed by looking at their APIs, as there will usually be one API for each service being offered. It is common to see services in the beta or alpha stage only offered through the APIs for early users to experiment, test and provide feedback before the service is officially announced. While for all the services APIs, the structure, format, and style

tend to be similar for all the services provided by a Cloud Provider, there are no standards across Cloud Providers to define how these services should be exposed and which features they need to support.

Manually crafting complex requests against the Cloud Provider services APIs tends to be complex and error-prone. It is a common practice by Cloud Providers to simplify the developer's life by providing SDKs (Software Development Kits) that consume the services APIs implemented in different programming languages. This means developers can programmatically connect and use Cloud Provider's services by including a dependency (library, the Cloud Provider SDK) to their applications. While this is handy, it introduces some strong dependencies between the application's code and the cloud provider, sometimes requiring us to release our application code to upgrade these dependencies.

Figure 1.4 shows different Cloud Provider services and how consumers, in this case, a Script and an Application that depend on the Cloud Provider SDK, consume the available services APIs.

Figure 1.4 Cloud Provider's API-first clients (APIs and SDKs)

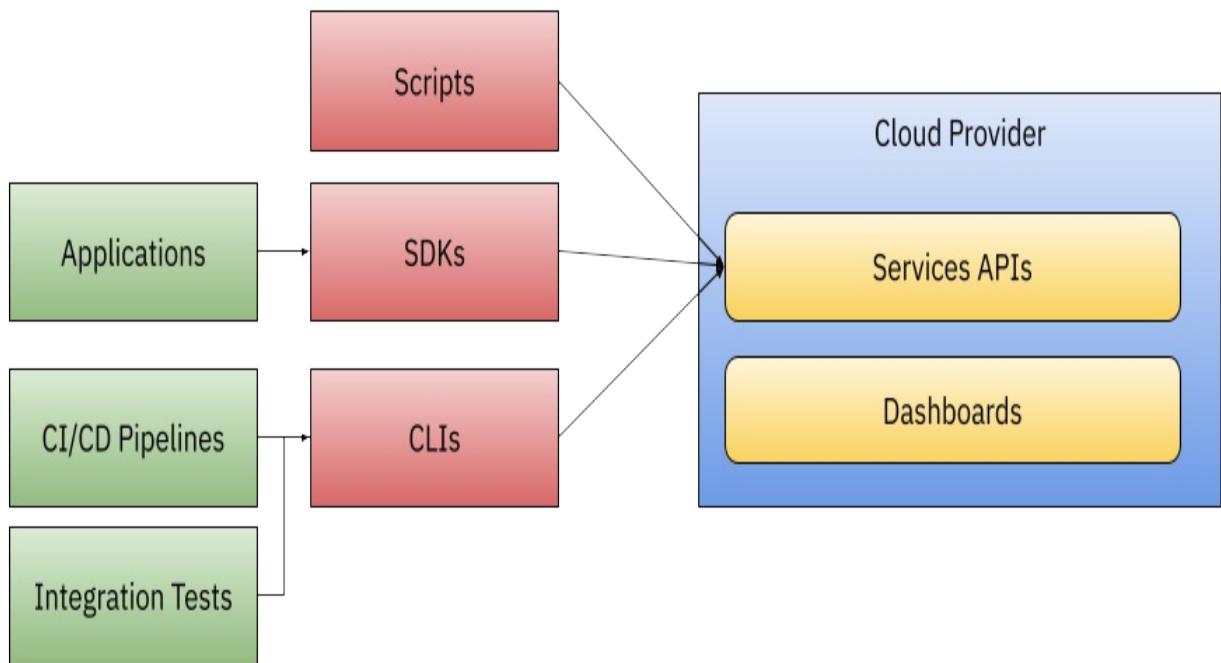


In the same way that with APIs, with SDKs, there are no standards, and each SDK heavily depends on the programming language ecosystem's best practices and tools. There are cases where the SDKs don't play nice with frameworks or tools that are popular in the programming language that you are using. In such cases, going directly to the API is possible but hard and usually discouraged, as your teams will maintain all the code required to connect to the Cloud Provider services.

Cloud Providers also provide CLIs (Command-line interfaces) tools for operations teams and some developers' workflows. CLIs are binaries you can download, install, and use from your operating system terminal. CLIs interact directly with the Cloud Provider APIs but don't require you to know how to create a new application to interact with the services as with SDKs. CLIs are particularly useful for Continuous Integration and automation pipelines, where resources might need to be created on demand, for example, to run our integration tests.

Figure 1.5 shows Scripts, Applications, and Automation tools all consuming the same APIs but using different tools designed by the Cloud Provider specifically to simplify these scenarios. The figure also shows the Dashboard component, usually running inside the Cloud Provider, which provides visual access to all the services and resources that are being created.

Figure 1.5 Cloud Provider's SDKs, CLIs, and Dashboards clients



Finally, due to the number of services provided and the interconnections between the services, Cloud Providers offer Dashboards and User Interfaces to access and interact with all the offered services. These dashboards also offer reporting, billing, and other functions that are hard to visualize using the CLIs or directly via the APIs. By using these Dashboards, users can access most of the standard features provided by the services and real-time access to see what is being created inside the Cloud Provider.

As mentioned, Dashboards, CLIs, and SDKs require your teams to learn about many Cloud Provider-specific flows, tools, and nomenclature. Because of the number of services provided by each Cloud Provider, it is no wonder why finding experts that can cover more than a single provider is challenging.

Because this is a Kubernetes-focused book, I wanted to show the experience

provided by a Cloud Provider to create a Kubernetes Cluster, which demonstrates the Dashboard, CLI, and API exposed by the Google Cloud Platform. Some Cloud Providers provide a better experience than others, but overall you should be able to achieve the same with all the major ones.

1.1.2 GCP Dashboard, CLIs and APIs

Look at the Google Kubernetes Engine dashboard to create new Kubernetes clusters. As soon as you hit Create a new Cluster, you will be presented with a form asking you to fill in a few required fields, such as the name of the cluster.

Figure 1.6 Google Kubernetes Engine creation form

≡ Google Cloud DevTest ▾ Search (/) for resources, docs, products, and more Q Search 🗑️ 🖼️ 6 🎉 ? :

← Create a Kube... ADD NODE POOL REMOVE NODE POOL USE A SETUP GUIDE ▾ SWITCH TO AUTOPILOT CLUSTER HELP ASSISTANT LEARN

Cluster basics

NODE POOLS

default-pool

CLUSTER

Automation

Networking

Security

Metadata

Features

Cluster basics

The new cluster will be created with the name, version, and in the location you specify here. After the cluster is created, name and location can't be changed.

To experiment with an affordable cluster, try My first cluster in the Cluster set-up guides

Name

Cluster names must start with a lowercase letter followed by up to 39 lowercase letters, numbers, or hyphens. They can't end with a hyphen. You cannot change the cluster's name once it's created.

Location type

Resource prices may vary between certain regions. [Learn more](#)

Zonal
 Regional

Zone ▾

Specify default node locations

Increase availability by selecting more than one zone
Current default: us-central1-c

CREATE CANCEL Equivalent: [REST](#) or [COMMAND LINE](#)

Cloud Providers do a fantastic job at having sensible defaults to avoid asking you to fill in 200 parameters before creating the needed resource. Once you have filled in all the required fields, the form offers a quick way to start the provisioning process by just hitting the “Create” button down the bottom. It is pretty interesting to see that, in this case, the Google Cloud platform offers you an estimated cost per hour of the resource that you have configured. You can start tweaking parameters to see how this cost changes (usually, it goes up).

Figure 1.7 Create via a dashboard, REST, or using a Command Line Interface (CLI) tool



Right beside the Create Button, you can see the REST API option. The Cloud Provider here helps you by crafting the REST request to their APIs needed to create the resource you can configure using the forms. This is quite handy if you don't want to spend hours looking at their API documents to find the shape of the payload and properties needed to create the request.

Figure 1.8 Create via Kubernetes cluster using a REST request (notice side scroll)

Equivalent REST request

This is the REST request with the parameters you have selected. [REST API reference](#)

```
POST https://container.googleapis.com/v1beta1/projects/strong-harbor-338418/zones/us-central1
{
  "cluster": {
    "name": "cluster-1",
    "masterAuth": {
      "clientCertificateConfig": {}
    },
    "network": "projects/strong-harbor-338418/global/networks/salaboy-vpc",
    "addonsConfig": {
      "httpLoadBalancing": {},
      "horizontalPodAutoscaling": {},
      "kubernetesDashboard": {
        "disabled": true
      },
      "dnsCacheConfig": {},
      "gcePersistentDiskCsiDriverConfig": {
        "enabled": true
      }
    },
    "subnetwork": "projects/strong-harbor-338418/regions/us-central1/subnetworks/salaboy-us-1"
  },
  "nodePools": [
    {
      "name": "node-pool-1",
      "nodeConfig": {
        "cpuCores": 2,
        "diskSizeGb": 30,
        "imageType": "DEBONAI_UBUNTU_16_04_LTS"
      },
      "maxPodsConstraint": {
        "maxPods": 11
      },
      "nodeCount": 3
    }
  ]
}
```

Line wrapping



COPY TO CLIPBOARD

CLOSE

Finally, the CLI command option, using the Cloud Provider CLI, in this case, `gcloud`, is once again crafted to contain all the parameters the CLI command needs based on what you have configured in the form.

Figure 1.9 Create via Kubernetes cluster using the `gcloud` CLI (notice horizontal scroll)

gcloud command line

This is the gcloud command line with the parameters you have selected. [gcloud reference](#)

```
$ gcloud beta container --project "strong-harbor-338418" clusters create "cluster-1" --zone "us-central1-c" --no-enable-basic-auth --cluster-version "1.24.9-gke.3200" --release-channel "regular" --machine-type "e2-medium" --image-type "COS_CONTAINERD" --disk-type "pd-balanced" --disk-size "100" --metadata disable-legacy-endpoints=true --scopes "https://www.googleapis.com/auth/devstorage.read_only","https://www.googleapis.com/auth--max-pods-per-node "110" --num-nodes "3" --logging=SYSTEM,WORKLOAD --monitoring=SYSTEM --enable-ip-alias --network "projects/strong-harbor-338418/global/networks/salaboy-vpc" --subnetwork "projects/strong-harbor-338418/regions/us-central1/subnetworks/salaboy-us-subnet" --no-enable-intra-node-visibility --default-max-pods-per-node "110" --no-enable-master-authorized-networks --addons HorizontalPodAutoscaling,HttpLoadBalancing,GcePersistentDiskCsiDriver --enable-autoupgrade --enable-autorepair --max-surge-upgrade 1 --max-unavailable-upgrade 0 --enable-shielded-nodes --node-locations "us-central1-c"
```

Line wrapping



COPY TO CLIPBOARD

RUN IN CLOUD SHELL

CLOSE

There are no differences between these approaches regarding the expected behavior, but you need to consider that when you use the Cloud Provider dashboard, your account credentials are being used from your current session. While if you are crafting a request or using the CLI from outside the Cloud Provider network, you must first authenticate with the Cloud Provider before issuing the request or executing the command to create the resource(s).

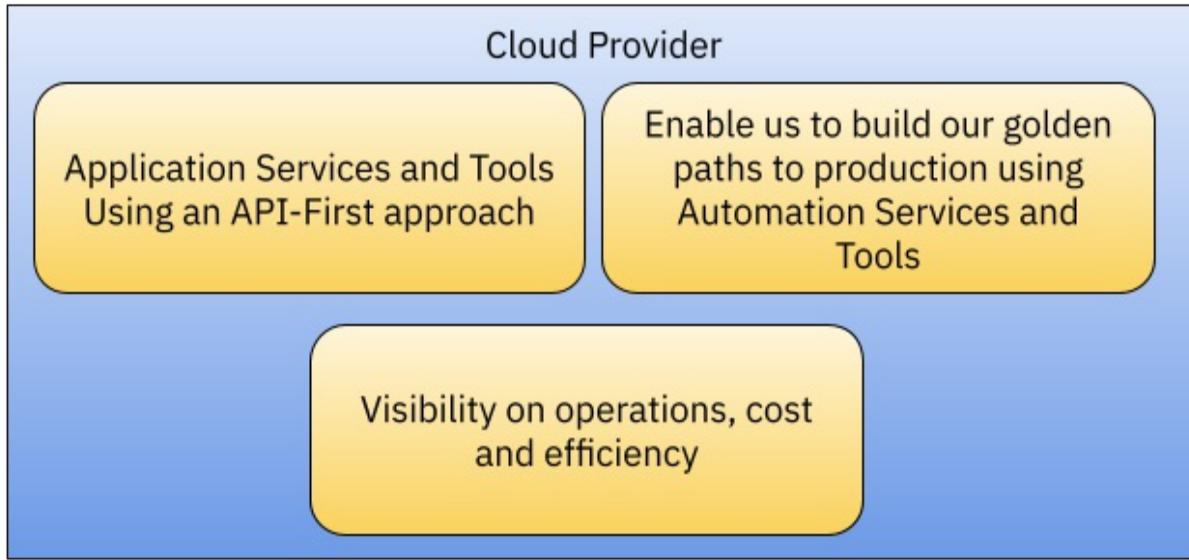
Why do Cloud Providers work?

While one can argue that Dashboards, CLIs, APIs, and SDKs are the primary artifacts we will consume from Cloud Providers, the big question is: how will we combine these tools to deliver software? Suppose you analyze why organizations worldwide trust AWS, Google Cloud Platform, and Microsoft Azure. In that case, you will find that they provide three main features that became the very same definition of a platform today.

By providing an API-First approach, Dashboards, CLIs, SDKs, and a myriad of services, these platforms provide teams with the following (also see Figure 1.10):

- **APIs (contracts):** no matter which tools you are using, the platform should expose an API that enables teams to consume or provision the resources that they need to do their work
- **Golden Paths to production:** the platform allows you to codify and automate the changes that development teams are creating to your production environments which service your applications in front of live customers/users.
- **Visibility:** at all times, by looking at the Cloud Provider dashboard, the organization can monitor which resources are being used, how much each service costs, deal with incidents and have a complete picture of how the organization delivers software.

Figure 1.10 Cloud Provider platforms' advantages



These key features are provided using a competitive pay-as-you-go model that heavily relies on demand (traffic), at a global scale (not for all services), allowing the organization to externalize all the operation and infrastructure costs.

While Cloud Providers are going higher and higher up the stack (providing high-level services not just about provisioning hardware and application infrastructure such as databases), not everything is happy in the clouds.

Unfortunately, this approach comes with some drawbacks, such as being dependent on a Cloud Provider (their tools, cost model, and services), and not having transparent practices or standards across Cloud Providers for cases when you want to migrate from one provider to another or cases when you want to use more than one Cloud Provider.

To understand how to mitigate some of these drawbacks, we need to talk about Kubernetes and the CNCF (Cloud-Native Computing Foundation / <https://www.cncf.io/>) landscape to mitigate these drawbacks. Let's move on to that next.

1.2 Platforms on top of Kubernetes

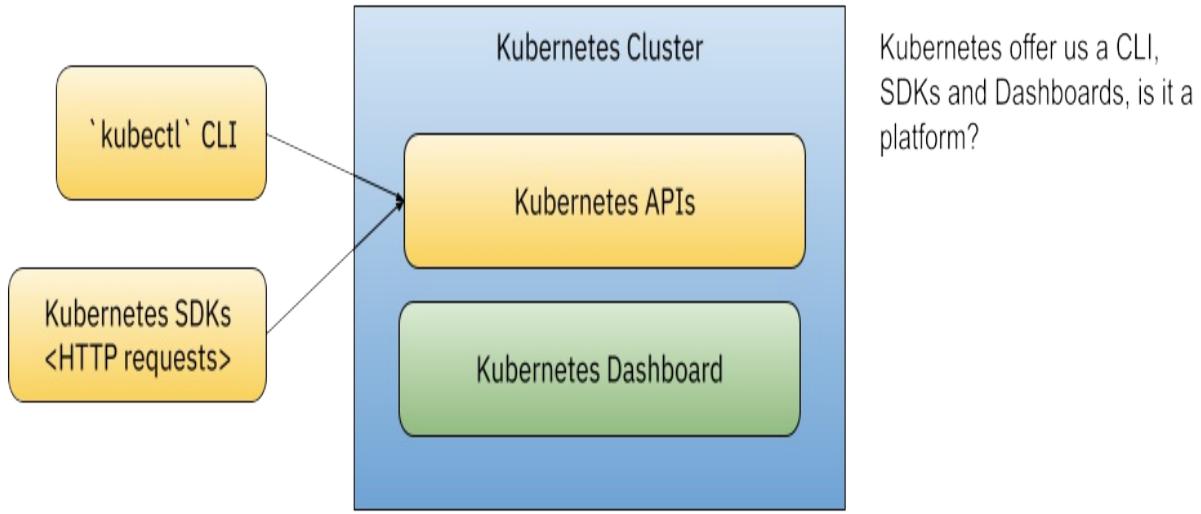
We have briefly discussed what Platforms are and how Cloud Providers are driving the way forward to define what these platforms can do for organizations and development teams in charge of delivering software. But how does this map to Kubernetes? Isn't Kubernetes a platform?

Kubernetes was designed to be a declarative system for our Cloud Native applications. Kubernetes defines a set of building blocks that allows us to run and deploy our workloads. Due to the adoption of the Kubernetes APIs by all the major cloud providers and standardized packaging (containers), Kubernetes became the defacto way to deploy workloads across cloud providers. Because Kubernetes comes with its tools and ecosystem (the CNCF landscape / <https://landscape.cncf.io/>), you can create cloud-agnostic workflows to build, run and monitor your applications. But learning Kubernetes is just the starting point, as the building blocks provided by Kubernetes are very low-level and designed to be composed to build tools and systems that solve more concrete scenarios. Combining these low-level building blocks provided by Kubernetes to build more complex tools to solve more specific challenges is a natural evolutionary step.

While Kubernetes provides us with APIs (the Kubernetes APIs), a CLI ('kubectl'), and a Dashboard (Kubernetes Dashboard - <https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>), Kubernetes is not a platform. Kubernetes is a meta-platform or a platform to build platforms, as it provides all the building blocks you need to build concrete platforms that will solve domain-specific challenges.

Figure 1.11 shows how Kubernetes tools and components maps to what we have been discussing so far about Platforms and Cloud Providers.

Figure 1.11 Kubernetes tools, is it a Platform?



Kubernetes can be extended, and that's why this book will look into specific projects using the Kubernetes APIs, tools, or internal mechanisms to solve generic challenges like Continuous Integration, Continuous Delivery, provisioning cloud resources, monitoring and observability, and developer experience, among others.

Once you get familiar with the tools in the CNCF landscape to solve different challenges, you run into another problem, you need to choose which tools your teams will use and glue them together to provide a unified experience to the consuming teams. This is a challenging task. Projects are going fast, and keeping up with what is happening is a full-time job. This is quite similar to keeping up with a single Cloud Provider's services, how they evolve, and how they complement each other.

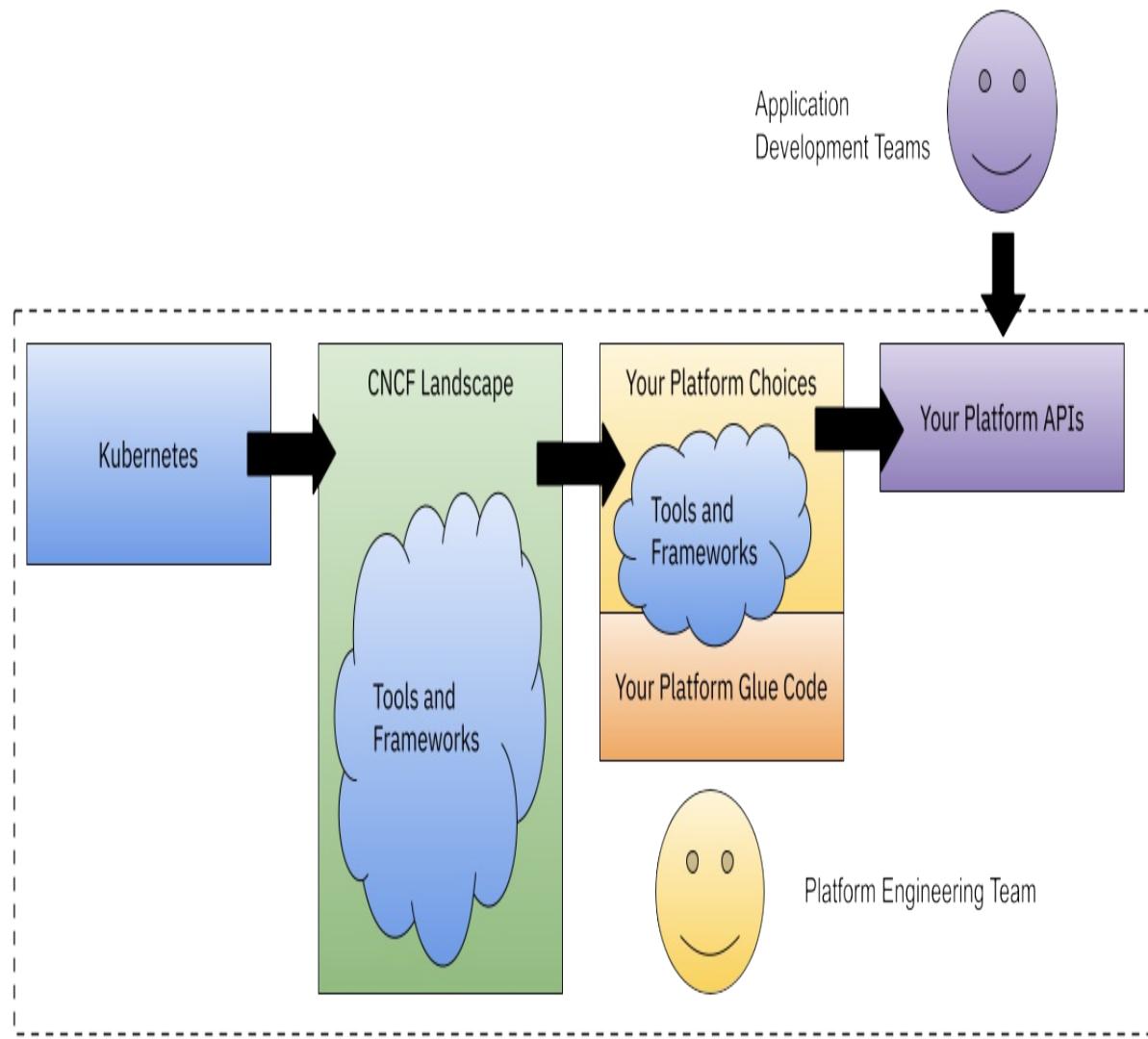
Once you choose which tools you need, you need to run them in production environments and keep them running, but also you need to make sure that your teams can use them, which involves training them. Next, you must ensure that the selected tools play nicely with each other and support your specific use cases and workflows. If they don't, someone must write some "glue code" or "bridge component" to ensure these tools can work together. The "glue code" will heavily depend on your choices and the complexity of the workflows you are trying to implement. Finally, you might want to abstract all the complexity of the tools you have chosen and all the glue code you write from application development teams who are merely consumers,

Kubernetes offer us a CLI, SDKs and Dashboards, is it a platform?

instead of experts, of these tools.

Usually, abstracting away means having a clear contract with your teams that specify what the platform can do for them. These contracts are exposed as APIs that they can interact with programmatically, using a dashboard or via automation. Figure 1.12 below shows a typical Kubernetes adoption journey toward platform engineering. The journey starts by adopting Kubernetes as the target platform to run your workloads, followed by researching and selecting tools, usually from the CNCF Landscape. When some initial tools are selected, your platform starts to shape up, and some investment is needed to configure and make these tools work for your teams. Finally, all these configurations and tools selected can be hidden away from the end user behind a friendlier platform API, allowing them to focus on their workflows instead of knowing every detail about the tools and glue code conforming “the platform”.

Figure 1.12 Platform journey on Kubernetes



Going through this journey, we can define platforms as how we encode the knowledge it takes to provide our development teams with all the workflows they need to be productive. The operational knowledge and the decisions on the tools used to implement these workflows are encapsulated behind a contract materialized as the Platform APIs. These APIs can leverage the Kubernetes APIs to provide a declarative approach, but this is optional. Some platforms might opt to hide that the platform is using Kubernetes, which can also reduce the cognitive load from teams interacting with it.

While I've tried to cover from a very high level what a Platform is, I prefer to delegate all the formal definitions to working groups in the Cloud-Native

space that are in charge of defining and keeping terms updated. I strongly suggest you check the App Delivery TAG - Platform Working Group from the CNCF White Paper on Platforms (<https://tag-app-delivery.cncf.io/whitepapers/platforms/>), which takes on the work of trying to define what platforms are.

Their current definition, at the time of writing this book, reads as follows:

“A platform for cloud-native computing is an integrated collection of capabilities defined and presented according to the needs of the platform’s users. It is a cross-cutting layer that ensures a consistent experience for acquiring and integrating typical capabilities and services for a broad set of applications and use cases. A good platform provides consistent user experiences for using and managing its capabilities and services, such as Web portals, project templates, and self-service APIs.”

In this book, we will embark on this journey of building our platform, by looking at the available Cloud-Native tools and see how they can provide different platform capabilities. But where do we find these tools? Do these tools work together? How do we choose between different alternatives? Let’s take a quick look at the CNCF Landscape.

1.2.1 The CNCF Landscape Puzzle

Keeping up with Cloud Provider services is a full-time job, and each Cloud Provider hosts a yearly conference and more minor events to announce what is new and shiny. In the Kubernetes and Cloud Native space, you can expect the same. The CNCF landscape is continuously expanding and evolving. As you can see in the next figure, the landscape is huge and very difficult to read at first sight:

Figure 1.13 CNCF landscape. See it for yourself at landscape.cncf.io



CNCF Cloud Native Interactive Landscape



Landscape Guide

The cloud native landscape (png, pdf), serverless landscape (png, pdf), and member landscape (png, pdf) are dynamically generated below. Please open a pull request to correct any issues. Greyed logos are not open source. Last updated: 2023-04-25T21:41:35.

Reset Filters

Grouping
CNCF Relation

Sort By
Alphabetical (a to z)

Category

Any

Project

Any

License

Any

Organization

Any

Headquarters

Any

Company Type

Any

Industry

Any

Download as CSV

Example filters

Cards by age

Open source landscape

Member cards

Cards by stars

Cards from China

Certified K8s/KCSP/KTP

Cards by MCap/Funding

Cards without

bestpractices.dev



Landscape

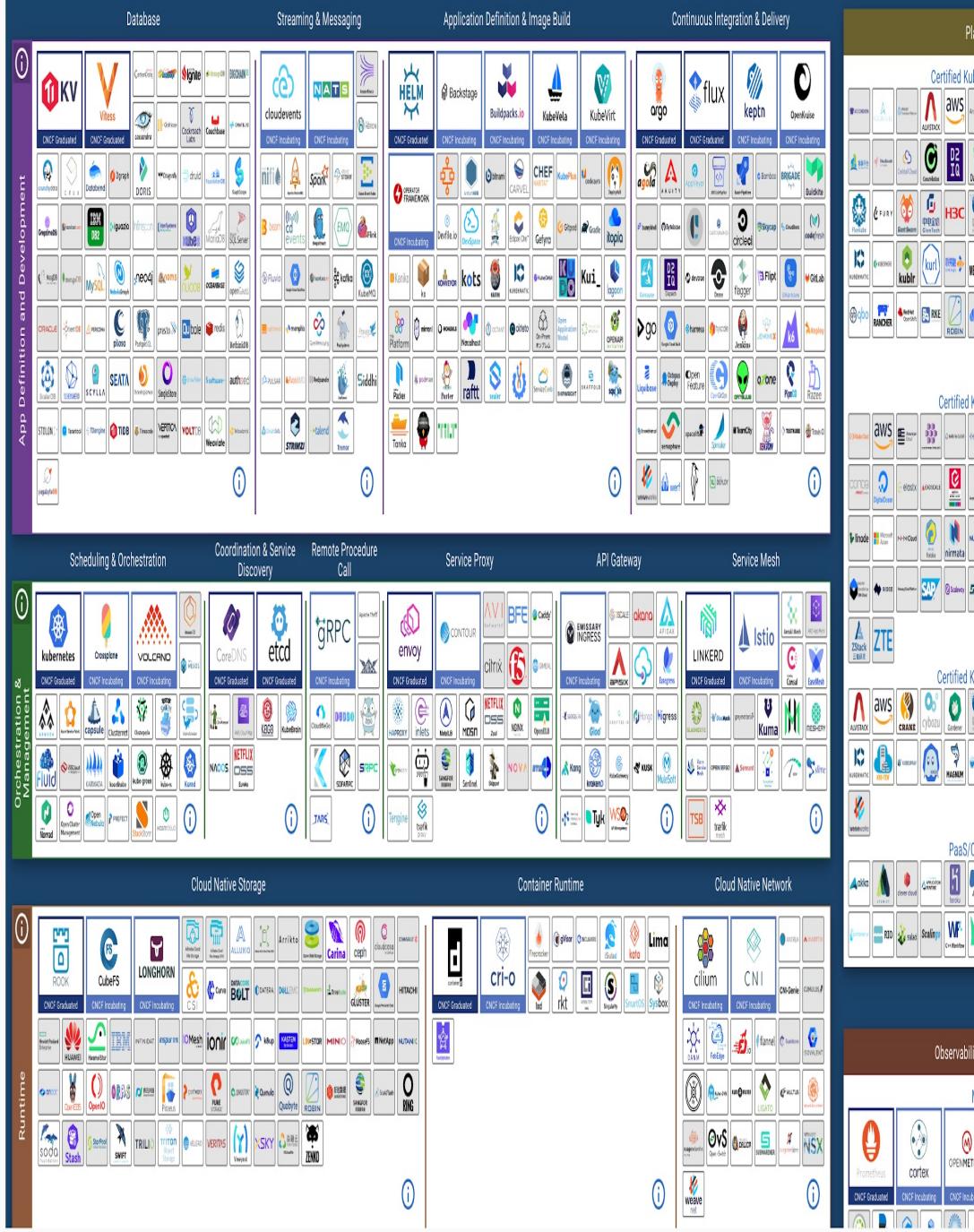
Card Mode

Members

Serverless

Wasm

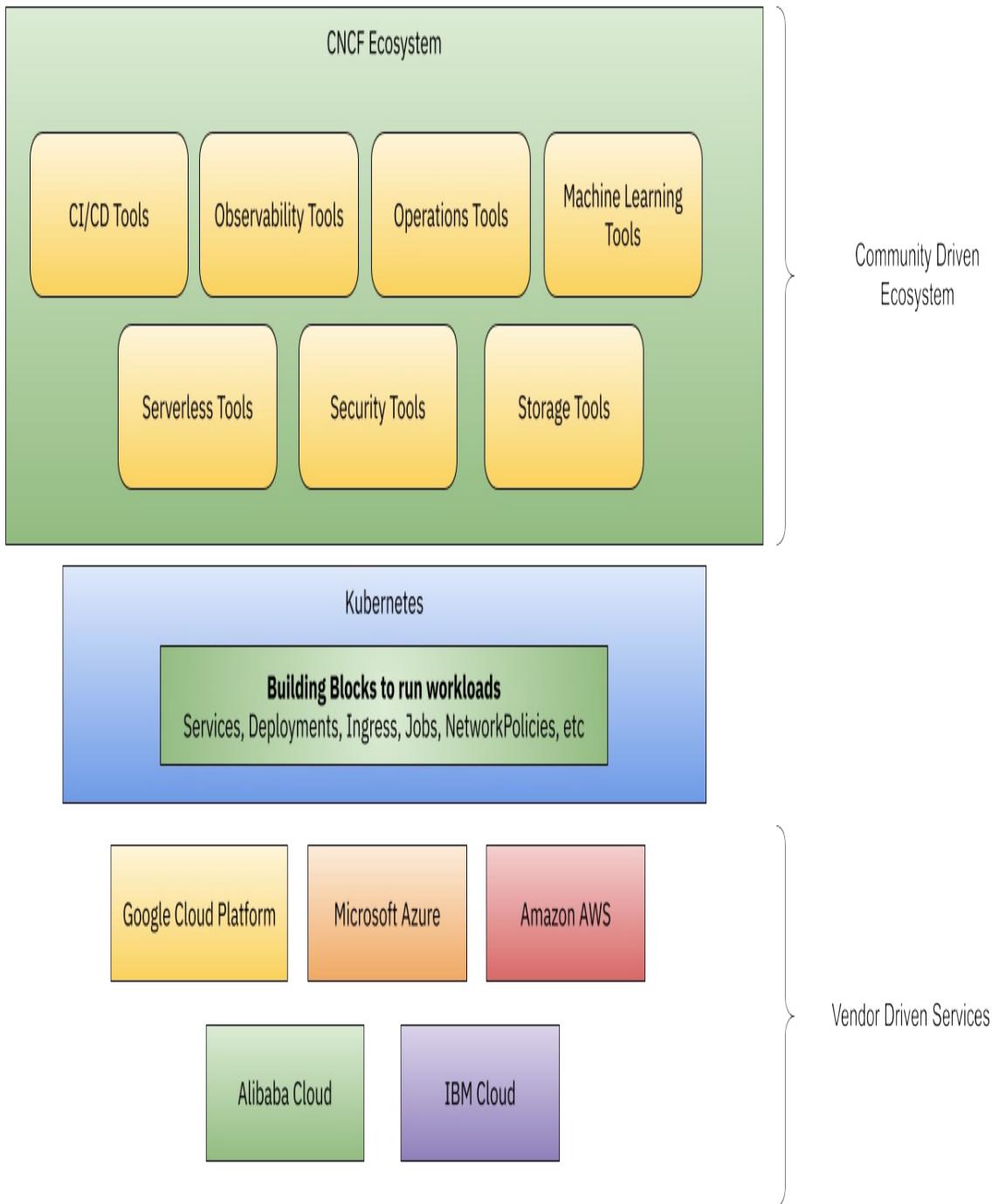
J L T R 100% +



A significant difference compared with Cloud Providers offered services is the maturity model that each project in the CNCF must follow to obtain the graduated status.

While Cloud Providers have defined the Cloud's shape, most are now involved in CNCF projects pushing for these open initiatives to succeed. They are working on tools that can be used across Cloud Providers, removing barriers and allowing innovation in the open instead of behind each Cloud Provider's door. Figure 1.14 shows how Kubernetes enabled the Cloud Native innovation ecosystem to flourish outside Cloud Providers. Cloud Provider hasn't stopped offering new, more specialized services, but in the last 5 years, we have seen a shift towards improved collaborations across Cloud Providers and software vendors to develop new tools and innovation to happen in the open.

Figure 1.14 Kubernetes enabling a multi-cloud Cloud-Native ecosystem



A common denominator from most CNCF-hosted projects is that they all work with Kubernetes, extending it and solving high-level challenges closer to development teams. The CNCF reached a point where more and more tools are being created to simplify development tools and workflows.

Interestingly enough, most of these tools don't focus just on developers. They also enable operation teams and system integrators to glue projects and define new developer experiences native to Kubernetes, development teams don't need to worry about the tooling and integrations required to go to their day-to-day workflows. The increased maturity level of the communities involved in the CNCF landscape and this push to simplify how development teams interact with all these tools gave birth to the conversations around Platform Engineering. The next section will explore these conversations, why you can't buy a platform, and how the rest of the book will work.

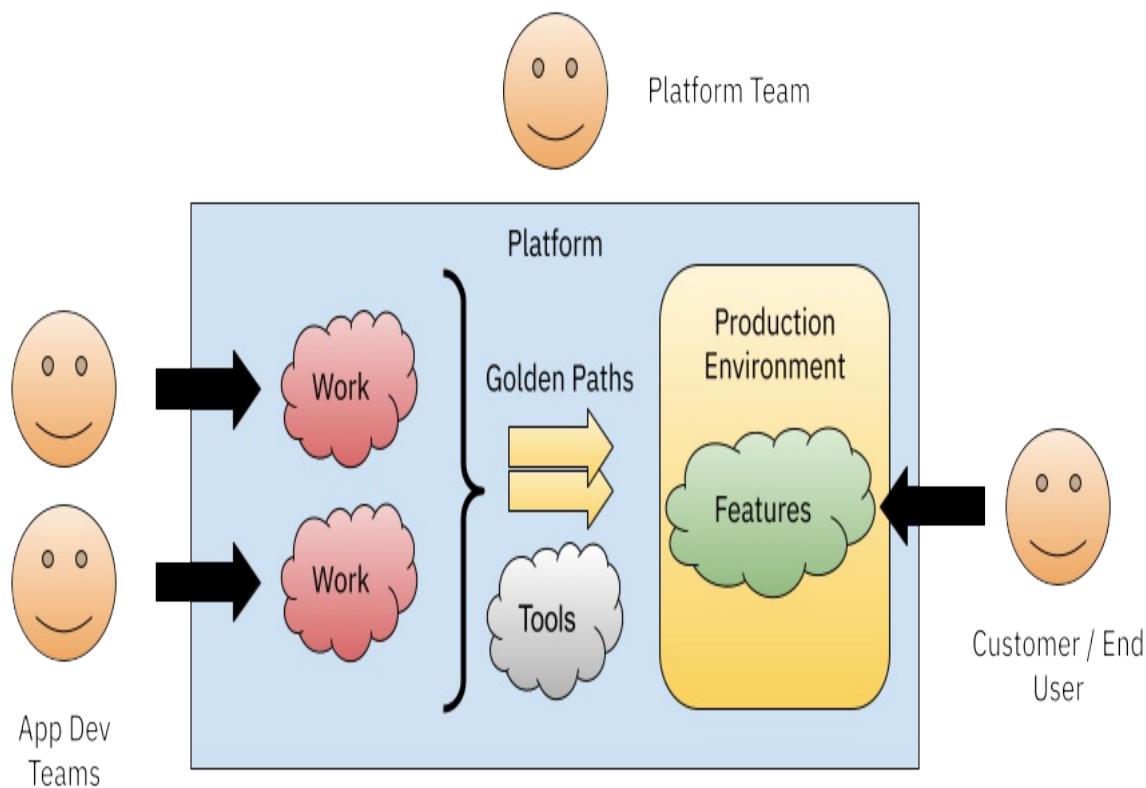
1.3 Platform Engineering

In the same way that Cloud Providers have internal teams defining which new services will be offered, how these services are going to scale, and which tools and APIs need to be exposed to their customers, it became clear that organizations can benefit from having their internal Platform Engineering teams in charge of enabling development teams by making sense and deciding which tools are going to be used to solve and speed up their software delivery process.

A common trend is having a dedicated platform engineering team to define these APIs and make platform-wide decisions. The platform team collaborates with development teams, operations teams, and Cloud Provider experts to implement the workflows application teams need. Besides having a dedicated platform engineering team, a key cultural change promoted by the book Team Topologies (<https://teamtopologies.com/>) is to treat the platform itself as an internal product and your development teams as customers. This is not new, but it pushes the platform team to focus on these internal development teams' satisfaction while using the platform's tools.

Figure 1.15 shows how application development teams (App Dev Teams) can focus on working on new features using their preferred tools while the Platform Team creates Golden Paths (to production), which takes all the work that is being produced by these teams, enable testers to validate the functionality to finally deliver all these changes to our organization customers/end users.

Figure 1.15 Kubernetes enabling a multi-cloud Cloud-Native ecosystem



This relationship between the platform and development teams creates synergy, focusing on improving the entire organization's software delivery practices. By creating Golden Paths, the platform doesn't stop on the day-to-day development tasks. Still, it also aims to automate how the changes made by development teams reach our organization's end customers/consumers.

By adding visibility to the whole process, you can help the entire organization understand and see how teams produce new features and when those features will be available to our end users. This can be very valuable to make business decisions, marketing, and for planning in general.

1.3.1 Why can't I just buy a Platform?

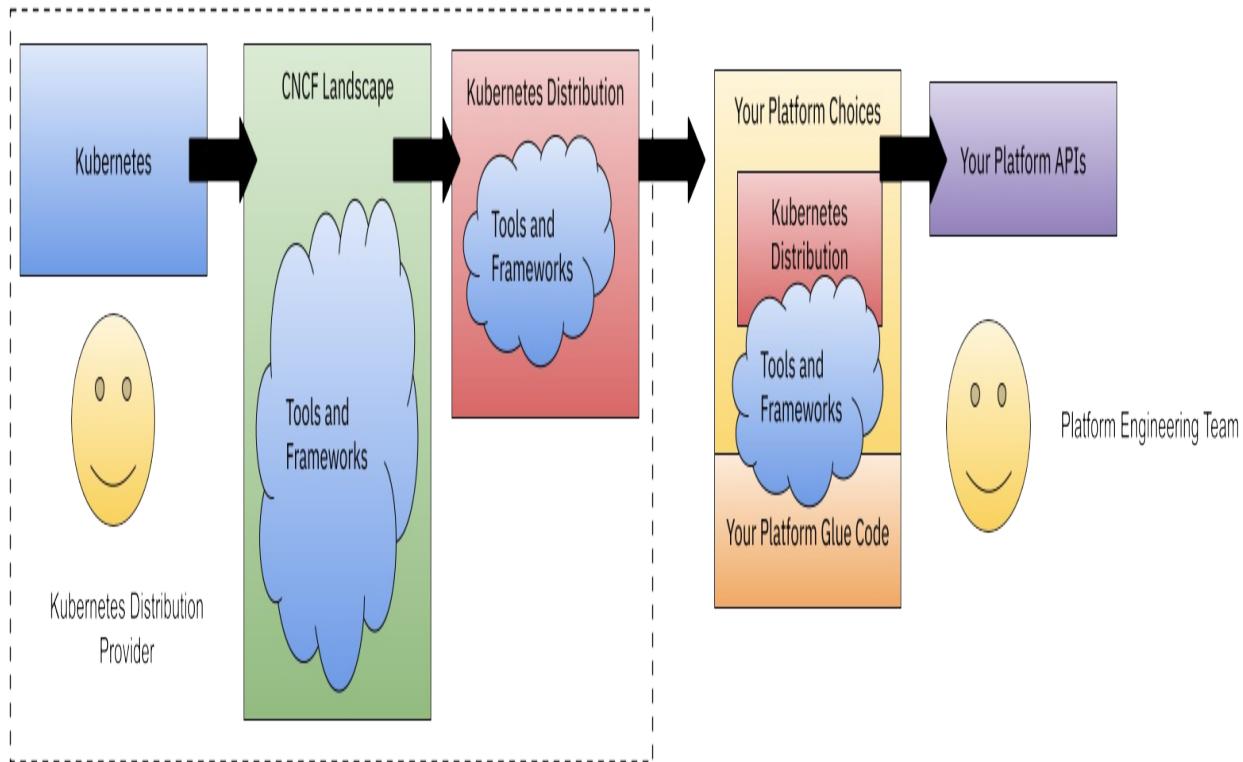
Unfortunately, you can't buy an off-the-shelf platform to solve all of your organization's needs. As we discussed, you can buy one or more Cloud Provider services, but your internal teams will need to figure out which services and how they must be combined to solve specific problems. This

exercise of figuring out which tools and services your organization needs and then encapsulating these decisions behind interfaces that teams can consume using a self-service approach is usually something that you can't buy.

There are tools designed with this situation in mind, trying to enable Platform Teams to do less gluing by implementing a set of out-of-the-box workflows or having a very opinionated set of tools they support. Tools in this category that are also heavily using and extending Kubernetes are Red Hat Openshift (<https://www.redhat.com/en/technologies/cloud-computing/openshift>) and VMware Tanzu (<https://tanzu.vmware.com/tanzu>). These tools are very attractive to CTOs (Chief Technology Officers) and Architects as they cover most topics they need solutions for, such as CI/CD, Operations, developer tooling, and frameworks. Based on my experience, while these tools are helpful in many scenarios, Platforms Teams require flexibility in their chosen tools that fits their existing practices. At the end of the day, if you buy these tools, your teams will also need to spend time learning them, which is why these tools like Red Hat Openshift and VMware Tanzu are sold with consulting services, which is another cost to factor into the equation.

Figure 1.16 shows how the journey changes depending on which tools the platform team chooses to rely on. These Kubernetes distributions (Openshift, Tanzu, among others) can limit the number of choices the platform teams can make, but they can also save time and come with services such as training and consulting that your teams can rely on.

Figure 1.16 Building Platforms on top of Kubernetes Distributions



No matter if you are already a customer of these tools, you will still be in charge of building a platform on top of these tools. If you have Red Hat Openshift or VMware Tanzu available to your teams, I strongly encourage you to familiarize yourself with the tools they support and their design choices and decisions. Aligning with the tools you have and consulting with their architects might help you find shortcuts to build your layers on top of these tools.

A word of caution: It is crucial to notice that these tools can be considered Kubernetes distributions. Distributions in the same sense as Linux distributions mean that I expect more and more distributions to appear, tackling different challenges and use cases. Tools like K0s and MicroK8s for IoT and edge cases are other examples. While you can adopt any of these distributions, ensure they align with your organization's goals.

Because I want to keep this book as practical as possible, in the next section, we will look at a simple application we will use to go on our journey. We will not build a generic platform for a generic use case, so having a concrete example that you can run, experiment with, and change should help you map

the topics discussed around this example with your day-to-day challenges. The application introduced in the next section tries to cover challenges that you will find in most business domains. Hence, the platform we will build to support this application and its customers should also apply to your business.

1.4 The Need for a walking skeleton

In the Kubernetes ecosystem, it is common to need at least to integrate 10 or more projects or frameworks to deliver a simple PoC (proof of concept). From how you build these projects into containers that can run inside Kubernetes to how to route traffic to the REST endpoints provided in each container. If you want to experiment with new projects to see if they fit into your ecosystem, you build a PoC to validate your understanding of how this shiny new project works and how it will save your and your teams' time.

For this book, I have created a simple “Walking Skeleton”, which is a Cloud-Native application that goes beyond being a simple PoC and allows you to explore how different architectural patterns can be applied and how different tools and frameworks can be integrated, without the need of changing your projects for the sake of experimentation.

The primary purpose of this walking skeleton is to highlight how to solve very specific challenges from the architectural point of view, the requirements that your applications will need, and the delivery practices' angle. You should be able to map how these challenges are solved in the sample Cloud-Native application to your specific domain. Challenges are not always going to be the same, but I hope to highlight the principles behind each proposed solution and the approach taken to guide your own decisions.

With this walking skeleton, you can also figure out the minimum viable product you need and deploy it quickly to a production environment where you can improve. By taking the walking skeleton to a production environment, you can gain valuable insights into what you will need for other services and from an infrastructure perspective. It can also help your teams to understand what it takes to work with these projects and how and where things can go wrong.

The technology stack used to build the “walking skeleton” is not important. It is more important to understand how the pieces fit together and what tools and practices can enable each team behind a service (or a set of services) to evolve safely and efficiently.

1.4.1 Building a Conference Management Application

During this book, you will be working with a Conference Management application. This conference application can be deployed in different environments to serve different events. This application relies on containers, Kubernetes, and tools that will work across any major Cloud-Providers and On-Prem Kubernetes installations.

This is how the application's main page looks like:

Figure 1.17 Conference Management home page



Go to Back Office >

Accepted talks

AGENDA SERVICE (REST) V0.0.4

12

MON

MICROSERVICES 101

5:00 pm

By Well Known Speaker

13

TUE

THE FUTURE OF CLOUD NATIVE COMPUTING

4:00 pm

By Conference Organizer

CLOUD EVENTS ORCHESTRATION

1:00 pm

By Salaboy

AI/ML TRENDS 2020

3:00 pm

By AI/ML Engineer

Conference Agenda



Potential Speakers Submit



Submit Proposal

The conference application lists all the approved submissions on the Agenda Page. The main page will also allow potential speakers to submit proposals while the “Call for Proposals” window remains open. There is also a Back Office section for the organizers to review proposals and do admin tasks while organizing the conference:

Figure 1.18 Conference Platform Back Office Page

Received Proposals

C4P SERVICE (REST) V0.0.10

SHOW ONLY PENDING

Approved Proposals



CLOUD EVENTS ORCHESTRATION

16:00 APPROVED ✓

How to orchestrate Cloud Events in a friendly way

THE FUTURE OF CLOUD NATIVE COMPUTING

16:00 APPROVED ✓

What's coming in 2021 and beyond

MICROSERVICES 101

16:00 APPROVED ✓

Getting Started with microservices.

AI/ML TRENDS 2020

16:00 APPROVED ✓

New algorithms and industry trends in applied AI and ML

REJECT

please reject this abstract

16:00 DECLINED ✘

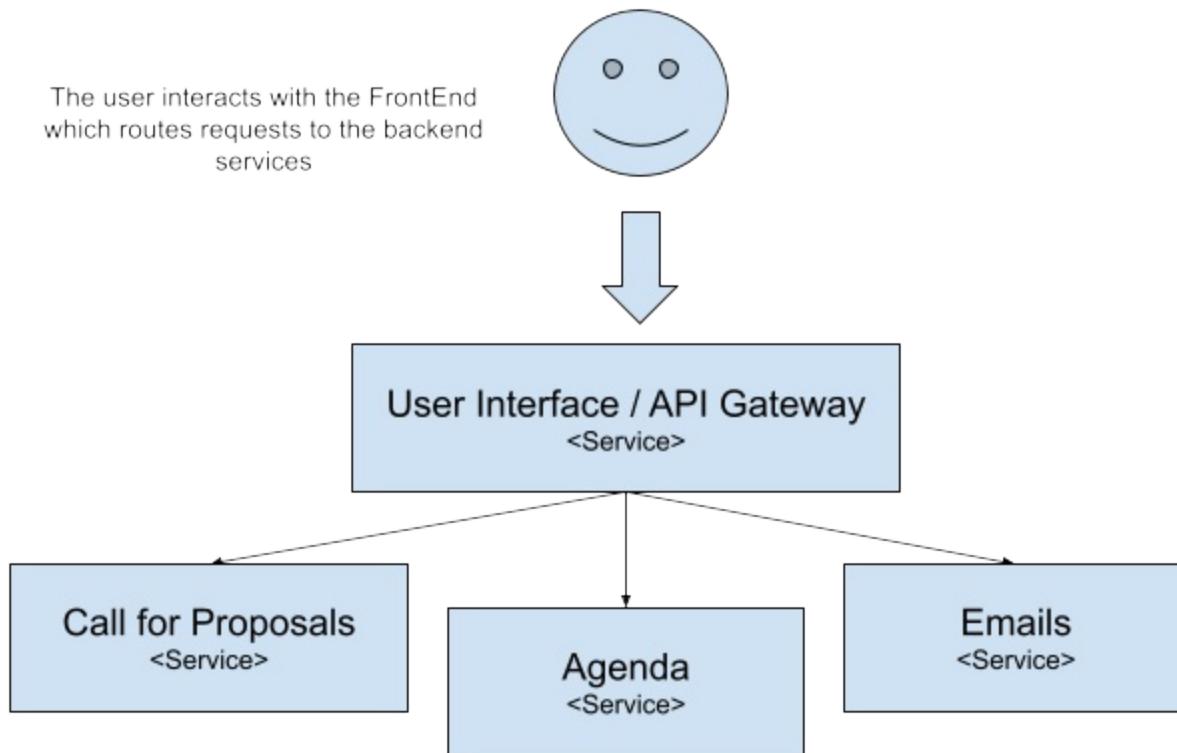
Communications

Rejected Proposals

Send Email

This application is composed of a set of services that have different responsibilities. Figure 1.19 shows the main components of the application that you control, in other words, the services that you can change and that you are in charge of delivering.

Figure 1.19 Conference Platform Services



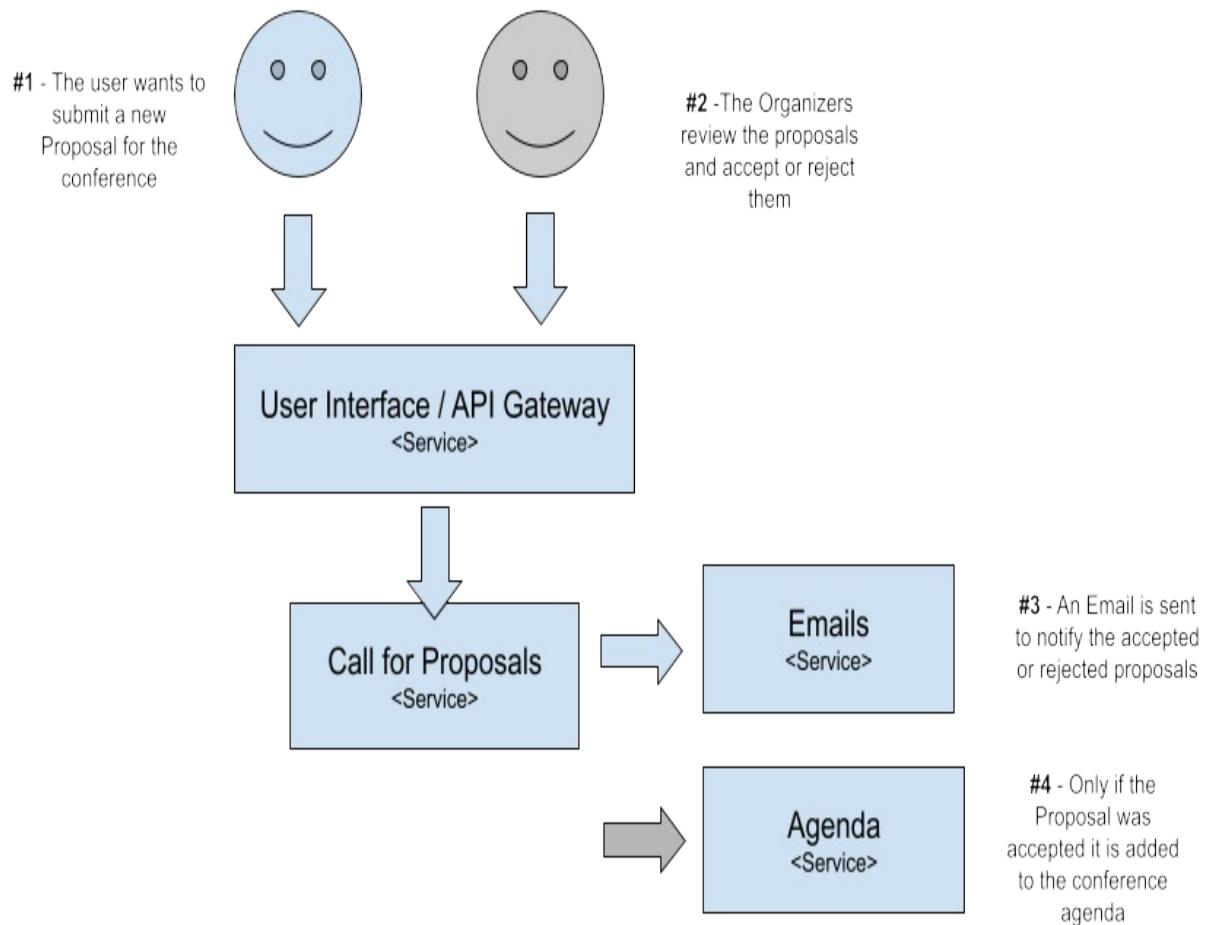
These services functionally compose the application, here is a brief description of each service:

- **User Interface:** This service is the main entry point for your users to access the application. For this reason, the service hosts the HTML, JavaScript and CSS files that will be downloaded by the client's browser interacting with the application.
- **Agenda Service:** This service deals with listing all the talks that were approved for the conference. This Service needs to be highly available during the conference dates, as the attendees will be hitting this service several times during the day to move between sessions.
- **Email Service:** This service is just a facade exposing rest endpoints to

abstract an SMTP email service that needs to be configured in the infrastructure where the application is running.

- **Call for Proposals (C4P):** This service contains the logic to deal with “Call for Proposals” use case (C4P for short) when the conference is being organized. As you can see in the following diagram, the C4P service calls both the Agenda and the Email Service. Hence these two services are considered “downstream” services from the C4P service perspective

Figure 1.20 Call for Proposals Use Case



It is pretty normal in Cloud-Native architectures to expose a single entry point for users to access the application. This is usually achieved using an API Gateway, which is in charge of routing requests to the backend services that are not exposed outside the cluster. The User Interface container serves

this purpose for this application, so no separate API Gateway is being used.

This simple application implements a set of well-defined use cases that are vital for the events to take place, such as:

- **Call for Proposals:** potential speakers submit proposals that need to be validated by the conference organizers. If approved, the proposals are published in the conference agenda.
- **Attendee Registration:** attendees need to buy a ticket to attend the conference
- **Event Agenda (schedule):** host the approved proposals, times, and descriptions.
- **Communications:** Sending organizers, attendees, and sponsors emails.

While looking at how these use cases are implemented, you need to consider also how to coordinate across teams when new use cases will be implemented or when changes need to be introduced. To improve collaboration, you need visibility, and you need to understand how the application is working.

You also need to consider the operation side of this Cloud-Native application. You can imagine there will be a period when the application will open the “Call for Proposals” request for potential speakers to submit proposals, then closer to the conference date, open the attendee registration page, etc.

During the length of the book, I will encourage you to experiment by adding new services and implementing new use cases. In chapter 2, when you deploy the application to a Kubernetes Cluster, you will inspect how these services are configured to work, how the data flows between the different services, and how to scale the services.

By playing around with a fictional application, you are free to change each service's internals, use different tools and compare results or even have different versions of each service to try in parallel. Each service provides all the resources needed to deploy these services to your environment. In Chapter 3, you will go deeper into each service to understand how to build each service so teams can change the current behavior and create new releases.

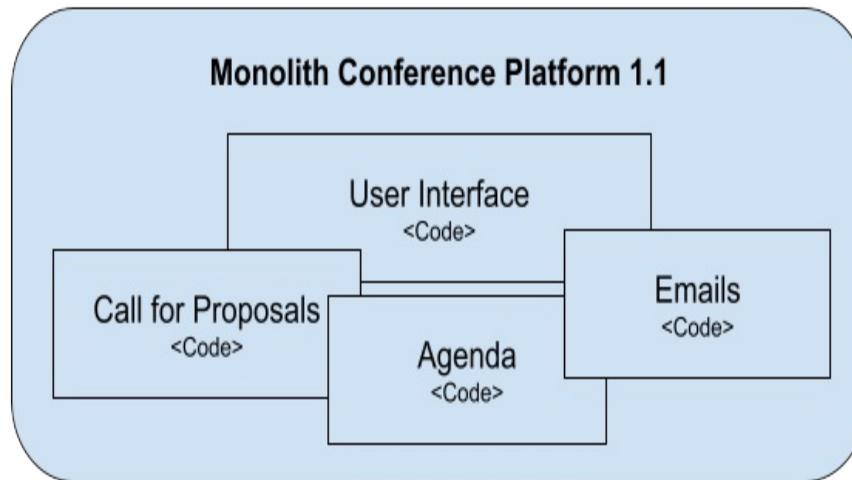
Before moving forward to deploy this Cloud-Native conference management application, it is important to mention some of the main differences with having all these functionalities bundled up in a single monolithic application.

The Cloud-Native Conference Platform was created based on a monolith application that was serving the same purposes, but it had several drawbacks. The monolithic application implemented exactly the same use cases, but it suffered from several drawbacks discussed in the next section.

1.4.2 Differences with a Monolith

Understanding the differences between having a single monolithic application instead of a fully distributed set of services is key to grasp why the increased complexity is worth the effort. If you are still working with monolithic applications that you want to split up to use a distributed approach, this section highlights the main differences that you will encounter between a monolith and a set of services implementing more or less the same functionalities from the end-user perspective.

Figure 1.21 The monolith approach



The monolith approach will include all the logic for different use cases tightly coupled under a single application version. This pushes teams all to work on the same code base and coordinate how changes are applied and released.

Functionally wise, they are the same, you can do the same amount of use cases, but the monolith application presented some drawbacks that you might be experiencing with your monolith applications already. The following points compare the Cloud-Native approach with the previous monolith

implementation:

- **Now services can evolve independently, teams are empowered to go faster, there is no bottleneck at the code-base level:** In the monolithic application, there was a single source code repository for different teams to work on, there was a single Continuous Integration pipeline for the project which was slow, and teams were using feature branches that caused problems with complex merges.
- **Now the application can scale differently for different scenarios:** from a scalability perspective, each service can be now scaled depending on the load level that they experience. With the monolith application, the operations team could only create new instances of the entire application if they needed to scale just a single functionality. Having fine-grained control over how different functionalities can be scaled can be a significant differentiator for your use case, but you must do your due diligence.
- **The Cloud-Native version is much more complex, as it is a distributed system:** but it leverages the flexibility and characteristics of the cloud infrastructure much better, allowing the operation teams to use tools to manage this complexity and the day-to-day operations. When building monolithic applications, creating your in-house mechanism to operate your large applications was much more common. In Cloud-Native environments, there are a lot of tools provided by Cloud Providers and Open Source projects that can be used to operate and monitor Cloud-Native applications.
- **Each service can be built using a different programming language or different frameworks:** with the monolith, application developers were stuck in old versions of libraries, as changing or upgrading a library usually involved large refactorings, and the whole application needed to be tested to guarantee that the application will not break. In a Cloud-Native approach, services are not forced to use a single technology stack. This allowed teams to be more autonomous in choosing their tools, which for some situations, can speed up the delivery times.
- **All or nothing with the monolith:** if the monolith application went down, the entire application was down, and users would not be able to access anything. With the Cloud-Native version of the application, users

can still access the application even if services are going down. The example “walking skeleton” shows how to support degraded services by adopting some popular tools. By using Kubernetes, which was designed to monitor your services and take action in case a service is misbehaving, the platform will try to self-heal your applications.

- **Each conference event required a different version of the monolith:** when dealing with different conference events, each conference required a version of the monolith that was slightly different from the other events, this caused divergence of codebases and duplication of the entire project. Most of the changes done for a conference were lost when the event was done. In the Cloud-Native approach, we promote reusability by having fine-grained services that can be swapped, avoiding duplication of the whole application.

While the monolith application is much more straightforward to operate and develop than the Cloud-Native application, the remainder of the book focuses on showing you how to deal with the complexity of building a distributed application. We’ll do that by looking at adopting the right tools and practices, which will unlock your teams to be more independent and efficient while promoting resiliency and robustness to your applications.

1.4.3 How the rest of the book works

Now that we understand the walking skeleton application that we will use, the rest of the book will take us on a Cloud-Native discovery journey. We will look at different aspects of working with Cloud-Native applications and the tools the Cloud-Native communities design to solve specific challenges. This journey will push us to make hard decisions and choices that will become critical for our Platform Engineering practices. The following list covers the main milestones in this journey without going into the details of the specific tools covered in each chapter.

- **Chapter 2: The challenges of Cloud-Native applications:** After getting the Conference application up and running in a Kubernetes Cluster, we will analyze the main and most common challenges that you will face when working and running Cloud-Native applications on top of Kubernetes. In this chapter, you will inspect the application from a

runtime perspective and try to break it into different ways to see how it behaves when things go wrong.

- **Chapter 3: Service Pipelines - Building Cloud-Native Applications:** Once the application is up and running, you and your teams will make changes to the application's services to add new features or fix bugs. This chapter covers what it takes to build these application services, including the new changes using Service Pipelines to create a release of the artifacts needed to deploy these new versions into live environments.
- **Chapter 4: Environment Pipelines - Deploying Cloud Native Applications:** If we sort out how to package and release new versions of our services, then we need to have a clear strategy on how to promote these new versions to different environments so they can be tested and validated before facing real customers. This chapter covers the concept of Environment Pipelines and a popular trend in the Cloud-Native community called GitOps to configure and deploy applications across different environments.
- **Chapter 5: Multi-cloud (App) infrastructure:** Your applications can't run in isolation, application's services need application infrastructure components such as databases, message brokers, identity services, etc. This chapter focuses on how to provision the components that our application's services need using a multi-cloud and Kubernetes-native approach.
- **Chapter 6: Let's build a platform on top of Kubernetes:** Once we have understood how the application runs, how it is built and deployed, and how it connects to cloud infrastructure, we will focus our attention on abstracting the complexity introduced by all the tools that we are using from the teams working to make changes to the application. We don't want our development teams to get distracted setting up cloud provider accounts, configuring the servers where the build pipelines will run, or worrying about where their environment is running. Welcome to the Platform Engineering team!
- **Chapter 7: Platform Capabilities I: Enable Developers to Experiment:** Now that we have a platform that takes care of provisioning environments for our teams to do their work, what else can the platform do for the application development teams? If you enable your teams to run more than a single version of your application's services at the same time, new features or fixes can be rolled-out

incrementally, which allows the organization to find issues sooner and also reduce the stress of each release process. This chapter covers implementing different release strategies using tools like Knative Serving and Argo Rollouts.

- **Chapter 8: Platform Capabilities II: Shared Application Concerns:** How can we reduce friction and dependencies between application and operation teams? How can we decouple even further the logic of our applications from the components that these applications need to run? This chapter covers a set of platform capabilities that enable application developers to focus on writing code. The platform team can focus on deciding how to wire all the components required by the application and then expose simple and standardized APIs that developers can consume. This chapter covers two CNCF projects: Dapr and OpenFeature, to demonstrate how you can build platform-level capabilities.
- **Chapter 9: Measuring our Platforms using DORA metrics:** Platforms are as good as the improvements they bring to the entire organization. We need to measure our platform performance to see how well it's doing, as we should use a continuous improvement approach to ensure that the tools we are using are helping our teams deliver faster and more efficiently. This chapter focuses on using the DORA metrics to understand how well the organization is delivering software and how platform changes can improve the throughput of our delivery pipelines.

Now that you know what is coming, let's get ready to deploy and try to break our Cloud-Native Conference application.

1.5 Summary

- Cloud-Native applications follow the 12-factor principles to build robust and resilient distributed applications that can scale. You are going to be using Docker containers and Kubernetes to run these distributed applications.
- The Continuous Delivery practices goal aims to reduce the cycle time between a decision for a change is made and the change is live for the end-users.

2 Cloud-native applications challenges

This chapter covers

- Working with a Cloud-Native application running in a Kubernetes Cluster
- Choosing between local and remote clusters
- Exploring the application to understand the main components and Kubernetes Resources
- How to break a Cloud-Native application and understanding the challenges of distributed applications

When I want to try something new, a framework, a new tool, or just a new application, I tend to be impatient; I want to see it running right away. Then, when it is running, I want to be able to dig deeper and understand how it is working. I tend to break things to experiment and validate that I understand how these tools, frameworks or applications are internally working. That is the sort of approach we'll take in this chapter!

To have a Cloud-Native application up and running you will need a Kubernetes Cluster. In this chapter, you are going to work with a local Kubernetes Cluster using a project called KinD (Kubernetes in Docker - <https://kind.sigs.k8s.io/>). This local cluster will allow you to deploy applications locally for development and experimentation purposes. To install a set of microservices you will be using Helm, a project that helps us to package, deploy and distribute Kubernetes applications. You will be installing the walking skeleton services introduced in Chapter 1 which implements a Conference application.

Once the services for the Conference application are up and running you will inspect its Kubernetes resources to understand how the application was architected and its inner workings by using `kubectl`. Once you get an overview of the main pieces inside the application, you will jump ahead to try

to break the application, finding common challenges and pitfalls that your Cloud-Native applications can face. This chapter covers the basics of running Cloud-Native applications in a modern technology stack based on Kubernetes highlighting the good and the bad that comes with developing, deploying and maintaining distributed applications. The following chapters tackle these associated challenges by looking into projects whose main focus is to speed up and make more efficient the delivery of your projects.

2.1 Running the walking skeleton

To understand the innate challenges of Cloud-Native applications, we need to be able to experiment with a “simple” example that we can control, configure, and break for educational purposes. In the context of Cloud-Native applications “simple” cannot be a single service hence for “simple” application we will need to deal with the complexities of distributed applications such as networking latency, resilience to failure on some of the applications’ services and eventual inconsistencies. To run a Cloud-Native application, in this case, the walking skeleton introduced in chapter 1, you need a Kubernetes cluster. Where this cluster is going to be installed and who will be responsible for setting it up is the first question that developers will have. It is quite common for developers to want to run things locally, in their laptop or workstation, and with Kubernetes, this is definitely possible... but is it optimal? Let’s analyze the advantages and disadvantages of running a local cluster against other options.

2.1.1 Choosing the best Kubernetes environment for you

This section doesn’t cover a comprehensive list of all the available Kubernetes flavors, but it focuses on common patterns on how Kubernetes clusters can be provisioned and managed.

There are three possible alternatives. All them with advantages and drawbacks:

- **Local Kubernetes in your laptop/desktop computer:**

I tend to discourage people from using Kubernetes running on their laptops, as you will see in the rest of the book, running your software in

similar environments to production is highly recommended to avoid issues that can be summed up as “but it works on my laptop”. These issues are most of the time caused by the fact that when you run Kubernetes on your laptop, you are not running on top of a real cluster of machines. Hence, there is no network, no round-trips and no real load balancing.

- Pros: lightweight, fast to get started, good for testing, experimenting and local development. Good for running small applications.
 - Cons: not a real cluster, it behaves differently, reduced hardware to run workloads. You will not be able to run a large application on your laptop.
- **On-Premise Kubernetes in your data center:**

This is a typical option for companies where they have private clouds. This approach requires the company to have a dedicated team and hardware to create, maintain and operate these Clusters. If your company is mature enough, it might have a self-service platform that allows users to request new Kubernetes Clusters on demand.

 - Pros: real cluster on top of real hardware, will behave closer to how a production cluster will work. You will have a clear picture of which features are available for your applications to use in your environments.
 - Cons: it requires a mature operation team to set up clusters and gives credentials to users, it requires to have dedicated hardware for developers to work on their experiments
 - **Managed Service Kubernetes offering in a Cloud Provider:**

I tend to be in favor of this approach, as using a Cloud Provider service allows you to pay for what you use, and services like Google Kubernetes Engine (GKE), Azure AKS and AWS EKS are all built with a self-service approach in mind, enabling developers to spin up new Kubernetes Cluster quickly. There are two primary considerations:
1) You need to choose one and have an account with a big credit card to pay for what your teams are going to consume, this might involve setting up some caps in the budget and defining who has access. By selecting a Cloud Provider, you might be going into a vendor lock-in situation, if you are not careful.
2) Everything is remote, and for some people, this is too big of a change.

It takes time for developers to adapt, as the tools and most of the workloads will be running remotely. This is also an advantage, as the environments used by your developers and the applications that they are deploying are going to behave as if they were running in a production environment.

- **Pros:** You are working with real (fully-fledged) clusters, you can define how much resources you need for your tasks, when you are done you can delete it to release resources. You don't need to invest in hardware upfront.
- **Cons:** you need a potentially big credit card, you need your developers to work against remote clusters and services.

A final recommendation is to check the following repository which contains free Kubernetes credits in major Cloud Providers:

<https://github.com/learnk8s/free-kubernetes>. I've created this repository to keep an updated list of these free trials that you can use to get all the examples in the book up and running on top of real infrastructure.

Figure 2.1 Kubernetes Cluster Local vs Remote setups

Developers are used
to work locally



These are in-house
setups, but they feel
remote



These are fully remote setups,
developers needs to get used
to new tools and procedures

Local	On Prem	Cloud Provider
<p>Pros</p> <ul style="list-style-type: none">- Lightweight- Fast to get started- Good for local development- Good for testing (CI)- Good for small applications <p>Cons</p> <ul style="list-style-type: none">- Limited capacity- It doesn't behave as a real cluster as it is not running on top of a real machines- Network bandwidth to download containers to your local environment	<p>Pros</p> <ul style="list-style-type: none">- Real Cluster on top of machines- It behaves closer to a production environment- Provides a remote environment for developers to work <p>Cons</p> <ul style="list-style-type: none">- Requires to have hardware available- Requires a mature operation team to provision and distribute credentials- It might lack features that are present in Cloud Providers such as integrations with Databases and Message Brokers- Difficult to scale the operation for a large number of clusters	<p>Pros</p> <ul style="list-style-type: none">- Fully-Fledged Managed Clusters- You don't need to deal with hardware- Easy to scale and manage- Other services provided- Pay-as-you-go <p>Cons</p> <ul style="list-style-type: none">- Difficult to estimate cost- Possible vendor lock-in- Cloud Provider specific expertise required- You might need a big credit card to pay for what you use

While these three options are all valid and have drawbacks, in the next sections, you will be using Kubernetes KinD (Kubernetes in Docker: <https://kind.sigs.k8s.io/>) to deploy the walking skeleton introduced in chapter one in a local Kubernetes environment running in your laptop/pc.

2.1.2 Installing Kubernetes KinD locally

There are several options to create Local Kubernetes Clusters for development. This book has chosen KinD because it supports different platforms and has ease of customization to run your clusters with minimum dependencies, as KinD doesn't require you to download a Virtual Machine.

For practical reasons, having access to a local Kubernetes environment can help you to get started. It is essential to understand that most of the steps are not tied to Kubernetes KinD in any way, meaning that you can run the same commands against a remote Kubernetes Cluster (on-prem or in a cloud provider). If you have access to a fully-fledged Kubernetes Cluster I encourage you to use that one instead, you can skip the following section on KinD and move straight away to 2.X: Installing the application with Helm.

For the examples in this section to work you need to have installed:

- **Docker**, follow the documentation provided on their website to install: <https://docs.docker.com/get-docker/>
- **Kubernetes KinD** (Kubernetes in Docker), follow the documentation provided on their website to install KinD in your laptop: <https://kind.sigs.k8s.io/docs/user/quick-start/#installation>
- `kubectl`, follow the documentation provided in the official Kubernetes site to install `kubectl` <https://kubernetes.io/docs/tasks/tools/>
- **Helm**, you can find the instructions to install Helm in their website: <https://helm.sh/docs/intro/install/>

Once we have everything installed, we can start working with KinD, which is a project that enables you to run local Kubernetes clusters, using Docker container “nodes”.

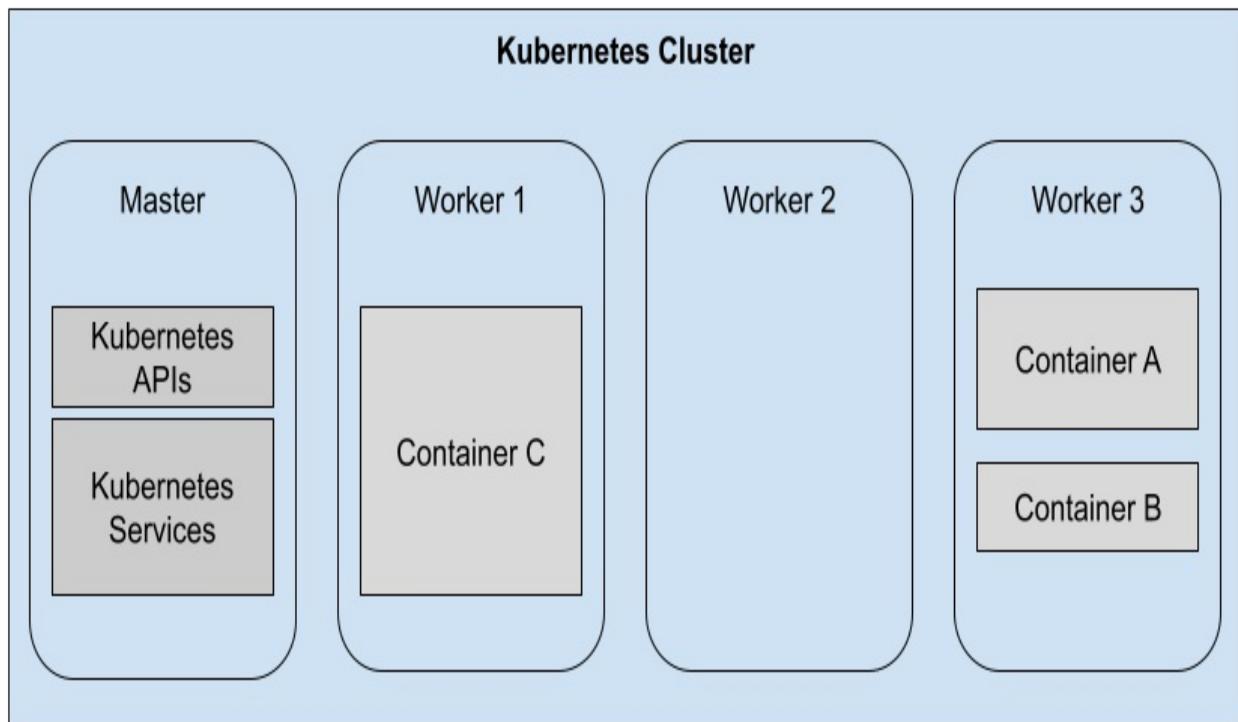
In this section, you will be creating a local Kubernetes Cluster in your laptop/pc and setting it up so you can access the applications running inside it.

By using KinD, you can quickly provision a Kubernetes Cluster for running and testing your applications; hence it makes a lot of sense when working with applications composed of several services to use a tool like this to run integration tests as part of your Continuous Integration pipelines.

Once you have `kind` installed in your environment, you can create clusters by running a single line in the terminal.

The cluster you are going to create will be called `dev`, and will have four nodes, three workers, and a master node (control plane), as seen in Figure 2.2. We want to be able to see in which nodes our application services are placed inside our Kubernetes Cluster.

Figure 2.2 Kubernetes Cluster topology



KinD will simulate a real cluster conformed by a set of machines or virtual machines, in this case, each Node will be a Docker Container. When you deploy an application on top of these nodes, Kubernetes will decide where the containers for the application will run based on the overall cluster utilization. Kubernetes will also deal with failures of these nodes to minimize your applications downtimes. Because you are running a local Kubernetes cluster, this has limitations, such as your laptop/pc available CPUs and Memory. In real life clusters, each of these nodes is a different physical or virtual machine that can be running in different locations to maximize resilience.

You can create the cluster by running the following command in the terminal:

```
cat <node-labels: "ingress-ready=true"
extraPortMappings:
- containerPort: 80
  hostPort: 80
  protocol: TCP
- containerPort: 443
  hostPort: 443
  protocol: TCP
- role: worker
- role: worker
- role: worker
EOF
```

You can copy the previous command and following commands from GitHub:
<https://github.com/salaboy/from-monolith-to-k8s/blob/main/kind/README.md>

Notice that besides creating a cluster you will also need to set up an Ingress Controller (hence the labels in the control plane node: `node-labels: "ingress-ready=true"`) and some port-mappings to route traffic from your laptop to the services running inside the cluster.

You should see something similar to Figure 2.3 after you run the previous command:

```
Creating cluster "dev"
✓ Ensuring node image (kindest/node:v1.23.4) 
✓ Preparing nodes 
✓ Writing configuration 
✓ Starting control-plane 
✓ Installing CNI 
✓ Installing StorageClass 
✓ Joining worker nodes 
Set kubectl context to "kind-dev"
You can now use your cluster with:
```

```
kubectl cluster-info --context kind-dev
```

Not sure what to do next? 😊 Check out
<https://kind.sigs.k8s.io/docs/user/quick-start/>.

Listing 2.3 KinD cluster created

```
To connect your `kubectl` CLI tool with this newly created, you must run:  
kubectl cluster-info --context kind-dev
```

You should see something similar to:

Listing 2.4 Setting the context for `kubectl`

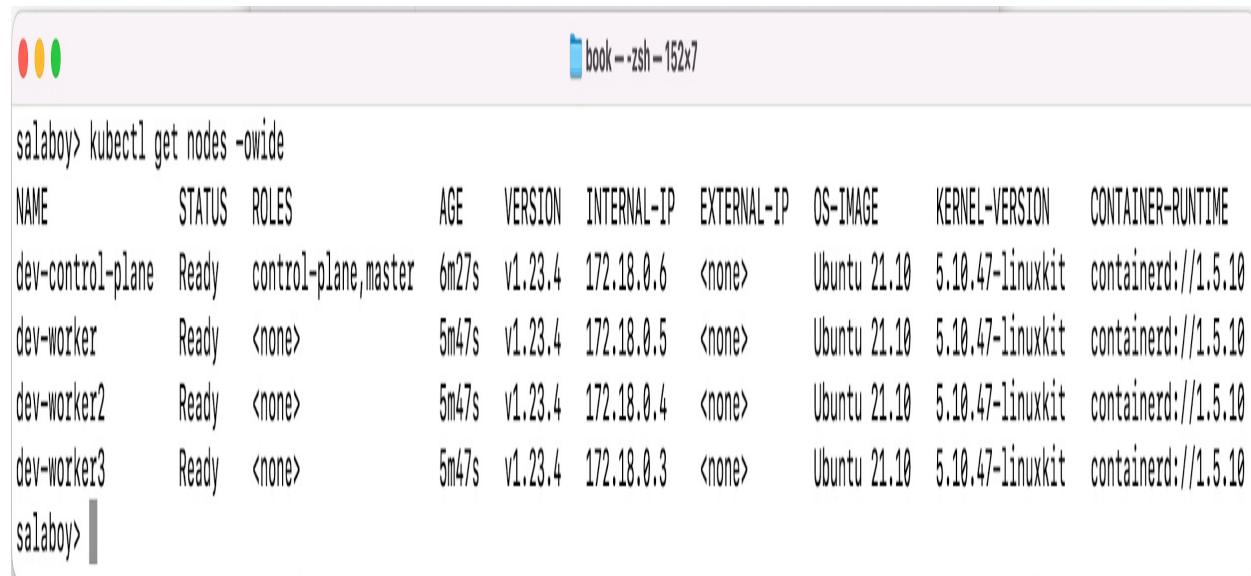
```
<pre class="codeacxspfirst">Kubernetes control plane  
</pre> <pre class="codeacxspmiddle">CoreDNS is running  
</pre> <pre class="codeacxspmiddle">&  
</pre> <pre class="codeacxsplast">To further debug  
</pre>
```

Once you have connected with the cluster you can start interacting with it. For example, you can check the cluster nodes by running:

```
kubectl get nodes -owide
```

The output of running that command should look similar to this:

Figure 2.5 Listing all Kubernetes Nodes



NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP	OS-IMAGE	KERNEL-VERSION	CONTAINER-RUNTIME
dev-control-plane	Ready	control-plane, master	6m27s	v1.23.4	172.18.0.6	<none>	Ubuntu 21.10	5.10.47-linuxkit	containerd://1.5.10
dev-worker	Ready	<none>	5m47s	v1.23.4	172.18.0.5	<none>	Ubuntu 21.10	5.10.47-linuxkit	containerd://1.5.10
dev-worker2	Ready	<none>	5m47s	v1.23.4	172.18.0.4	<none>	Ubuntu 21.10	5.10.47-linuxkit	containerd://1.5.10
dev-worker3	Ready	<none>	5m47s	v1.23.4	172.18.0.3	<none>	Ubuntu 21.10	5.10.47-linuxkit	containerd://1.5.10

As you can see, your Kubernetes Cluster is composed of four nodes, and one of those is the control plane. Notice that you are using the `--output wide` flag to get

more information about your nodes.

Finally, you will use NGINX Ingress Controller (more detailed instructions can be found here: (<https://kind.sigs.k8s.io/docs/user/ingress/>) to route traffic from outside the Kubernetes Cluster to the applications that are running inside the cluster. There are a number of Ingress Controllers implementations that you can install to do this routing, but NGINX Ingress Controller is widely adopted and the most popular option. For a non-extensive list of available options, you can check the Kubernetes website: <https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/>. To install the NGINX Ingress Controller you need to run the following command:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ing
```

This command creates a set of resources inside our Kubernetes Cluster required to run the NGINX Ingress Controller in a new Kubernetes Namespace called `ingress-nginx`:

Listing 2.6 Installing NGINX Ingress Controller

```
namespace/ingress-nginx created
serviceaccount/ingress-nginx created
serviceaccount/ingress-nginx-admission created
role.rbac.authorization.k8s.io/ingress-nginx created
role.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrole.rbac.authorization.k8s.io/ingress-nginx created
clusterrole.rbac.authorization.k8s.io/ingress-nginx-admission created
rolebinding.rbac.authorization.k8s.io/ingress-nginx created
rolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
configmap/ingress-nginx-controller created
service/ingress-nginx-controller created
service/ingress-nginx-controller-admission created
deployment.apps/ingress-nginx-controller created
job.batch/ingress-nginx-admission-create created
job.batch/ingress-nginx-admission-patch created
ingressclass.networking.k8s.io/nginx created
validatingwebhookconfiguration.admissionregistration.k8s.io/ingre
```

Listing 2.6 shows all the resources that were created inside the cluster to

install our Ingress Controller.

As a side note, you can check where this Ingress Controller is running in your cluster by running:

```
kubectl get pods -n ingress-nginx -owide
```

The output of this command should look like:

Listing 2.7 Ingress Controller running in `control-plane` node

NAME	READY	STATUS	R
ingress-nginx-admission-create-dwrgc	0/1	Completed	0
ingress-nginx-admission-patch-mm5jg	0/1	Completed	2
ingress-nginx-controller-8695d45448-p6v2w	1/1	Running	0

Here it can be seen that the Ingress Controller pod is running in the Control Plane node.

There you have it; your Cluster is up and running, and your `kubectl` command-line interface is configured to work against your new cluster! Now you are ready to install applications in your newly created cluster.

2.1.3 Installing the walking skeleton

To run containerized applications on top of Kubernetes you will need to have each of the services packaged as a container image, plus you will need to define how these containers will be configured to run in your Kubernetes cluster. To do so, Kubernetes allows you to define different kinds of resources (using YAML format) to configure how your containers will run and communicate with each other. The most common kinds of resources are:

- **Deployments:** declaratively define how many replicas of your container need to be up for your application to work correctly. Deployments also allows us to choose which container (or containers) do we want to run and how these containers need to be configured (using Environment Variables).
- **Services:** declaratively define a high-level abstraction to route traffic to the containers created by your deployments. It also acts as a load-

balancer between the replicas inside your deployments. Services enable other services and applications inside the cluster to use the service name instead of the physical IP address of the containers to communicate, providing what is known as Service Discovery.

- **Ingress:** declaratively define a route to route traffic from outside the cluster to services inside the cluster. By using Ingress definitions, we can only expose the services that are required by client applications that run outside the cluster.
- **ConfigMap/Secrets:** declaratively define and store configuration objects to set up our services instances. Secrets are considered sensitive information that should have protected access.

If you have large applications with tens of services, these YAML files are going to be complex and hard to manage. Keeping track of the changes and deploying applications by applying these files using `kubectl` becomes a complex job. It is beyond the scope of this book to cover an in-detail view of these resources, as there are other resources available. In this book, we will concentrate on how to deal with these resources for large applications and the tools that can help us with that task. The following section provides an overview about the tools that you can use to package and install components into your Kubernetes Cluster.

Packaging and Installing Kubernetes Applications

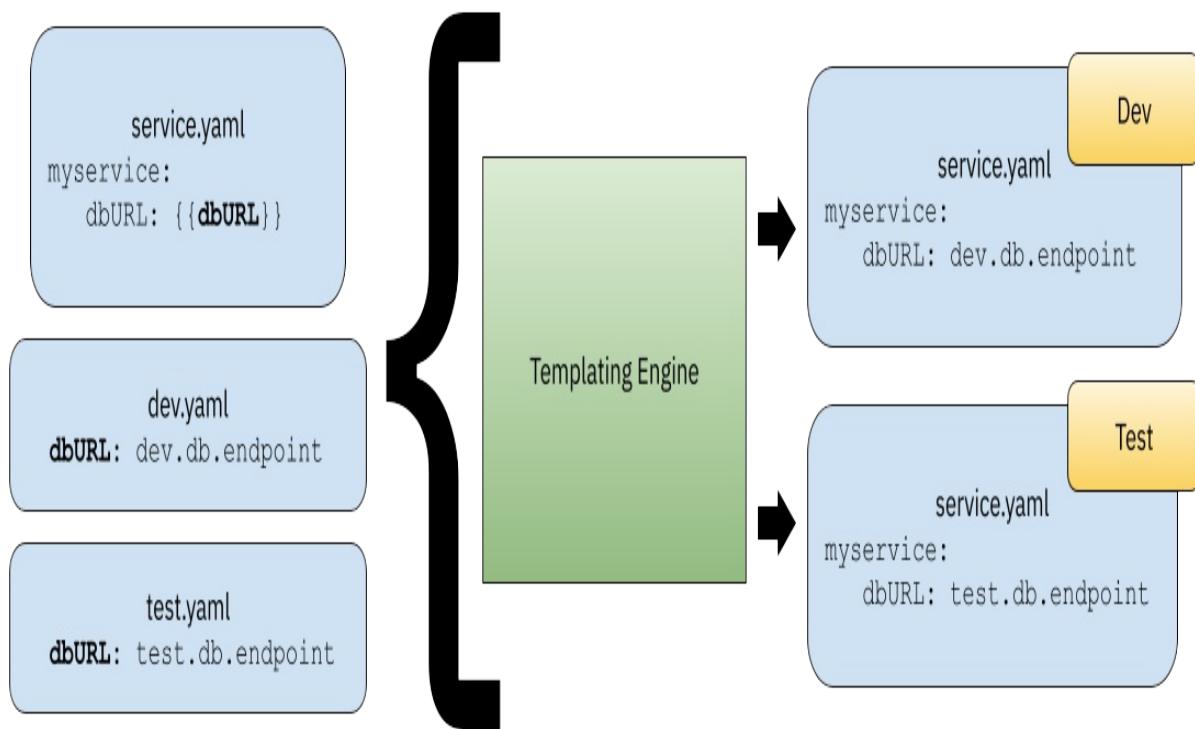
There are different tools to package and manage your Kubernetes applications. Most of the time we can separate these tools into two main categories: templating engines and package managers. For real-life scenarios, you will probably need both kinds of tools to get things done.

Let's talk a bit about these two kinds of tools: why would you need a templating engine? What kind of packages do you want to manage?

A templating engine allows you to reuse the same resource definitions into different environments where applications might require slightly different parameters. The textbook example for the need of templating your resources are database URLs. If your service needs to connect to different database instances in different environments, for example to the testing database in the

testing environment and to the production database in the production environment, you want to avoid having to maintain two copies of the same YAML file but with different URLs. Figure 2.x shows how you can now add variables into the YAML files and the engine then will find and replace these variables with different values depending where you want to use the final (rendered) resource.

Figure 2.x Templating engines render YAML resources by replacing variables



Using a templating engine can save you a lot of time on maintaining different copies of the same file and when files start to pile up maintaining them becomes a full-time job. There are several tools in the community to deal with templating Kubernetes files, some tools just deal with YAML files and some other tools are more targeted to Kubernetes resources specifically. Some projects that you should check out are:

- Kustomize: <https://kustomize.io/>
- Carvel YTT: <https://carvel.dev/ytt/>
- Helm Templates:

https://helm.sh/docs/chart_best_practices/templates/#helm

Now, what do you do with all these files? It is quite a natural urge to try to organize these files in logical packages. If you are building an application that is composed of different services it might make sense to group all the resources related to a Service inside the same directory, or even in the same repository that contains the source code for that service. You also want to make sure that you can distribute these files to the teams deploying these services to different environments, and you quickly realize that you need to version these files in some way. This versioning might be related to the version of your service itself or with a high-level logical aggregation that makes sense for your application. When we talk about grouping, versioning and distributing these resources we are basically describing the responsibility of a package manager. Developers and Operations teams are already used to working with package managers no matter the technology stack that they are using. Maven/Gradle for Java, NPM for NodeJS, APT-GET for Linux/Debian/Ubuntu packages, and more recently containers and container registries for Cloud-Native applications.

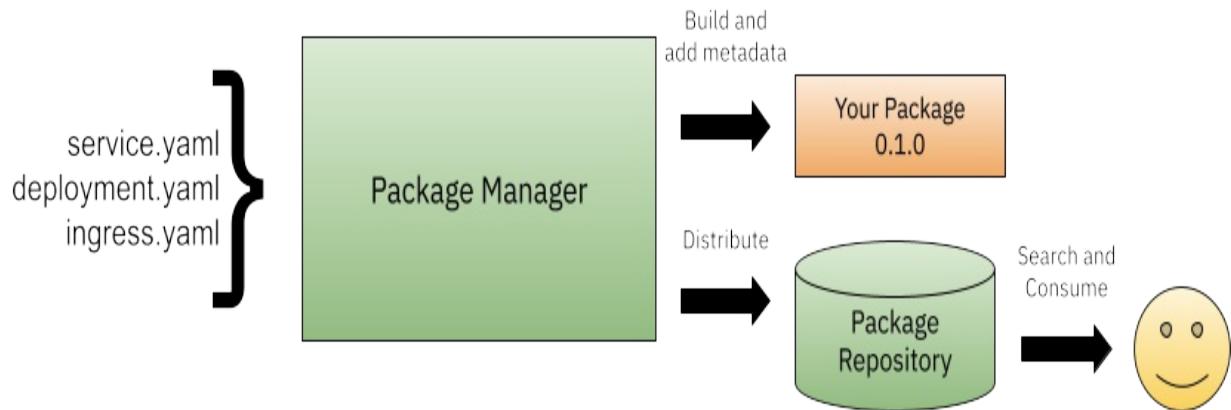
So what does a Package Manager for YAML files look like? What are the Package Manager's main responsibilities?

As a user, a package manager gives you a way to browse available packages and their metadata so you can decide which package you want to install. Once you have decided which package you want to use, you should be able to download it and then install it. Once the package is installed you would expect, as a user, to be able to upgrade to a newer version of the package when it becomes available. Upgrading/Updating a package is something that requires manual intervention, meaning that as a user you would explicitly tell the package manager to upgrade the installation of a certain package to a newer (or latest) version.

From a package provider's point of view, a package manager should offer a convention and structure to create packages and a tool to package the files that you want to distribute. Package managers deal with versions and dependencies, meaning that if you create a package you will need to associate a version number to it. Some package managers use the semver (semantic versioning) approach which uses 3 numbers to describe the package maturity

(1.0.1 where these numbers represent the major, minor, and patch versions). It is not mandatory for a package manager to provide a centralized package repository, but they often do. This package repository is in charge of hosting packages for users to consume. Central repositories are really useful as they provide access to developers with thousands of packages ready to be used, some examples of these central repositories are Maven Central, NPM, Docker Hub, and Github Container Registry, etc. These repositories are in charge of indexing the package's metadata (which can include versions, labels, dependencies, and short descriptions) to make them searchable by users. These repositories also deal with access control to have public and private packages, but at the end of the day, the main responsibility of the package repository is to allow package producers to upload packages and package consumers to download packages from it.

Figure 2.x Package Managers' responsibilities: build, package and distribute



When we talk about Kubernetes Helm is a very popular tool that provides both a package manager and a templating engine. But there are others worth looking into, such as:

- Imgpkg (<https://carvel.dev/imgpkg/>) which uses Container registries to store the packages
- Kapp (<https://carvel.dev/kapp/>) which provides higher-level abstractions to group resources as applications
- And tools like Terraform, Pulumi, and Ansible allow you to create packages closer to the infrastructure

In the following section, we will look into how Helm (<http://helm.sh>) can help us to package, distribute and manage our Kubernetes resources.

Installing the walking skeleton using Helm

In this section, we are going to use Helm, a package manager for Kubernetes applications to install our Walking Skeleton into our freshly created Kubernetes Cluster. As we installed an Ingress Controller, we should be able to access the application from our Laptop's favorite browser.

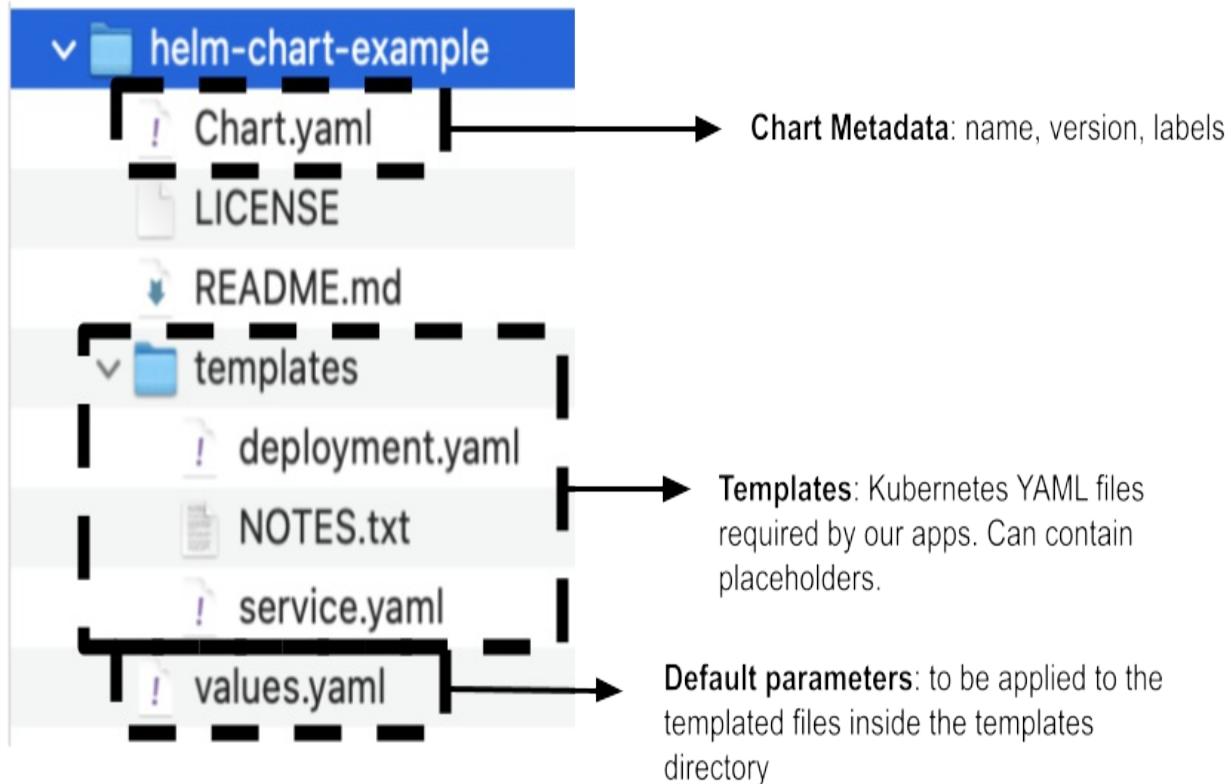
Helm Basics

Helm was created to package all YAML files from a service or an entire application into packages called Charts.

To use Helm you create one of these charts (packages). A Helm Chart is defined by a set of files organized using a very specific directory structure. You can version these Charts to deal with configuration changes and new versions of your application/service.

As you can see in the following figure, a `Chart.yaml` file is required to define the Chart metadata such as name and version. The `templates` directory contains all of our YAML files required to deploy and configure our service/application. As the name of the directory indicates, the files inside the `templates` directory can include parameterizable values that you can replace when you are installing the chart into a specific environment. Finally, the `values.yaml` file contains the default values for the parameterizable placeholders included in the templates. When installing a chart you can provide your own `values.yaml` file to override the defaults.

Figure 2.8 Simple Helm Chart



Helm also provides a command-line tool (`helm`) to package, search and install these packages. Helm Charts can be stored in “Helm Repositories” and distributed for other users to use. The most commonly used `helm` command that you need to learn how to use is `helm install <release name> <chart name>`. This will install the chart into a Kubernetes Cluster. In which cluster you might be wondering? Helm uses the same configuration used by `kubectl` to interact with the Kubernetes APIs, hence if you can connect with `kubectl` to your cluster Helm will be able to install charts in that cluster.

You can check out each of the files for this example chart here:
<https://github.com/salaboy/helm-chart-example> The `README.md` file also includes how to run the most common operations to package and install the chart in your own Kubernetes Cluster.

When you install a Chart into a Kubernetes, Helm will create a Release, that we can upgrade at any time if a new version of the chart is available. Helm will keep track of these releases, allowing us to roll back to previous releases if something goes wrong with a newer version of your application.

To install Helm Charts (packages/applications) you can add new repositories in the same location as where your applications are stored. For java developers, these repositories are like Maven Central, Nexus or Artifactory.

```
<pre class="codeacxspfirst">&gt;helm repo add nbs
</pre> <pre class="codeacxsplast">&gt;helm repo repo&nbs;u
</pre>
```

You should see the following output:

Listing 2.9 Adding custom Helm repository

```
> helm repo add fmtok8s https://salaboy.github.io/helm/
"fmtok8s" has been added to your repositories
> helm repo update
Hang tight while we grab the latest from your chart repositories.
...Successfully got an update from the "fmtok8s" chart repository
Update Complete. *Happy Helming!*
```

The previous two lines added a new repository to your Helm installation called **fmtok8s**; the second one fetched a file describing all the available packages and their versions for each repo that you have registered. Now that you have installed a new repository, you don't need to install the chart from the chart source code and you can use the published version in the `https://salaboy.github.io/helm/` repository. Notice that Helm Chart Repositories can be created inside Github for small setups as I am doing in the following repository: <https://github.com/salaboy/helm>. Check the repository README.md for more details about how this is working.

Before jumping into installing our walking skeleton it is also important to know that Helm also provides dependency management between these packages, meaning that you can define that a Chart depends on one or more Charts and Helm will download and install these dependent charts when you install your (parent) Chart. This allows us to install multiple services and other components at the same time without the need to package all the YAML files together. You can define dependencies by adding a section to the Chart.yaml file, for example:

```
dependencies:
- name: postgresql
```

```
repository: https://charts.bitnami.com/bitnami
version: 10.8.0
```

If you are using dependencies, make sure that before `helm package` you run `helm dependency build` to fetch these charts. This will make your chart package standalone and not depend on other Chart repositories to be available for your application to work.

Now, let's jump to install our walking skeleton.

Installing the Conference application with a single command

Now that your Helm installation fetched all the available packages from the **fmtok8s** Chart repository, you are ready to install the Conference Platform application, which was introduced in Chapter 1, Section X. This Conference Platform allows conference organizers to receive proposals from potential speakers, evaluate these proposals and keep an updated agenda with the approved submissions for the event. We will be using this application throughout the book to exemplify the challenges that you will face while building real-life applications. This application was built as a walking skeleton, which means it is not a complete application but has all the pieces required for some use cases to work and these pieces can be iterated further to support real-life scenarios. In the following sections, you will install the application into the cluster and interact with it to see how it behaves when it runs on top of Kubernetes.

Let's install the application with the following line:

```
> helm install conference fmtok8s/fmtok8s-conference-chart
```

You should see the following output:

Listing 2.10 Helm installed the chart fmtok8s-app version 0.1.0

```
NAME: conference
LAST DEPLOYED: Wed Jun 22 16:04:36 2022
NAMESPACE: default
STATUS: deployed
REVISION: 1
```

TEST SUITE: None

NOTES:

Cloud-Native Conference Platform V1

Chart Deployed: fmtok8s-conference-chart - v0.1.0

Release Name: conference

`helm install` creates a Helm Release, which means that you have created an application instance, in this case, the instance is called `conference`. With Helm, you can deploy multiple instances of the application if you want to. You can list Helm releases by running:

```
> helm list
```

The output should look like this:

Listing 2.11 List Helm releases

NAME	NAMESPACE	REVISION	UPDATED
conference	default	1	2022-06-22 16:07:02.129083 +0100 BST

Note:

If instead of using `helm install` you run `helm template fmtok8s/fmtok8s-conference-chart` Helm will output the YAML files which will apply against the cluster. There are situations where you might want to do that instead of `helm install`, for example, if you want to override values that the Helm charts don't allow you to parameterize or apply any other transformations before sending the request to Kubernetes.

Verifying that the application is up and running

Once the application is deployed, containers will be downloaded to your laptop to run, and this can take a while. You can monitor the progress by listing all the pods running in your cluster, once again, using the `--owide` flag to get more information:

```
> kubectl get pods --owide
```

The output should look like:

Listing 2.12 Listing application Pods

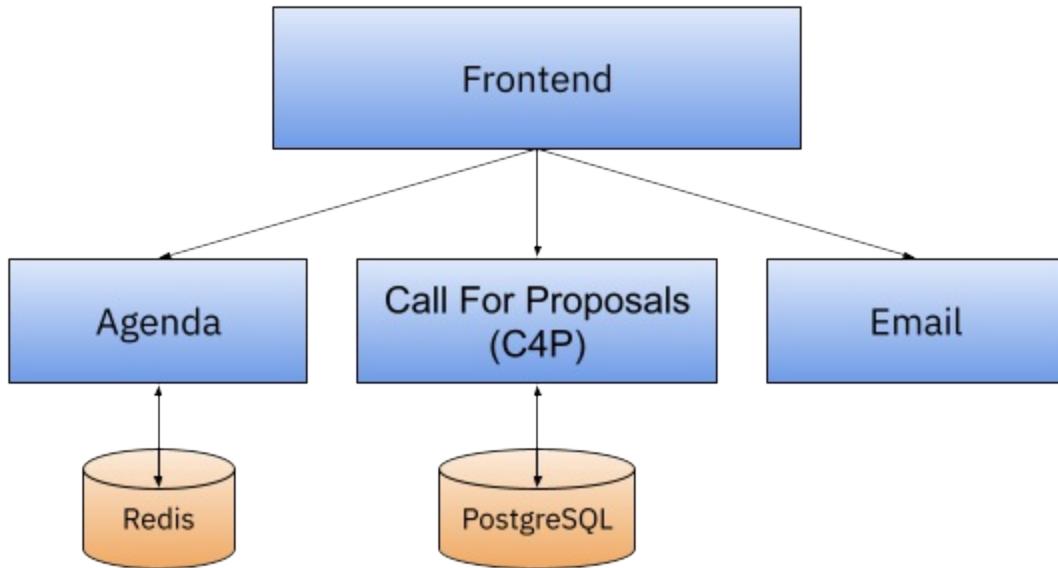
NAME	READY	STAT
conference-fmtok8s-agenda-service-57576cb65c-hzfxv	1/1	Runn
conference-fmtok8s-c4p-service-6c6f9449b5-z22jk	1/1	Runn
conference-fmtok8s-email-service-6fdf958bdd-8622z	1/1	Runn
conference-fmtok8s-frontend-5bf68cf65-s7rn2	1/1	Runn
conference-postgresql-0	1/1	Runn
conference-redis-master-0	1/1	Runn
conference-redis-replicas-0	1/1	Runn

Something that you might notice in the list of pods is that we are not only running the application's services, but we are also running Redis and PostgreSQL as the C4P and Agenda services need persistent storage. Besides the services, we will have these two databases running inside our Kubernetes Cluster.

On Listing 2.x, you need to pay attention to the READY and STATUS columns, where 1/1 in the READY column means that one replica of the `Pod` is running and one is expected to be running. As you can see the RESTART column is showing 3 for the Call for proposals Service (`fmtok8s-c4p-service`), this is due to the fact that the service depends on Redis to be up and running for the service to be able to connect to it. While Redis is bootstrapping the application will try to connect and if it fails it will automatically restart to try again, as soon as Redis is up the service will connect to it.

Both the `fmtok8s-agenda-service` and `fmtok8s-c4p-service` Helm charts can be configured to not create these databases if we want to connect our services with existing databases outside our Kubernetes cluster. To quickly recap, our application services and the databases that we are running look like figure 2.x:

Figure 2.x Application Services and databases



Notice that `Pod`s can be scheduled in different nodes. You can check this in the `NODE` column; this is Kubernetes efficiently using the cluster resources.

If all the `Pod`s are up and running, you've made it! The application is now up and running, and you can access it by pointing your favorite browser to: <http://localhost>

If you are interested in Helm and how you can build your own Conference Application Helm chart, I recommend you to check the tutorial which you can find here: <https://github.com/salaboy/from-monolith-to-k8s/tree/main/helm>.

2.1.4 Interacting with your application

In the previous section, we installed the application into our local Kubernetes Cluster. In this section we will quickly interact with the application to understand how the services are interacting to accomplish a simple use case: “Receiving and approving Proposals”. Remember that you can access the application by pointing your browser to: <http://localhost>

The Conference Platform application should look like the following screenshot:

Figure 2.X Conference main page



Agenda Proposals About



Cloud Conference 2025

O2 Arena
London.
Aug 7th/9th

Join us this year to the 10th anniversary of
the Cloud Conference! More about the
latest and greatest in the industry.

If you switch to the Agenda section now you should see something like this:

Figure 2.X Conference empty Agenda when we first install the application



Agenda Proposals About



Conference Agenda

Monday's sessions

There are no confirmed talks just yet.

Tuesday's sessions

There are no confirmed talks just yet.

The application's Agenda Page lists all the talks scheduled for the conference. Potential speakers can submit proposals that will be reviewed by the conference organizers. When you start the application for the first time, there will be no talks on the agenda, but you can now go ahead and submit a proposal from the Proposals section.

Figure 2.X Submitting a proposal for organizers to review



Agenda Proposals About



Submit your proposal

Title

Author

Email

Abstract

Generate

Join us as a speaker

Are you passionate about Cloud,
Kubernetes, Docker or other
technologies related with the Cloud.
Submit your proposal to share your
knowledge with our amazing
community!

Notice that there are four fields (Title, Author, Email and Abstract) in the form that you need to fill to submit a proposal. The application currently offers a “Generate” button to create random content that you can submit. The organizers will use this information to evaluate your proposal and get in touch with you via email if your proposal gets approved or rejected. Once the proposal is submitted, you can go to the `Back Office` and `Approve` or `Reject` submitted proposals. You will be acting as a conference organizer on this screen:

Figure 2.X Conference organizers can Accept or Reject incoming proposals



Back to Site

Welcome Conference Organizers

Features

Proposals

Proposals to Review

All

Pending

minim deserunt labore in non
laborum Lorem consectetur@mail.com

PENDING

Approve

Reject

Fugiat Lorem consequat officia esse magna sint minim non eiusmod
cupiditat sint ullamco laboris duis occaecat. Anim sit veniam aliquip

Accepted proposals will appear on the Main Page. Attendees who visit the page at this stage can have a glance at the conference's main speakers.

Figure 2.X Your proposal is now live on the agenda!



[Agenda](#) [Proposals](#) [About](#)



Conference Agenda

Monday's sessions

Monday

2:00 pm

minim deserunt labore in non

laborum Lorem

Tuesday's sessions

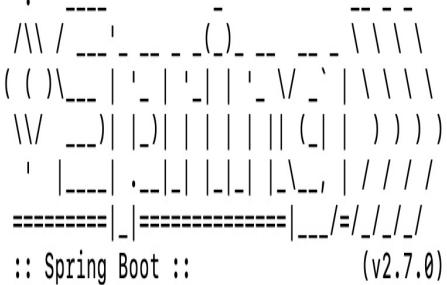
There are no confirmed talks just yet.

At this stage, the potential speaker should have received an email about the approval or rejection of his/her proposal. You can check this by looking at the Email Service logs, using `kubectl` from your terminal:

```
> kubectl logs -f conference-fmtok8s-email-service-<POD_ID>
```

Figure 2.X Email Service logs

```
|salaboy> kubectl logs -f conference-fmtok8s-email-service-6fdf958bdd-cvvjh
AOT mode enabled
```



```
Starting EmailServiceApplication using Java 17.0.3.1 on conference-fmtok8s-email-service-6fdf958bdd-cvvjh with PID 1
(/workspace/com.salaboy.conferences.email.EmailServiceApplication started by cnb in /workspace)
```

```
The following 1 profile is active: "dev"
```

```
GC notifications will not be available because MemoryPoolMXBeans are not provided by the JVM
```

```
Exposing 5 endpoint(s) beneath base path '/actuator'
```

```
Netty started on port 8080
```

```
Started EmailServiceApplication in 0.172 seconds (JVM running for 0.176)
```

```
> REST ENDPOINT INVOKED for Sending an Email Notification about a proposal from: consectetur@mail.com
```

```
+-----+
```

```
Email Sent to: consectetur@mail.com
```

```
Email Title: Conference Committee Communication
```

```
Email Body: Dear laborum Lorem,
```

```
We are happy to inform you that:
```

```
'minim deserunt labore in non' -> 'Fugiat Lorem consequat officia esse magna sint minim non  
eiusmod cupidatat sint ullamco laboris duis occaecat. Anim sit veniam aliquip commodo aute incididunt non. Nostrud el  
it sit nostrud occaecat enim. Sit occaecat voluptate pariatur fugiat et commodo deserunt nisi dolor elit aliqua. Eu c  
illum commodo laborum duis occaecat labore ex. Incididunt consequat do et eiusmod Lorem et nisi officia veniam aute.  
Lorem commodo exercitation do tempor amet tempor nisi enim ex elit esse nostrud.'
```

```
Cupidatat deserunt aute sunt laborum est aute ipsum reprehenderit excepteur eu tempor. Incididunt deserunt ullamco ex  
ercitation consectetur sit cupidatat duis veniam nulla labore irure nisi. Fugiat adipisicing ipsum ipsum offic  
ia esse dolore consequat qui illum est incididunt adipisicing occaecat. Culpa esse occaecat fugiat. Adipisicing ame  
t eiusmod Lorem excepteur aliquip ut mollit officia ullamco ipsum mollit laborum nisi excepteur amet.'
```

```
was approved for this conference.
```

```
+-----+
```

```
>     EventsEnabled: false
```

If you made it so far, congrats, the Conference application is working as expected. I encourage you to submit another proposal and reject it, to validate that the correct email is being sent to the potential speaker.

In this section you have installed the Conference application using Helm, then verified that the application is up and running and that the platform can be used by potential speakers and conference organizers to submit proposals, approve or reject these proposals and notify potential speakers about these decisions via email.

This simple application allows us to demonstrate a basic use case that now we can expand and improve to support real users. We have seen that installing a new instance of the application is quite simple, we have used Helm to install a set of services that are connected together as well as some infrastructural components such as Redis and PostgreSQL and now in the next section we will go deeper into understand what we have installed and how the application is working.

2.2 Inspecting the walking skeleton

If you have been using Kubernetes for a while, you probably know all about `kubectl`. Because this version of the application uses native Kubernetes Deployments and Services, you can inspect and troubleshoot these Kubernetes resources using `kubectl`.

Usually, instead of just looking at the Pods running (with kubectl get pods), to understand and operate the application, you will be looking at Services and Deployments. Let's explore the Deployment resources first.

2.2.1 Kubernetes Deployments basics

Let's start with Deployments. Deployments in Kubernetes are in charge of containing the recipe for running our containers. Deployments are also in charge of defining how containers will run and how they will be upgraded to newer versions when needed. By looking at the Deployment details, you can get very useful information, such as

- The **container** that this deployment is using: notice that this is just a simple docker container, meaning that you can even run this container locally if you want to with `docker run`. This is fundamental to troubleshooting problems.
- The number of **replicas** required by the deployment: for this example is set to 1, but you will change this in the next section. Having more replicas adds more resiliency to the application, as these replicas can go down. Kubernetes will spawn new instances to keep the number of desired replicas up at all times.
- The **resources allocation** for the container: depending on the load and the technology stack that you used to build your service, you will need to fine-tune how many resources Kubernetes allow your container to use.
- The status of the ‘**readiness**’ and ‘**liveness**’ **probes**: Kubernetes by default, will monitor the health of your container. It does that by executing two probes: 1) The `readiness probe` checks if the container is ready to answer requests 2) The `liveness probe` checks if the main process of the container is running.
- The rolling updates strategy defines how our Pods will be updated to avoid downtime to our users. With the `RollingUpdateStrategy` you can define how many replicas are allowed to go down while triggering and update to a newer version.

First, let’s list all the available Deployments with:

```
> kubectl get deployments
```

With an output looking like this:

Listing 2.X Listing your application’s Deployments

NAME	READY	UP-TO-DATE	AVAILABL
conference-fmtok8s-agenda-service	1/1	1	1
conference-fmtok8s-c4p-service	1/1	1	1
conference-fmtok8s-email-service	1/1	1	1
conference-fmtok8s-frontend	1/1	1	1

Exploring Deployments

In the following example, you will describe the Frontend deployment. You can describe each deployment in more detail with:

Listing 2.X Describing a deployment to see its details

```
> kubectl describe deployment conference-fmtok8s-frontend
Name:                  conference-fmtok8s-frontend
Namespace:              default
CreationTimestamp:      Sat, 02 Jul 2022 09:20:04 +0100
Labels:                app.kubernetes.io/managed-by=Helm
                        chart=fmtok8s-frontend-v0.1.1
                        draft=draft-app
Annotations:            deployment.kubernetes.io/revision: 1
                        meta.helm.sh/release-name: conference
                        meta.helm.sh/release-namespace: default
Selector:               app=conference-fmtok8s-frontend
Replicas:               1 desired | 1 updated | 1 total | 1 avail
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:    app=conference-fmtok8s-frontend
             draft=draft-app
  Containers:
    fmtok8s-frontend:
      Image:      ghcr.io/salaboy/fmtok8s-frontend:v0.1.1 #B
      Port:       8080/TCP
      Host Port: 0/TCP
      Limits:    #C
        cpu:     1
        memory: 256Mi
      Requests:
        cpu:     100m
        memory: 256Mi
      Liveness:  http-get http://:8080/actuator/health delay=60s ti
      Readiness: http-get http://:8080/actuator/health delay=0s ti
      Environment:
...
  NewReplicaSet:  conference-fmtok8s-frontend-58889887cf (1/1 repl
Events: #D
  Type      Reason          Age    From            Message
  ----      ----          ----   ----            -----
  Normal   ScalingReplicaSet 21m   deployment-controller  Scaled
```

Listing 2.x shows that describing deployments in this way is very helpful if for some reason the deployment is not working as expected. For example, if

the number of replicas required is not met, describing the resource will give you insights into where the problem might be. Always check at the bottom for the events associated with the resource to get more insights about the resource status, in this case, the deployment was scaled to have 1 replica 21 minutes ago.

As mentioned before, Deployments are also responsible for coordinating version or configuration upgrades and rollbacks. The deployment update strategy is set by default to “Rolling Update”, which means that the deployment will incrementally upgrade Pods one after the other to minimize downtime. An alternative strategy can be set called “Recreate” which will shut down all the Pods and create new ones.

In contrast with Pods, Deployments are not ephemeral; hence if you create a `Deployment` it will be there for you to query no matter if the containers under the hood are failing. By default, when you create a Deployment resource, Kubernetes creates an intermediate resource for handling and checking the Deployment requested replicas.

ReplicaSets

Having multiple replicas of your containers is an important feature to be able to scale your applications. If your application is experiencing loads of traffic from your users, you can easily scale up the number of replicas of your services to accommodate all the incoming requests. In a similar way, if your application is not experiencing a large number of requests these replicas can be scaled down to save resources. The object created by Kubernetes is called `ReplicaSet`, and it can be queried by running:

```
> kubectl get replicaset
```

The output should look like:

Listing 2.X Listing Deployment's Replica Sets

NAME	DESIRED	CURRENT
conference-fmtok8s-agenda-service-57576cb65c	1	1
conference-fmtok8s-c4p-service-6c6f9449b5	1	1

conference-fmtok8s-email-service-6fdf958bdd	1	1
conference-fmtok8s-frontend-58889887cf	1	1

These `ReplicaSet` objects are fully managed by the `Deployment` resource, and usually, you shouldn't need to deal with them.

ReplicaSets are also essential when dealing with Rolling Updates, and you can find more information about this topic here:

<https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/>.

You will be performing updates to the application with Helm in later chapters where these mechanisms will kick in.

If you want to change the number of replicas for a deployment you can use once again `kubectl` to do so:

```
kubectl scale --replicas=2 deployments/<DEPLOYMENT_ID>
```

You can try this out with the Frontend deployment:

```
kubectl scale --replicas=2 deployments/conference-fmtok8s-fronten
```

If we now list the application pods we will see that there are two replicas for the frontend service:

conference-fmtok8s-frontend-58889887cf-dq71k	0/1	Cont
conference-fmtok8s-frontend-58889887cf-hb6sc	1/1	Runn

This command changes the deployment resource in Kubernetes and triggers the creation of a second replica for the Frontend deployment. Increasing the number of replicas of your user-facing services is quite common as it is the service that all users will hit when visiting the conference page.

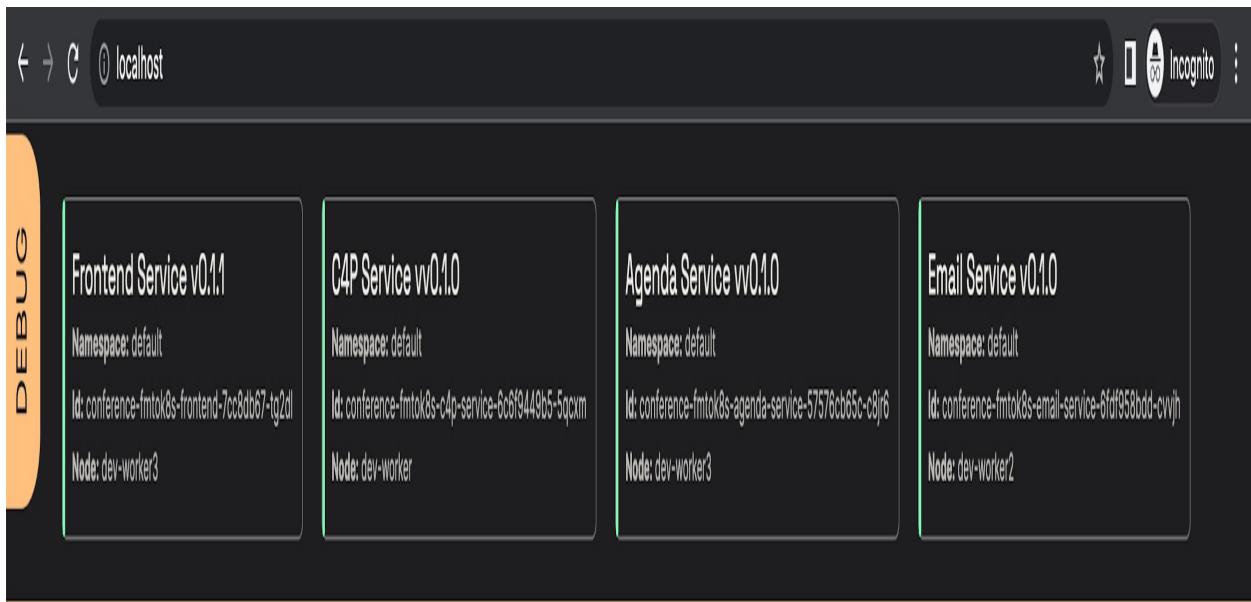
If we access the application right now, as end users we will not notice any difference, but everytime we refresh we might be served by a different replica. To make this more evident we can turn on a feature that is built into the Frontend service which shows us more information about the application containers. You can enable this feature by setting an environment variable:

```
> kubectl set env deployment/conference-fmtok8s-frontend FEATURE_
```

Notice that when you change the deployment object configuration (anything inside `spec.template.spec`) the rolling update mechanism will kick in, so all the existing pods managed by this deployment will be upgraded to have the new specification, but this upgrade, by default, will happen by starting a new pod at the time and as soon as it is ready an old version will be terminated and the next pod will be created. This process will be repeated until all the pods are using the new configuration.

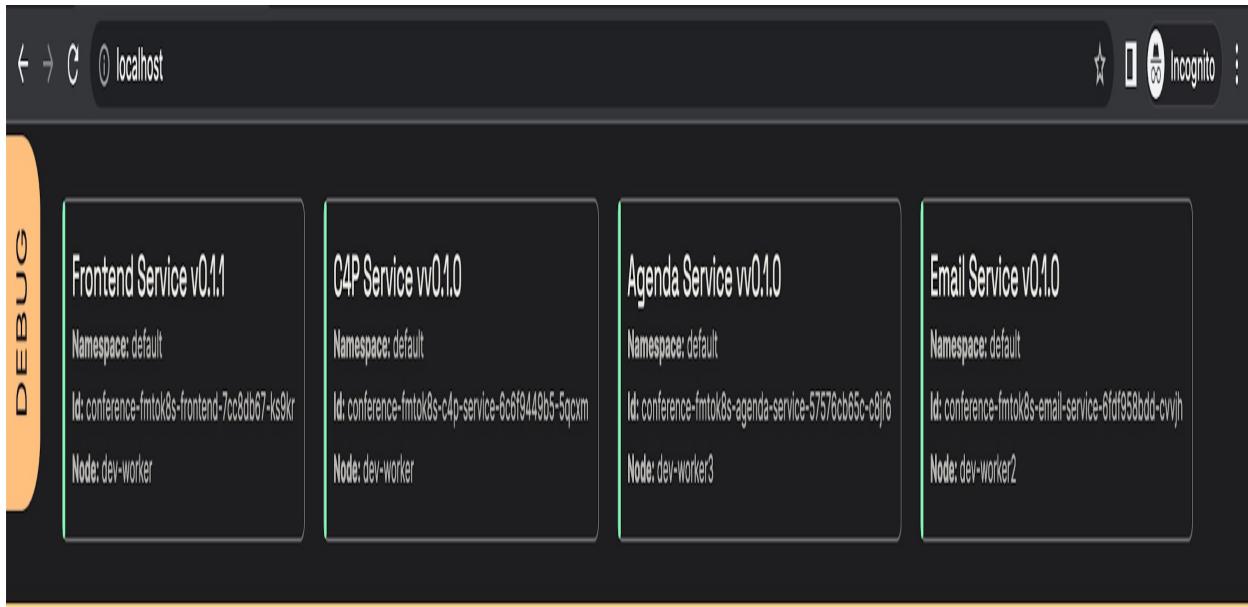
If you access the application again in your browser (you might need to access using Incognito Mode as the browser might have cached the website), at the top of the screen, there is a new Debug banner, you can see the Pod Id and the Node where the Pod is running:

Figure 2.X First replica answering your request



If you now refresh the page, the second replica should answer your request:

Figure 2.X Second replica is now answering; this replica might be on a different node



By default, Kubernetes will load-balance the requests between the replicas. Being able to scale by just changing the number of replicas, there is no need to deploy anything new, Kubernetes will take care of provisioning a new Pod (with a new container in it) to deal with more traffic. Kubernetes will also make sure that there is the amount of desired replicas at all times. You can test this by deleting one pod and watching how Kubernetes recreates it automatically. For this scenario you need to be careful, as the web application frontend is executing several requests to fetch the HTML, CSS and JavaScript libraries, hence each of these requests can land in a different replica.

2.2.2 Connecting services together

We have looked at Deployments, which are in charge of getting our containers up and running and keeping them that way, but so far, these containers can only be accessed inside the Kubernetes Cluster. If we want other services to interact with these containers we need to look at another Kubernetes resource called “Service”. Kubernetes provides an advanced Service discovery mechanism that allows services to communicate with each other by just knowing their names. This is essential to connect many services without the need to use IP addresses of containers that can change over time.

Exploring Services

To expose your containers to other services, you need to use a `Kubernetes Service` resource. Each application service defines this `Service` resource, so other services and clients can connect to them. In Kubernetes, Services will be in charge of routing traffic to your application containers. These `Service`s represent a logical name that you can use to abstract where your containers are running. If you have multiple replicas of your containers, the Service resource will be in charge of load balancing the traffic among all the replicas.

You can list all the services by running:

```
kubectl get services
```

After running the command, you should see something like figure 2.X that follows:

Listing 2.X Listing application's services

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
conference-postgresql	ClusterIP	10.96.238.72	<none>
conference-postgresql-h1	ClusterIP	None	<none>
conference-redis-headless	ClusterIP	None	<none>
conference-redis-master	ClusterIP	10.96.96.14	<none>
conference-redis-replicas	ClusterIP	10.96.68.41	<none>
fmtok8s-agenda	ClusterIP	10.96.233.3	<none>
fmtok8s-c4p	ClusterIP	10.96.181.242	<none>
fmtok8s-email	ClusterIP	10.96.186.144	<none>
fmtok8s-frontend	ClusterIP	10.96.47.19	<none>
kubernetes	ClusterIP	10.96.0.1	<none>

And you can also describe a Service to see more information about it with:

```
kubectl describe service fmtok8s-frontend
```

This should give you something, like we see in Listing 2.X. Services and Deployments, are linked by the Selector property, highlighted in the following image. In other words, the Service will route traffic to all the Pods created by a Deployment containing the label **app=conference-fmtok8s-frontend**.

Listing 2.X Describing the Frontend Service

```
Name:           fmtok8s-frontend
Namespace:      default
Labels:         app.kubernetes.io/managed-by=Helm
                chart=fmtok8s-frontend-v0.1.1
                draft=draft-app
Annotations:    meta.helm.sh/release-name: conference
                meta.helm.sh/release-namespace: default
Selector:       app=conference-fmtok8s-frontend #A
Type:          ClusterIP
IP Family Policy: SingleStack
IP Families:   IPv4
IP:            10.96.47.19
IPs:           10.96.47.19
Port:          http  80/TCP
TargetPort:    8080/TCP
Endpoints:     10.244.1.13:8080, 10.244.3.12:8080
Session Affinity: None
Events:
```

Service Discovery in Kubernetes

By using Services, if your application service needs to send a request to any other service it can use the Kubernetes Services name and port, which in most cases, you can use port 80 if you are using HTTP requests, leaving the need to only use the Service name.

If you look at the source code of the services, you will see that HTTP requests are created against the service name, no IP addresses or Ports are needed.

Finally, if you want to expose your Services outside of the Kubernetes Cluster, you need an Ingress resource. As the name represents, this Kubernetes resource is in charge of routing traffic from outside the cluster to services that are inside the cluster. Usually, you will not expose multiple services, limiting the entry points for your applications.

You can get all the available Ingress resources by running the following command:

```
kubectl get ingress
```

The output should look like:

Listing 2.X Listing application's Ingress Resources

NAME	CLASS	HOSTS	ADDRESS	PORTS	AGE
frontend	<none>	*	localhost	80	67m

And then you can describe the Ingress resource in the same way as you did with other resource types to get more information about it:

```
kubectl describe ingress frontend
```

You should expect the output to look like this:

Listing 2.X Describing Ingress Resource

```
Name: frontend
Namespace: default
Address: localhost
Default backend: default-http-backend:80 ()
Rules:
  Host      Path  Backends
  ----      ---  -----
  *
           /   fmtok8s-frontend:80 (10.244.1.13:8080,10.244.3.
Annotations: kubernetes.io/ingress.class: nginx
               meta.helm.sh/release-name: conference
               meta.helm.sh/release-namespace: default
               nginx.ingress.kubernetes.io/rewrite-target: /
Events:    < none >
```

As you can see, Ingresses also uses Services name to route traffic. Also, for this to work, you need to have an Ingress Controller, as we installed when we created the KinD Cluster. If you are running in a Cloud Provider you might need to install an Ingress controller.

The following spreadsheet is a community resource created to keep track of the different options of Ingress Controllers that are available for you to use:

<https://docs.google.com/spreadsheets/u/2/d/191WWNpjJ2za6-nbG4ZoUMXMpUK8KIClosvQB0f-oq3k/edit#gid=907731238>

With Ingresses you can configure a single entry-point and use path-based routing to redirect traffic to each service you need to expose. The previous Ingress resource in Listing 2.X routes all the traffic sent to `/` to the

`fmtok8s-frontend` service. Notice that Ingress rules are pretty simple and you shouldn't add any business logic routing at this level.

Troubleshooting internal Services

Sometimes, it is important to access internal services to debug or troubleshoot services that are not working. For such situations, you can use the `kubectl port-forward` command to temporarily access services that are not exposed outside of the cluster using an Ingress resource. For example, to access the Agenda Service without going through the Frontend you can use the following command:

```
kubectl port-forward svc/fmtok8s-agenda 8080:80
```

You should see the following output, make sure that you don't kill the command:

Listing 2.X `kubectl port-forward` allows you to expose a service for debugging purposes

```
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
```

And then using your browser, curl in a different tab or any other tool to point to `http://localhost:8080/info` to access the exposed Agenda Service. The following image shows how you can `curl` the Agenda Service info endpoint and print a pretty/colorful JSON payload with the help of `jq`, which you need to install separately.

Listing 2.X curl localhost:8080 to access Agenda Service using `port-forward`

```
curl -s localhost:8080/info | jq --color-output
{
  "name": "Agenda Service",
  "version": "v0.1.0",
  "source": "https://github.com/salaboy/fmtok8s-agenda-service/re",
  "podId": "conference-fmtok8s-agenda-service-57576cb65c-c8jr6",
  "podNamespace": "default",
  "podNodeName": "dev-worker3"
}
```

In this section, you have inspected the main Kubernetes Resources that were created to run your application's containers inside Kubernetes. By looking at these resources and their relationships, you will be able to troubleshoot problems when they arise.

For everyday operations, the `kubectl` command line tool might not be optimal and different dashboards can be used to explore and manage your Kubernetes workloads such as: k9s, Kubernetes Dashboard and Octant. You can check Appendix X where the same application is explored using Octant.

2.3 Cloud-Native applications Challenges

In contrast to a Monolithic application, which will go down entirely if something goes wrong, Cloud-Native applications shouldn't crash if a service goes down. Cloud-Native applications are designed for failure and should keep providing valuable functionality in the case of errors. A degraded service while fixing issues is better than having no access to the application at all. In this section, you will change some of the service configurations in Kubernetes to understand how the application will behave in different situations.

In some cases, application/service developers will need to make sure that they build their services to be resilient and some concerns will be solved by Kubernetes or the infrastructure.

This section covers some of the most common challenges associated with Cloud-Native applications. I find it useful to know what are the things that are going to go wrong in advance rather than when I am already building and delivering the application. This is not an extensive list, it is just the beginning to make sure that you don't get stuck with problems that are widely known. The following sections will exemplify and highlight these challenges with the Conference platform.

- **Downtime is not allowed:** If you are building and running a Cloud-Native application on top of Kubernetes and you are still suffering from application downtime, then you are not capitalizing on the advantages of the technology stack that you are using.

- **Service's built-in resiliency:** downstream services will go down and you need to make sure that your services are prepared for that. Kubernetes helps with dynamic Service Discovery, but that is not enough for your application to be resilient.
- **Dealing with the application state is not trivial:** we have to understand each service infrastructural requirements to efficiently allow Kubernetes to scale up and down our services.
- **Inconsistent data:** a common problem of working with distributed applications is that data is not stored in a single place and tends to be distributed. The application will need to be ready to deal with cases where different services have different views of the state of the world.
- **Understanding how the application is working (monitoring, tracing and telemetry):** having a clear understanding on how the application is performing and that it is doing what it is supposed to be doing is essential to quickly find problems when things go wrong.
- **Application Security and Identity Management:** dealing with users and security is always an after-thought. For distributed applications, having these aspects clearly documented and implemented early on will help you to refine the application requirements by defining “who can do what and when”.

Let's start with the first of the challenges: Downtime is not allowed.

2.3.1 Downtime is not allowed

When using Kubernetes we can easily scale up and down our services replicas. This is a great feature when your services were created based on the assumption that they will be scaled by the platform by creating new copies of the containers running the service. So, what happens when the service is not ready to handle replication, or when there are no replicas available for a given service?

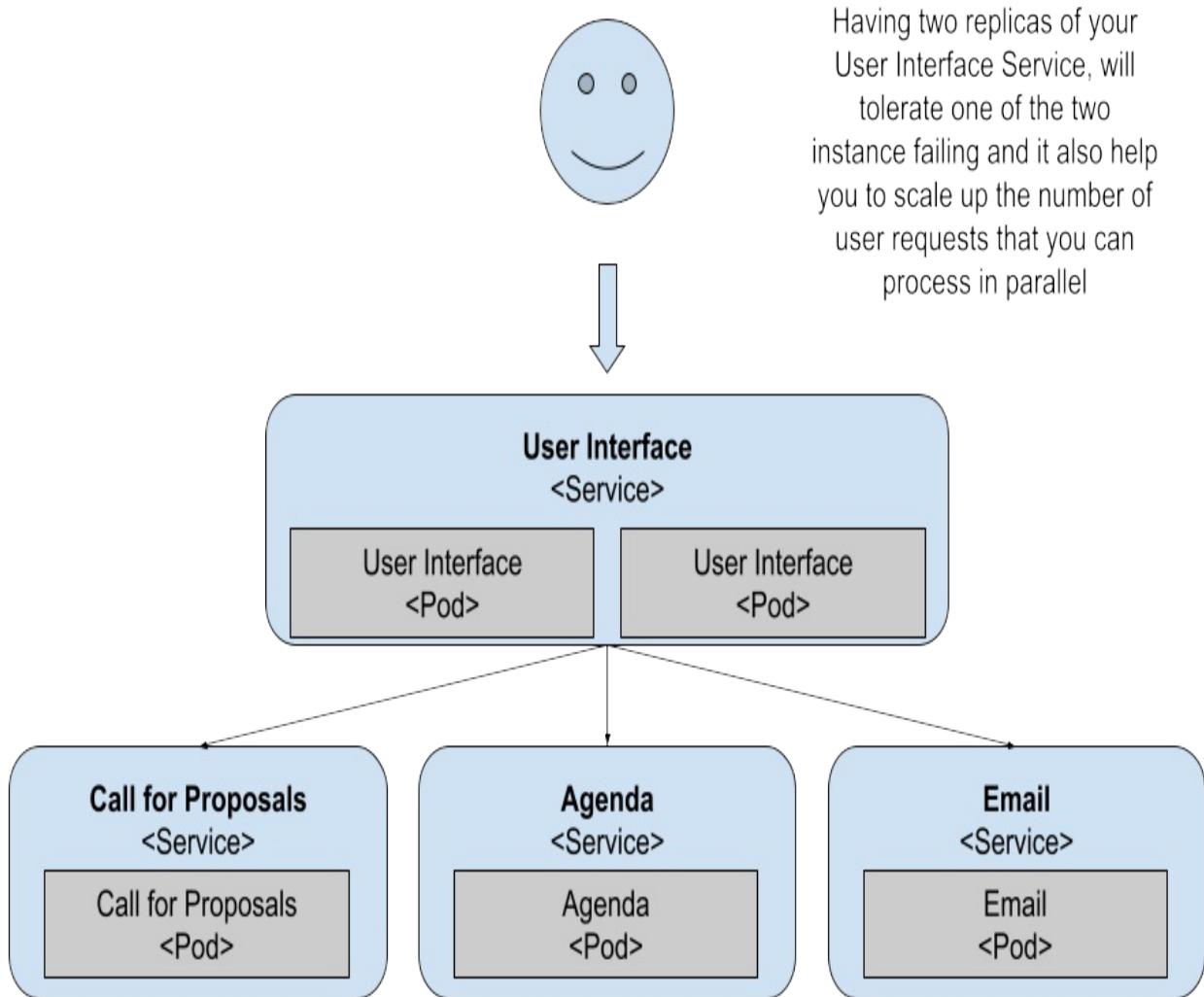
Let's scale up the Frontend service to have two replicas running all the time. To achieve this you can run the following command:

```
> kubectl scale --replicas=2 deployments/conference-fmtok8s-front
```

If one of the replicas stops running or breaks for any reason, Kubernetes will

try to start another one to make sure that 2 replicas are up all the time.

Figure 2.X Two replicas for Frontend service



You can quickly try this self-healing feature of Kubernetes by killing one of the two pods of the application Frontend. You can do this by running the following commands:

Listing 2.X Checking that the two replicas are up and running

```
> kubectl get pods
```

NAME	READY	STAT
conference-fmtok8s-agenda-service-57576cb65c-s127p	1/1	Runn

conference-fmtok8s-c4p-service-6c6f9449b5-j6ksv	1/1	Runn
conference-fmtok8s-email-service-6fdf958bdd-4pzww	1/1	Runn
conference-fmtok8s-frontend-5bf68cf65-pwk5d	1/1	Runn
conference-fmtok8s-frontend-5bf68cf65-zvlzq	1/1	Runn
conference-postgresql-0	1/1	Runn
conference-redis-master-0	1/1	Runn
conference-redis-replicas-0	1/1	Runn

Now, copy one of the two Pods Id and delete it:

```
> kubectl delete pod conference-fmtok8s-frontend-5bf68cf65-fc295
```

Then list the pods again:

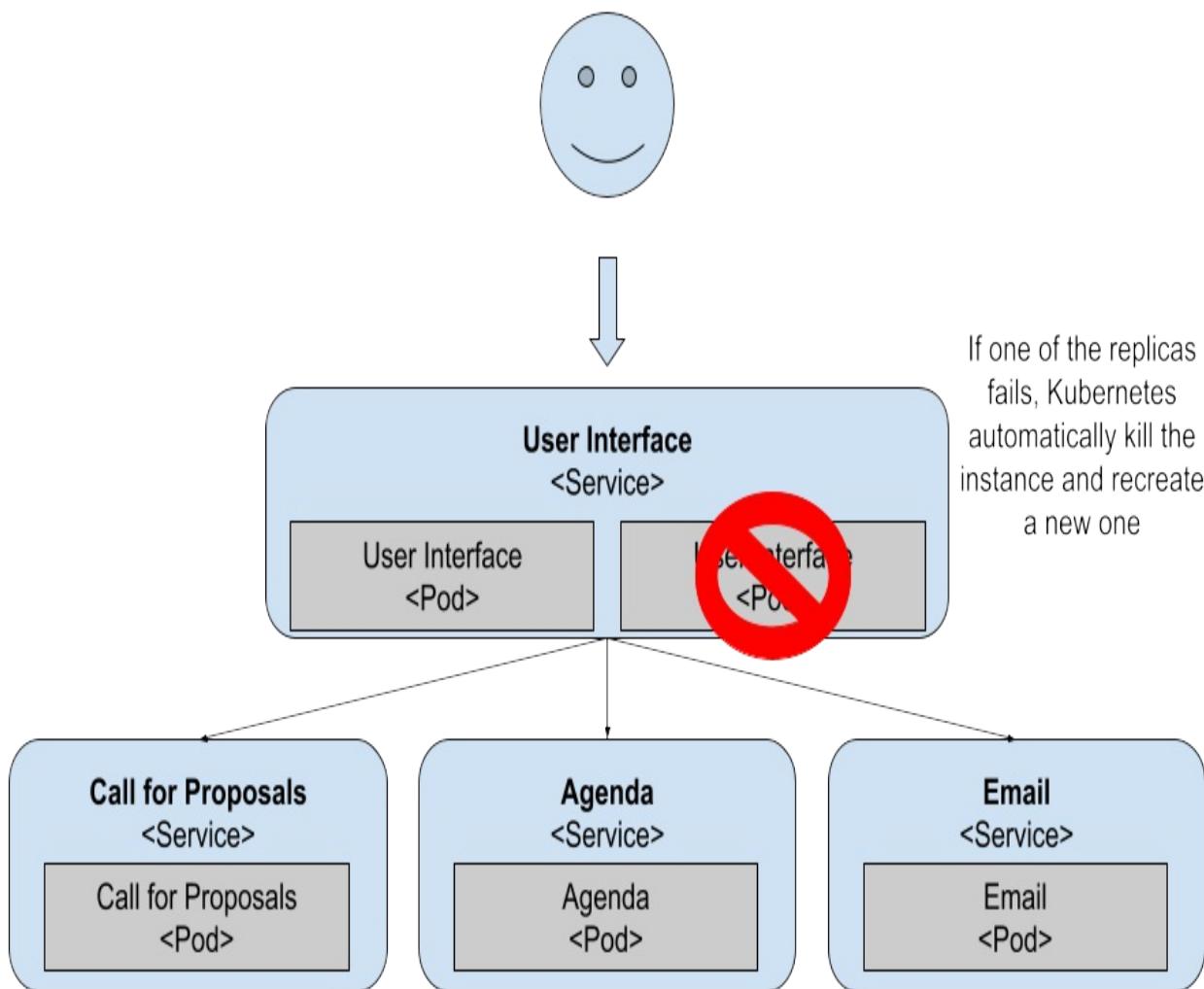
Listing 2.X A new replica is automatically created by Kubernetes as soon as one goes down

```
> kubectl get pods
```

NAME	READY	STAT
conference-fmtok8s-agenda-service-57576cb65c-s127p	1/1	Runn
conference-fmtok8s-c4p-service-6c6f9449b5-j6ksv	1/1	Runn
conference-fmtok8s-email-service-6fdf958bdd-4pzww	1/1	Runn
conference-fmtok8s-frontend-5bf68cf65-8j71t	0/1	Runn
conference-fmtok8s-frontend-5bf68cf65-fc295	1/1	Term
conference-fmtok8s-frontend-5bf68cf65-zvlzq	1/1	Runn
conference-postgresql-0	1/1	Runn
conference-redis-master-0	1/1	Runn
conference-redis-replicas-0	1/1	Runn

You can see how Kubernetes (the ReplicaSet more specifically) immediately creates a new pod when it detects that there is only one running. While this new pod is being created and started, you have a single replica answering your requests until the second one is up and running. This mechanism ensures that there are at least two replicas answering your users' requests.

Figure 2.X As soon as Kubernetes detects one pod misbehaving it will kill it and create a new one



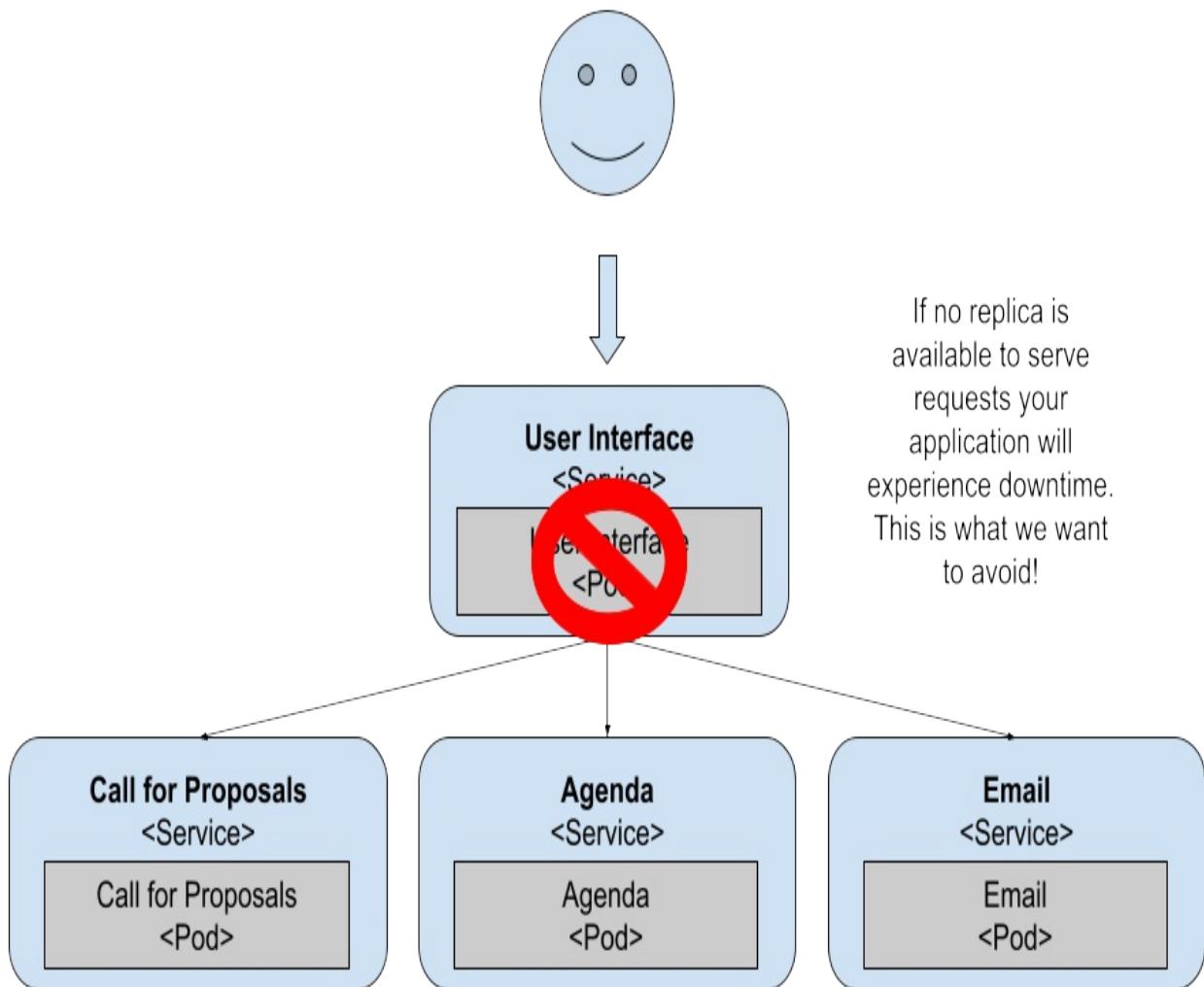
If you have a single replica, if you kill the running pod, you will have downtime in your application until the new container is created and ready to serve requests. You can revert back to a single replica with:

```
kubectl scale --replicas=1 deployments/conference-fmtok8s-fronten
```

Go ahead and try this out, delete only the replica available for the Frontend Pod:

```
> kubectl delete pod <POD_ID>
```

Figure 2.X With a single replica being restarted, there is no backup to answer user requests



Right after killing the pod, try to access the application by refreshing your browser (<http://localhost>). You should see “503 Service Temporarily Unavailable” in your browser, as the Ingress Controller (not shown in the previous figure for simplicity) cannot find a replica running behind the API Gateway service. If you wait for a bit, you will see the application come back up.

Figure 2.X With a single replica being restarted, there is no backup to answer user requests

localhost

503 Service Temporarily Unavailable

503 Service Temporarily Unavailable

nginx

Service Downtime

```
salaboy - kubectl delete pod app-fmtok8s-api-gateway-76478ffffd6-d6vh9 - 102x19
[salaboy] > k get pods
NAME                               READY   STATUS    RESTARTS   AGE
app-fmtok8s-agenda-rest-cfd77946f-zjgrq   1/1     Running   0          51m
app-fmtok8s-api-gateway-76478ffffd6-d6vh9   1/1     Running   0          51m
app-fmtok8s-c4p-rest-596b95594d-vb872      1/1     Running   0          51m
[salaboy] > k delete pod app-fmtok8s-api-gateway-76478ffffd6-d6vh9
pod "app-fmtok8s-api-gateway-76478ffffd6-d6vh9" deleted
```

Pod Deleted

This behavior is to be expected, as the Frontend Service is a user-facing service. If it goes down, users will not be able to access any functionality, hence having multiple replicas is recommended. From this perspective, we can assert that the FrontEnd service is the most important service of the entire application as our primary goal for our applications is to avoid downtime.

In summary, pay special attention to user-facing services exposed outside of your cluster. No matter if they are User Interfaces or just APIs, make sure that you have as many replicas as needed to deal with incoming requests. Having a single replica should be avoided for most use cases besides development.

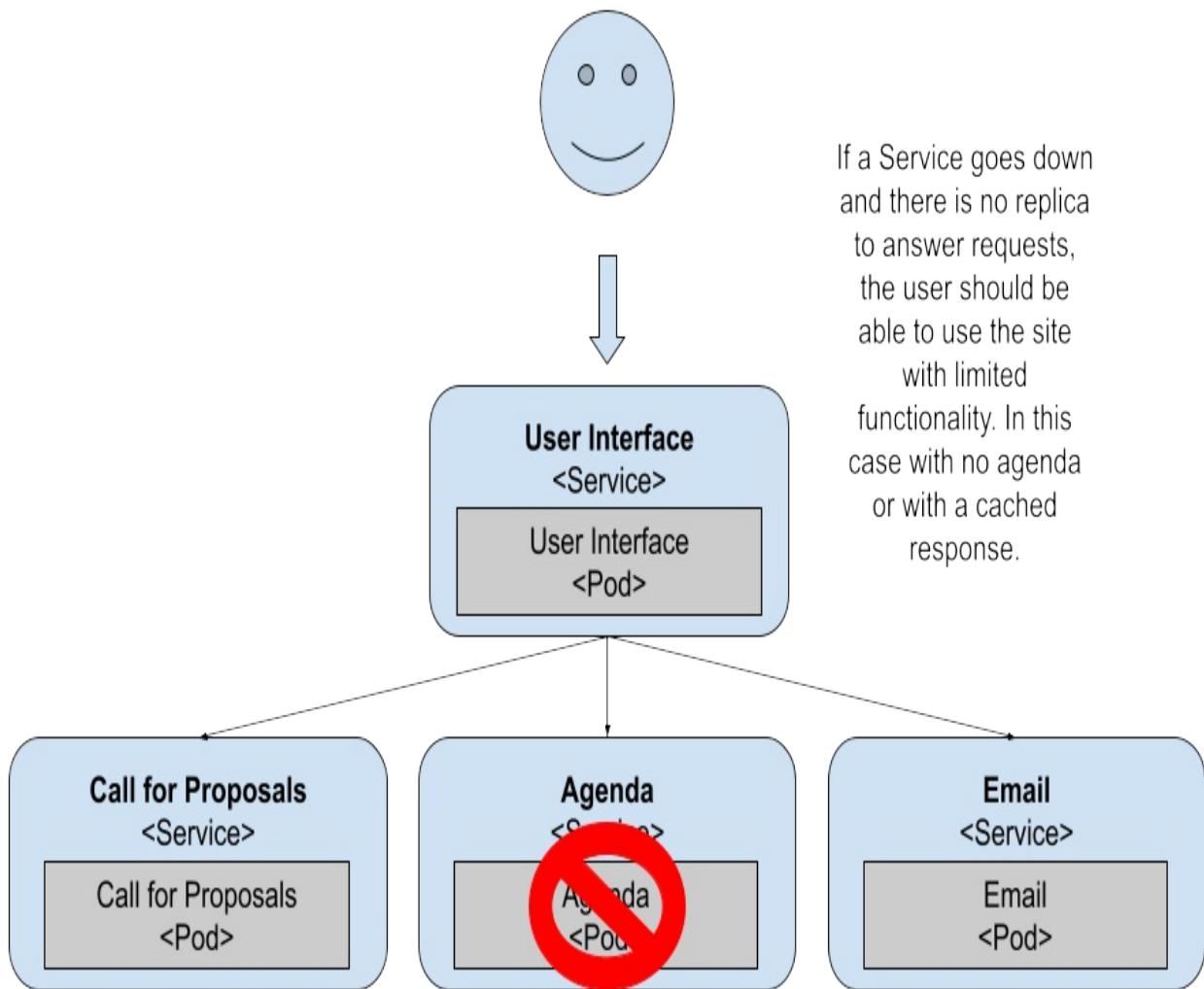
2.3.2 Service's resilience built-in

But now, what happens if the other services go down? For example, the Agenda Service, is just in charge of listing all the accepted proposals to the conference attendees.

This service is also critical, as the Agenda List is right there on the main page of the application. So, let's scale the service down:

```
> kubectl scale --replicas=0 deployments/conference-fmtok8s-agend
```

Figure 2.X No pods for the Agenda Service



If a Service goes down and there is no replica to answer requests, the user should be able to use the site with limited functionality. In this case with no agenda or with a cached response.

Right after running this command, the container will be killed and the service will not have any container answering its requests.

Try refreshing the application in your browser:

Figure 2.X If the Agenda service has no replica running, the Frontend is wise enough to show the user some cached entries

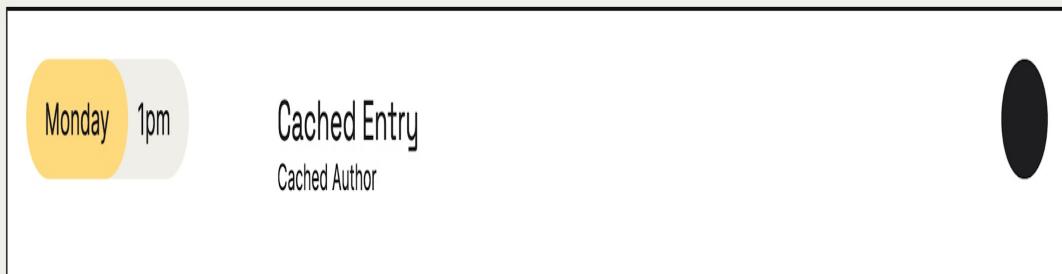


[Agenda](#) [Proposals](#) [About](#)

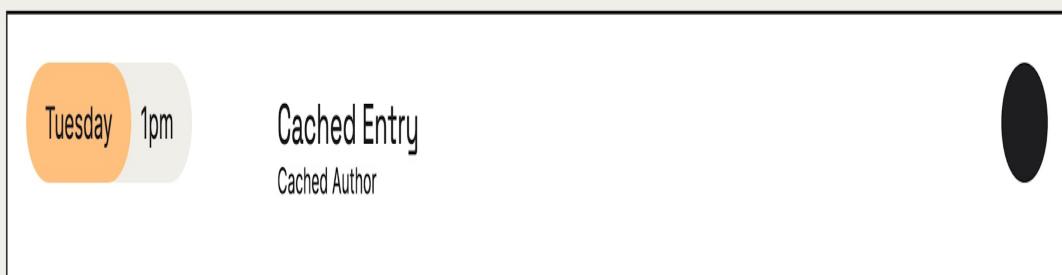
→

Conference Agenda

Monday's sessions



Tuesday's sessions



As you can see, the application is still running, but the Agenda Service is not available right now. You can prepare your application for such scenarios; in this case, the Frontend has a cached response to at least show something to the user. If for some reason the Agenda Service is down, at least the user will be able to access other services and other sections of the application. From the application perspective, it is important to not propagate the error back to the user. The user should be able to keep using other services of the application until the Agenda Service is restored.

You need to pay special attention when developing services that will run in Kubernetes as now your service is responsible for dealing with errors generated by downstream services. This is important to make sure that errors or services going down doesn't bring your entire application down. Having simple mechanisms such as cached responses will make your applications more resilient and it will also allow you to incrementally upgrade these services without worrying about bringing everything down. For our conference scenario, having a cron job that periodically caches the agenda entries might be enough. Remember, downtime is not allowed.

Let's now switch to talking about dealing with the state in our applications and how it is critical to understand how our application's services are handling the state from a scalability point of view. Since we will be talking about scalability, data consistency is the challenge that we will try to solve next.

2.3.3 Dealing with application state is not trivial

Let's scale up the agenda service again to have a single replica:

```
> kubectl scale --replicas=1 deployments/conference-fmtok8s-agend
```

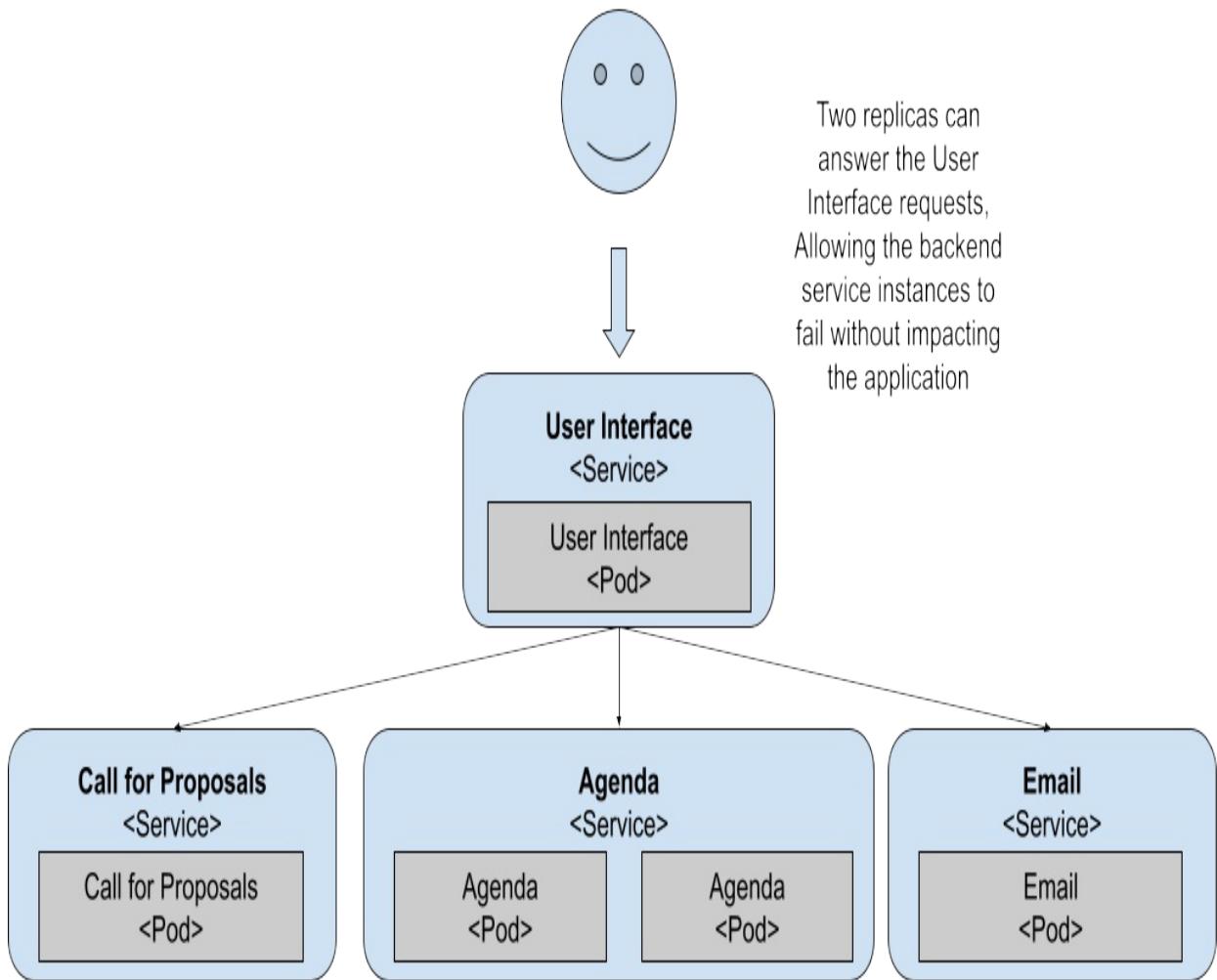
If you have created proposals before, you will notice that as soon as the Agenda service goes back up you will see the accepted proposals back again on the Agenda page. This is working only because both the Agenda Service and C4P Service store all the proposals and agenda items in external databases (PostgreSQL and Redis). In this context, external means outside of the pod memory.

What do you think will happen if we scale the Agenda Service up to two replicas?

```
> kubectl scale --replicas=2 deployments/conference-fmtok8s-agend
```

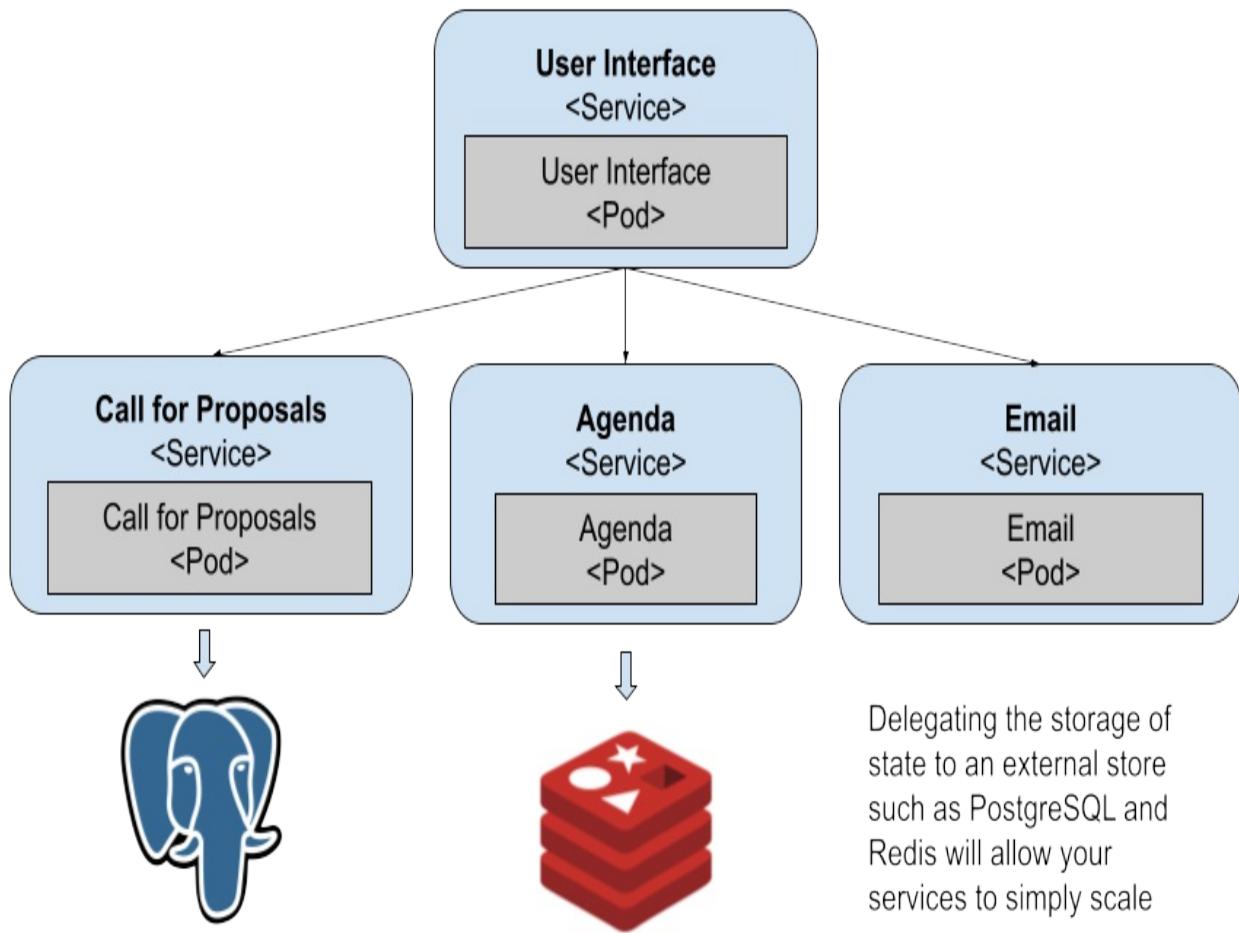
NAME	READY	STAT
conference-fmtok8s-agenda-service-57576cb65c-7pxf2	1/1	Runn
conference-fmtok8s-agenda-service-57576cb65c-cq75h	1/1	Runn
conference-fmtok8s-c4p-service-6c6f9449b5-j6ksv	1/1	Runn
conference-fmtok8s-email-service-6fdf958bdd-4pzww	1/1	Runn
conference-fmtok8s-frontend-5bf68cf65-zvlzq	1/1	Runn
conference-postgresql-0	1/1	Runn
conference-redis-master-0	1/1	Runn
conference-redis-replicas-0	1/1	Runn

Figure 2.X Two replicas can now deal with more traffic



With two replicas dealing with your user requests, now the Frontend will have two instances to query. Kubernetes will do the load balancing between the two replicas, but your application will have no control over which replica is the request hitting. Because we are using a database to back up the data outside of the context of the pod we can scale the replicas to many pods that can deal with the application demand.

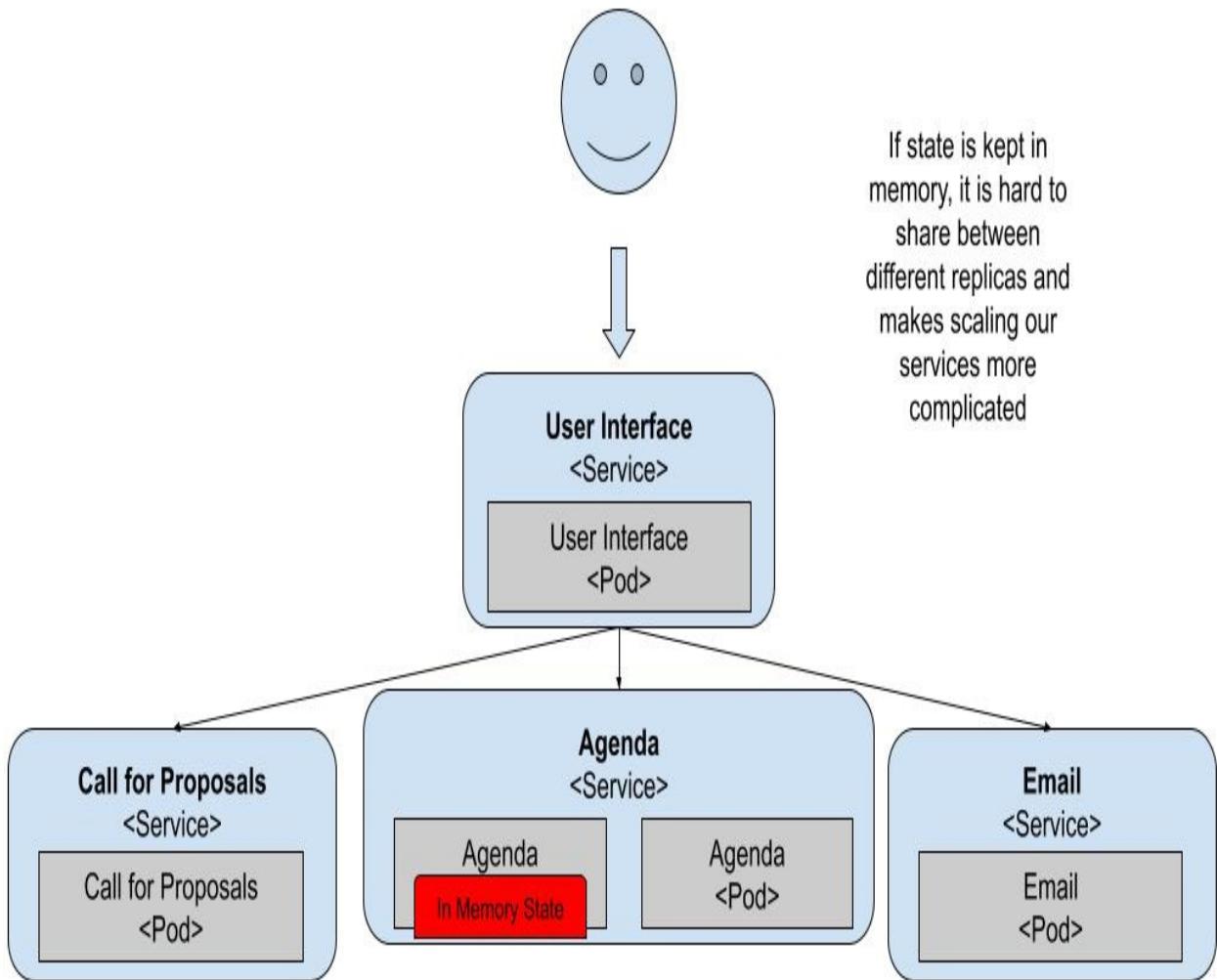
Figure 2.X Both data sensitive services use persistent stores



One of the limitations of this approach is the number of database connections that your database support in its default configuration. If you keep scaling up the replicas, always consider reviewing the database connection pool settings to make sure that your database can handle all the connections being created by all the replicas.

But for the sake of learning, let's imagine that we don't have a database, and our Agenda Service keeps all the agenda items in memory. How would the application behave if we started scaling up the Agenda Service pods?

Figure 2.X What would happen if the Agenda Service keeps state in-memory?



By scaling these services up, we have found an issue with the design of one of the application services. The Agenda Service is keeping state in-memory and that will affect the scaling capabilities from Kubernetes. For this kind of scenario, when Kubernetes balance the requests across different replicas the frontend service will receive different data depending on which replica processed the request.

When running existing applications in Kubernetes you will need to have a deep understanding of how much data they are keeping in-memory as this will affect how you can scale them up. For web applications that keep HTTP sessions and require sticky sessions (subsequent requests going to the same replica), you will need to set up HTTP session replication to get this working with multiple replicas. This might require more components being configured

at the infrastructure level, such as a cache.

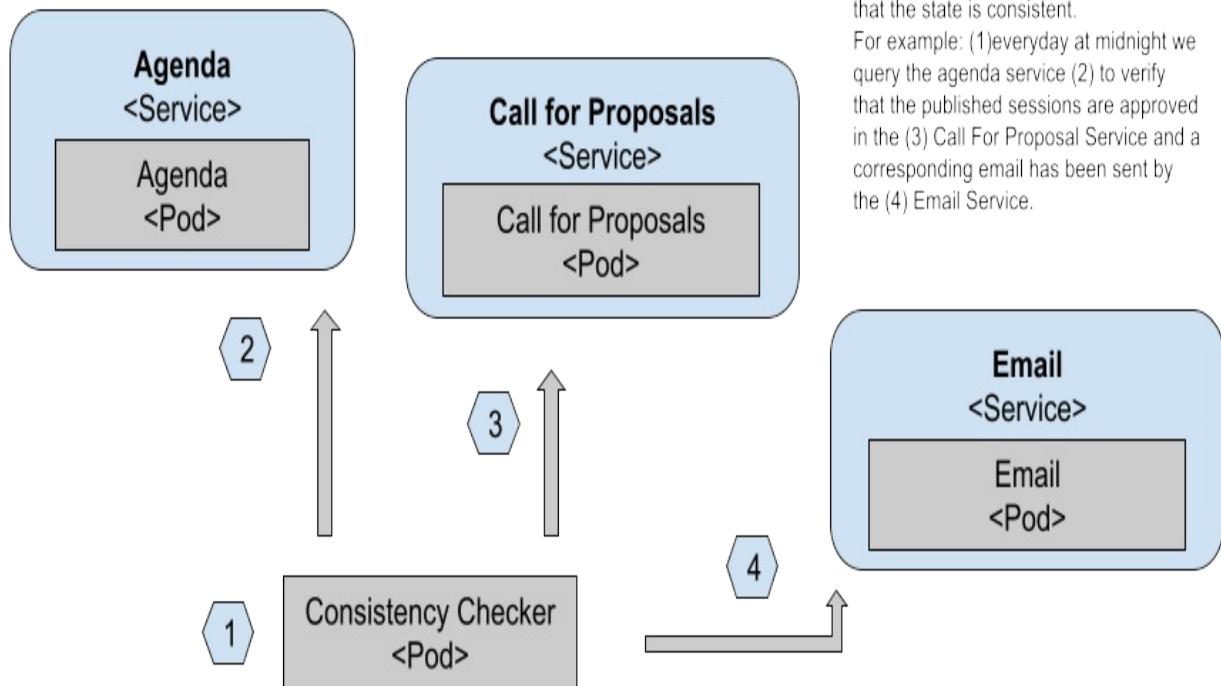
Understanding your service requirements will help you to plan and automate your infrastructural requirements, such as Databases, caches, message brokers, etc. The larger and more complex the application gets, the more dependencies on these infrastructural components it will have.

As we have seen before, we have installed Redis and PostgreSQL as part of the application Helm chart. This is usually not a good idea as databases and tools like message brokers will need special care by the operation team that can choose not to run these services inside Kubernetes. We will expand on this topic on Chapter 4 where we go deeper into how to deal with infrastructure when working with Kubernetes and Cloud Providers.

2.3.4 Dealing with inconsistent data

Having stored data in a relational data store like PostgreSQL or a NoSQL approach like Redis doesn't solve the problem of having inconsistent data across different stores. As these stores should be hidden away by the service API, you will need to have mechanisms to check that the data that the services are handling is consistent. In distributed systems it is quite common to talk about "eventual consistency", meaning that eventually, the system will be consistent. Having eventual consistency is definitely better than not having consistency at all. For this example, one thing that we can build is a simple check mechanism that once in a while (imagine once a day) checks for the accepted talks in the Agenda Service to see if they have been approved in the Call for Proposal Service. If there is an entry that hasn't been approved by the Call for Proposal Service (C4P), then we can raise some alerts or send an email to the conference organizers.

Figure 2.X Consistency checks can run as CronJobs



In figure 2.X we can see how a CronJob (1) will be executed every X period, depending on how important it is for us to fix consistency issues. Then it will query the Agenda Service public APIs (2) to check which accepted proposals are being listed and compare that with the Call for Proposals Service approved list (3). Finally, if any inconsistency is found, an email can be sent using the Email Service public APIs (4).

Think of the simple use case this application was designed for, what other checks would you need? One that immediately comes to my mind is about verifying that emails were sent correctly for Rejection and Approved proposals. For this use case, emails are really important, and we need to make sure that those emails are sent to our accepted and rejected speakers.

2.3.5 Understanding how the application is working

Distributed systems are complex beasts and fully understanding how they work from day one can help you to save time down the line when things go wrong. This has pushed the monitoring, tracing and telemetry communities really hard to come up with solutions that help us to understand how things

Every regular periods we can execute checks against the services to make sure that the state is consistent.

For example: (1)everyday at midnight we query the agenda service (2) to verify that the published sessions are approved in the (3) Call For Proposal Service and a corresponding email has been sent by the (4) Email Service.

are working at any given time.

The <https://opentelemetry.io/> OpenTelemetry community has evolved alongside Kubernetes and it can now provide most of the tools that you will need to monitor how your services are working. As stated on their website: “You can use it to instrument, generate, collect, and export telemetry data (metrics, logs, and traces) for analysis in order to understand your software's performance and behavior.” It is important to notice that OpenTelemetry focuses on both the behavior and performance of your software as they both will impact your users and user experience.

From the behavior point of view, you want to make sure that the application is doing what it is supposed to do and by that, you will need to understand which services are calling which other services or infrastructure to perform tasks.

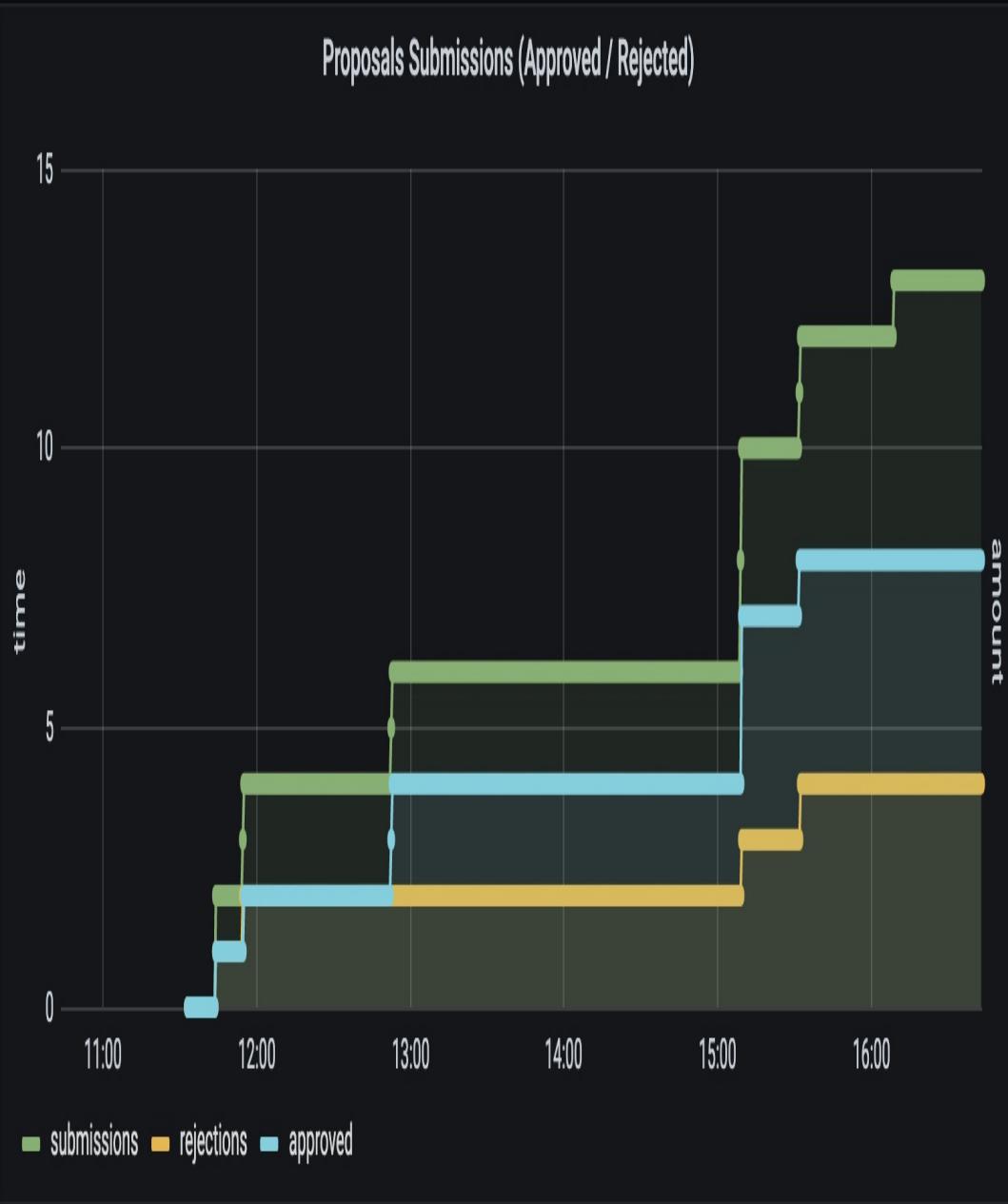
Figure 2.X Monitoring telemetry data with Prometheus & Grafana

← → C

localhost:3000/d/RWDSd-UGk/conference-submissions?orgId=1



Conference Submissions ★ ⚙

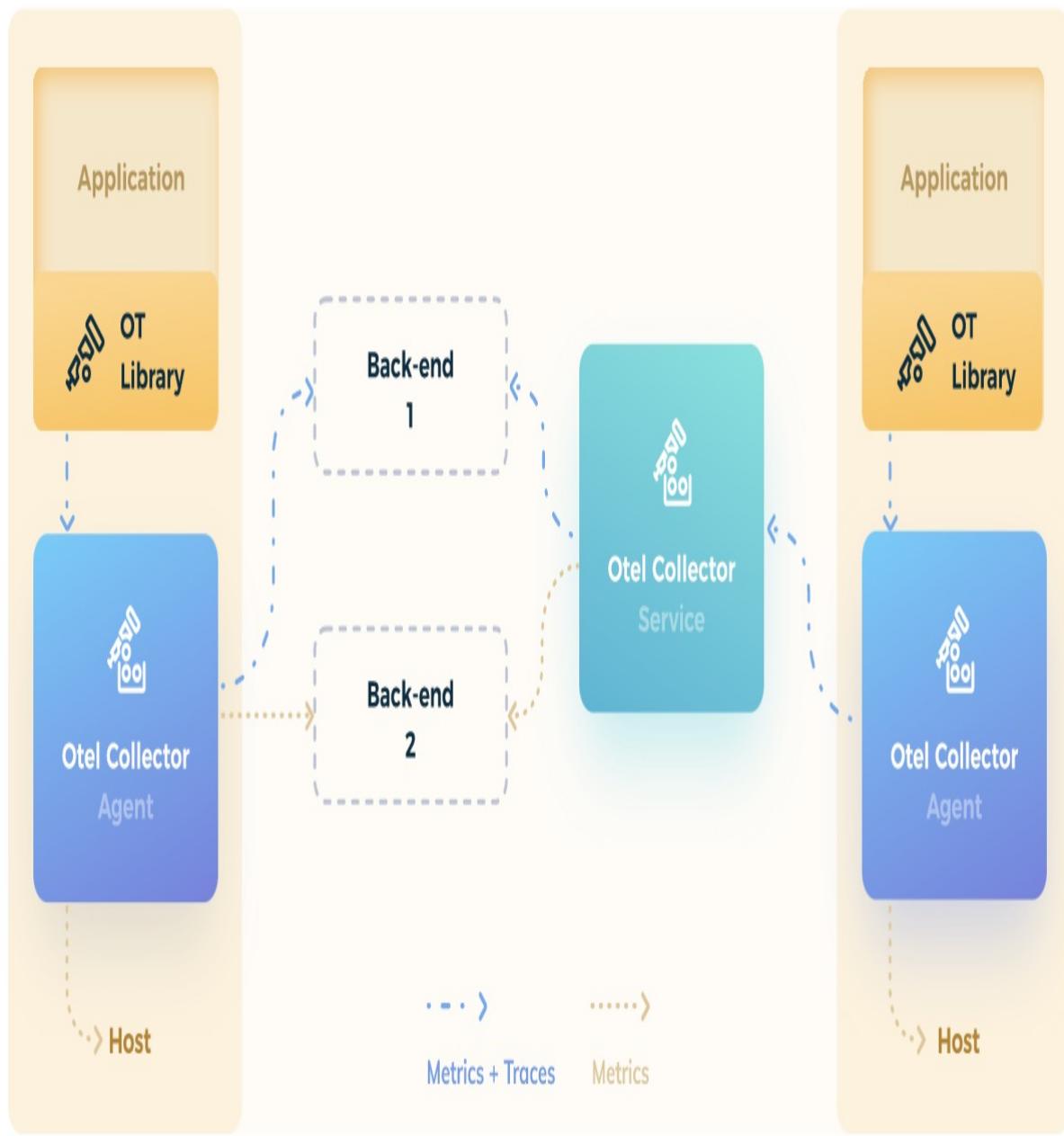


Using Prometheus and Grafana allows us to not only see the service telemetry, but also build domain specific dashboards to highlight certain application level metrics, for example the amount of Approved vs Rejected proposals over time as shown in figure 2.x.

From the performance point of view, you need to make sure that services are respecting their Service Level Agreements (SLAs) which basically means that they are not taking too long to answer requests. If one of your services is misbehaving and taking more than usual, you want to be aware of that.

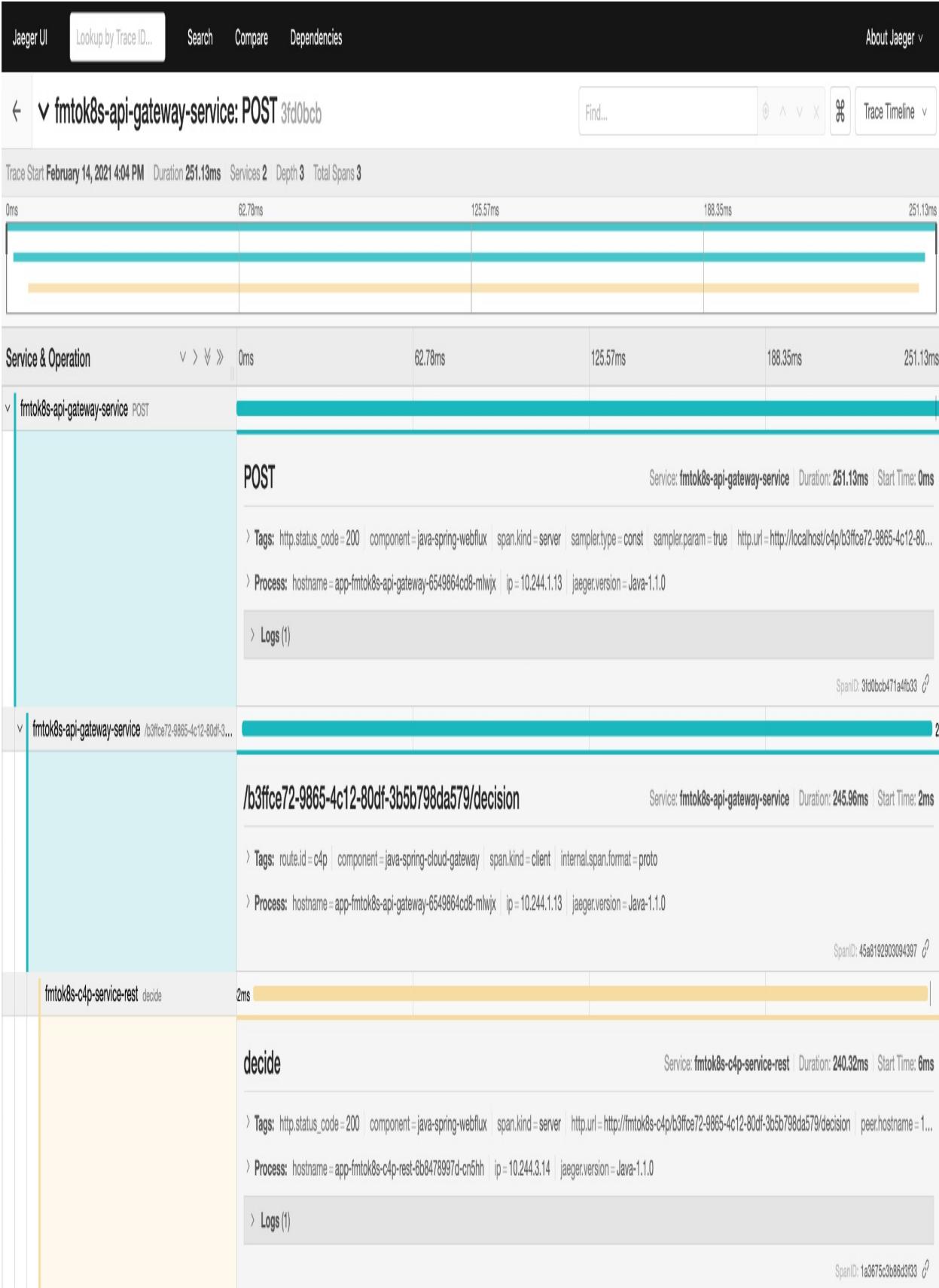
For tracing, you will need to modify your services if you are interested in understanding the internal operations and their performance. OpenTelemetry provides drop-in instrumentation libraries in most languages to externalize service metrics and traces.

Figure 2.X OpenTelemetry architecture and library



Once the instrumentation libraries are included in our services we can check the traces with tools like Jaeger:

Figure 2.X Tracing diagram from Jaeger



You can see in the previous figure, the amount of time used by each service while processing a single user-request.

The recommendation here is: if you are creating a walking skeleton, make sure that it has OpenTelemetry built-in. If you push monitoring to later stages of the project it will be too late, things will go wrong and finding out who is responsible will take you too much time.

2.3.6 Application security and identity management

If you have ever built a Web application you know that providing identity management (user accounts and user identity) plus Authentication and Authorization is quite an endeavor. A simple way to break any application (cloud-native or not) is to perform actions that you are not supposed to do, such as deleting all the proposed presentations unless you are a conference organizer.

In distributed systems, this becomes challenging as well, as authorization and user identity needs to be propagated across different services. In distributed architectures, it is quite common to have a component that generates requests on behalf of a user instead of exposing all the services for the user to interact directly. In our example, the Frontend service is this component. Most of the time you can use this external-facing component as the barrier between external and internal services. For this reason, it is quite common to configure the Frontend Service to connect with an Authorization & Authentication provider commonly using the OAuth2 protocol.

On the Identity Management front, you have seen that the application itself doesn't handle users or their data and that is a good thing for regulations such as GDPR. We might want to allow users to use their social media accounts to login into our applications without the need for them to create a separate accounts. This is usually known as social login.

There are some popular solutions that brings both OAuth2 and Identity Management together such as Keycloak (<https://www.keycloak.org/>) and Zitadel(<https://zitadel.comopensource>). These Open Source projects provide a one-stop-shop for Single Sign-On solution and advanced Identity

Management. In the case of Zitadel, it also provide a managed service that you can use if you don't want to install and maintain a component for SSO and Identity management inside your infrastructure.

Same as with Tracing and Monitoring, if you are planning to have users (and you will probably do, sooner or later) including Single Sign-On and identity management into the walking skeleton will push you to think the specifics of “who will be able to do what”, refining your use case even more.

2.3.7 Other challenges

In the previous sections, we have covered a few common challenges that you will face while building Cloud-Native applications, but these are not all. Can you think of other ways of breaking this first version of the application?

Notice that tackling the challenges discussed in this chapter will help, but there are other challenges related to how we deliver a continuously evolving application composed of a growing number of services.

2.4 Linking back to Platform Engineering

In previous sections, we have covered a wide range of topics. We started by creating a local Kubernetes cluster using KinD, we reviewed the options for packaging and distributing Kubernetes applications, to then install our walking skeleton in our just created cluster. We used the application by interacting with the application that was running in our KinD cluster and finally we jumped into analyzing how the application can break by looking at common Cloud-Native challenges that we will face when building this kind of distributed applications.

But you might be wondering how all these topics tied back to the title of this book, continuous delivery aspects and platform engineering in general?

In this section, we will be making more explicit connections to the topics introduced in chapter 1.

First of all, the intention behind creating a Kubernetes Cluster and running an

application on top of it was to make sure we cover Kubernetes built-in mechanisms for resilience and scaling up our application services. Kubernetes provides the building blocks to run our applications with zero downtime, even when we are constantly updating them. This allows us, Kubernetes users, to release new versions of our components more frequently, as we are not supposed to stop the entire application to update one of its parts. In chapter 7 (Promoting Cloud-Native experimentation), we will see how Kubernetes built-in mechanisms inside Deployments go a step further providing us mechanisms that we can use to implement different release strategies. The main takeaway here is that, if you are not using Kubernetes capabilities to speed up your release cycles, enabling your teams to push features from source to end users you might need to review what's stopping you. Quite often, this can be due to old practices from before Kubernetes that are getting in the middle, lack of automation, or not having clearly defined contracts between services that block upstream services to be released in an independent way. We will touch on this topic several times in future chapters as this is a fundamental principle when trying to improve your Continuous Delivery practice and something that the Platform Engineering team need to prioritize.

In this chapter, we have also seen how we can install a Cloud-Native application using a package manager which encapsulates the configuration files required to deploy our application into a Kubernetes Cluster. These configuration files (Kubernetes resources expressed as YAML files) describe our application topology and contain links to the containers used by each application's service. These YAML files also contain configuration for each of the services, such as the resource required to run each service for Kubernetes to know where to place each container based on the Kubernetes Cluster topology and available resources. By packaging and versioning these configuration files we can easily create new instances of the application in different environments, something that we will cover in Chapters 3.

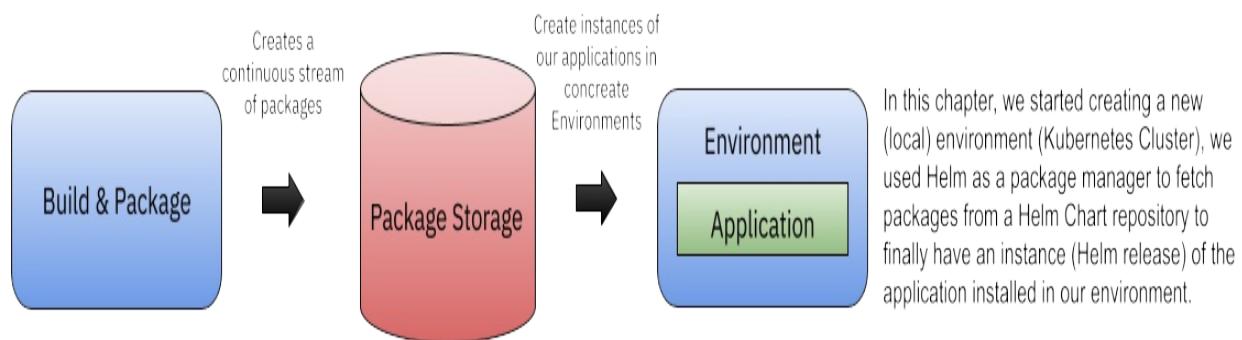
I highly recommend the Manning book *Grokkering Continuous Delivery*, by Christie Wilson, which covers how using configuration as code can help you to deliver more software in a more reliable way.

Tools like Helm rely on Kubernetes to apply the differences between

different versions of our configuration files, this means that if we create a new version of our application package, let's say 0.1.1 and we already have 0.1.0 running in our Kubernetes cluster (as we did in Section 2.1.3) only the changes in configuration will be applied, all the services that were not changed will not be affected. For those services that were changed, Kubernetes mechanisms will kick in to roll the updated configurations. Tools like Helm provide simple ways to do rollbacks to previous versions of the application when things go wrong. All these mechanisms and tools had been designed with the main purpose of enabling teams to be more confident in releasing new versions.

Because I wanted to make sure that you have an application to play around and because we needed to cover Kubernetes built-in mechanisms, I've made a conscious decision to start with an already packaged application that can be easily deployed into any Kubernetes Cluster, no matter if it is running locally or in a Cloud-Provider. We can clearly identify two different phases, one we haven't covered is how to produce these packages that can be deployed to any Kubernetes cluster and the second which we started playing with is when we run this application in a concrete cluster (we can consider this cluster an environment).

Figure 2.X Applications' lifecycle from building and packaging to running inside an environment

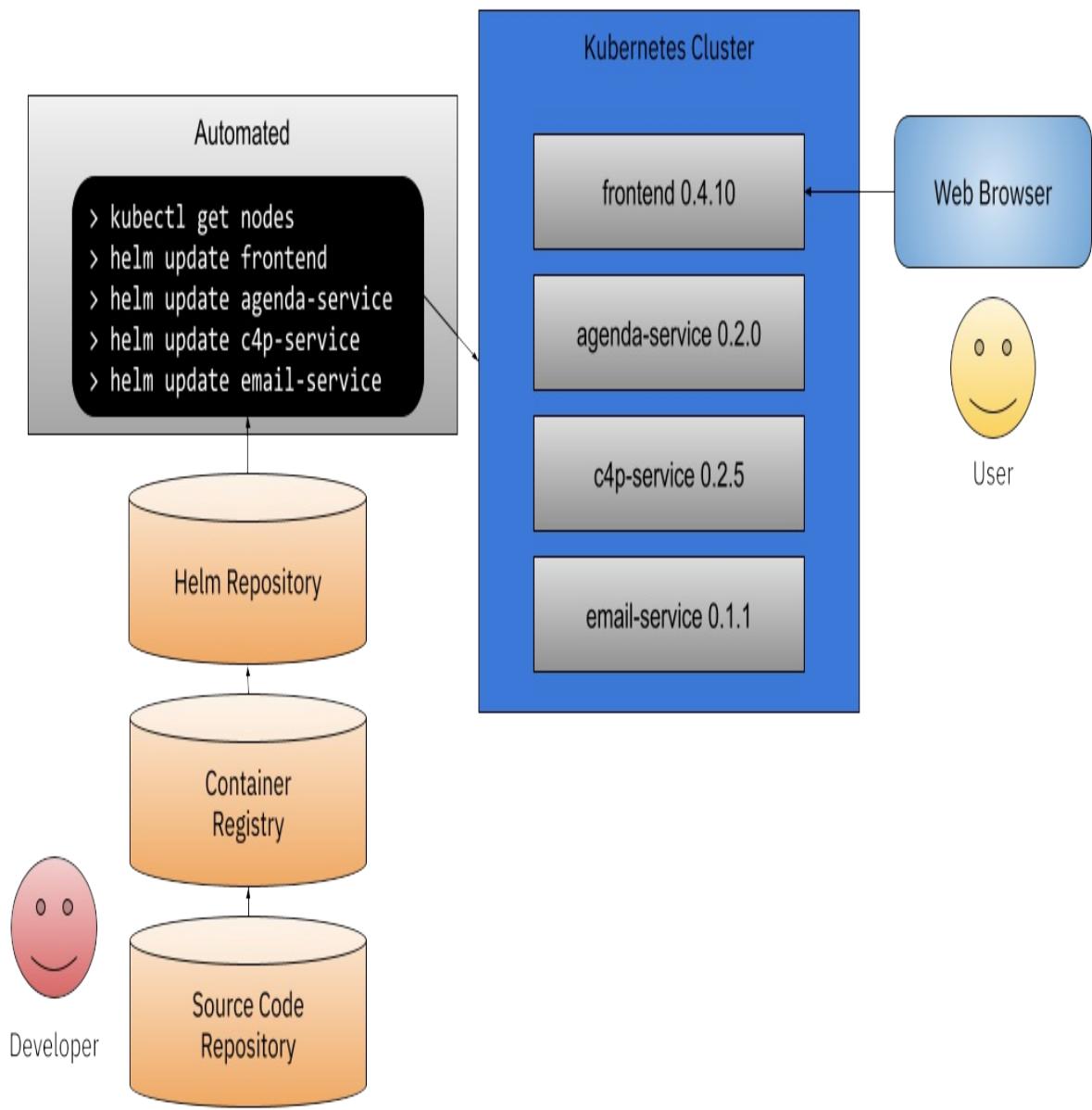


But no matter which tools you are using, one thing is key: if you don't automate how you use these tools your velocity of delivering software will suffer.

It is important to understand that the steps that we have executed for our local environment will work for any Kubernetes Cluster no matter the cluster size and location. While each Cloud Provider will have its own security and identity mechanisms, the Kubernetes APIs and resources that we created when we installed our application Helm Chart to the cluster are going to be exactly the same. If you now use Helm capabilities to fine-tune your application (for example resource consumptions and network configurations) for the target environment you can easily automate these deployments to any Kubernetes cluster.

Before moving on, it is worth also recognizing that a developer configuring application instances might not be the best use of their time. A developer having access to the Production Environment that Users/Customers are accessing might also not be optimal, hence we want to aim to make sure that developers are focused on building new features and improving our application. Figure 2.x shows how we should be automating all the steps involved in building, publishing, and deploying the artifacts that developers are creating, making sure that they can focus on keep adding features to the application instead of manually dealing with packaging, distributing, and deploying new versions when they are ready. This is the primary focus of this chapter.

Figure 2.X Developers can focus on building features, we need to automate the entire process



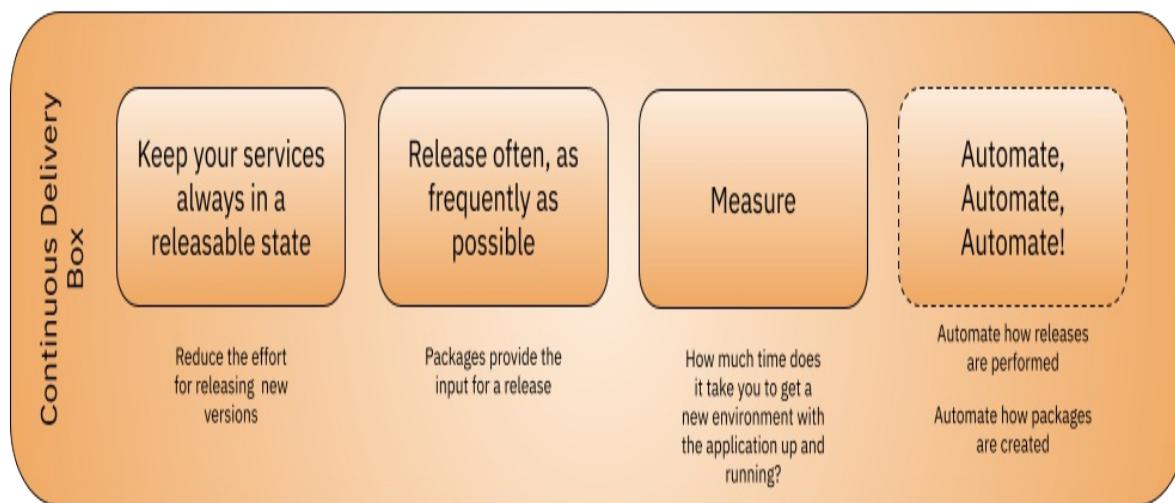
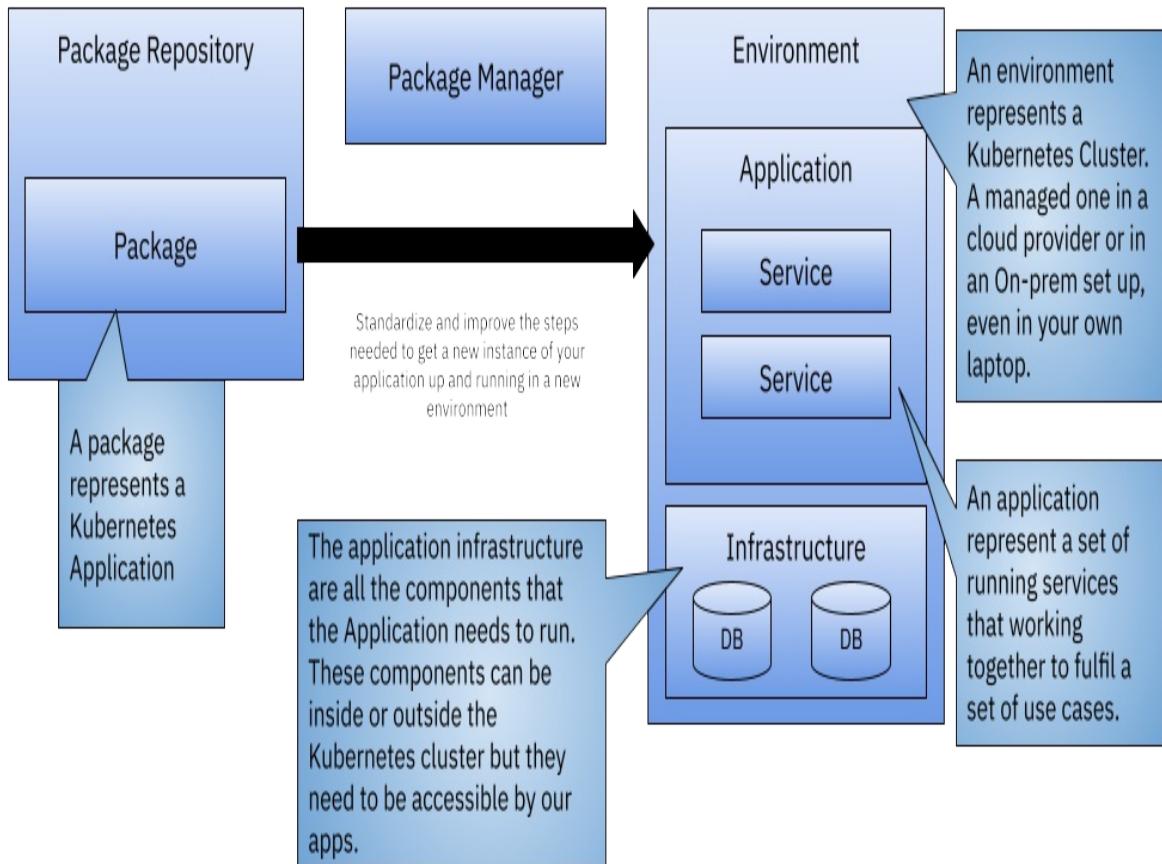
Understanding the tools that we can use to automate the path from source code changes to running software in a Kubernetes cluster is fundamental to enabling developers to focus on what they do best “code new features”. Another big difference that we will be tackling is that Cloud Native applications are not static, as you can see in the previous diagram we will not be installing a static application definition, we want to be able to release and deploy new versions of the services as they become available.

Manually installing applications is error-prone, manually changing

configurations in our Kubernetes clusters can make us end up in situations where we don't know how to replicate the current state of our application in a different environment, hence in the next chapter, we will jump to talk about automation using what is commonly known as Pipelines.

To conclude this chapter, here you can find a mental map of the concepts that we have discussed so far and how they relate to our continuous delivery goals.

Figure 2.X Linking back to Continuous Delivery practices and goals



In the next chapter, we will cover a more dynamic aspect of our distributed application with Pipelines to deliver new versions of our services, and following that, chapter 4 will explore how we can deal with infrastructural components that are vital for our application to work in a declarative way.

2.5 Summary

- Choosing between local and remote Kubernetes clusters requires serious considerations
 - You can use Kubernetes KinD to bootstrap a local Kubernetes Cluster to develop your application. The main drawback is that your cluster is limited by your local resources (CPU and Memory) and the fact that it is not a real cluster of machines.
 - You can have an account in a Cloud Provider and do all development against a remote cluster. The main drawback of this approach is that most developers are not used to work remotely all the time and the fact that someone needs to pay for the remote resources.
- Package managers, like Helm, helps you to package, distribute and install your Kubernetes applications, in this chapter you have fetched and installed an application into a local Kubernetes cluster with a single command line
- Understanding which Kubernetes resources are created by your application gives you an idea about how the application will behave when things go wrong and what extra considerations are needed in real-life scenarios.
- Even with very simple applications, you will be facing challenges that you will need to tackle one at a time. Knowing these challenges ahead of time helps you to plan and architect your services with the right mindset.
- Having a walking skeleton helps you to try different scenarios and technologies in a controlled environment. In this chapter you have experimented with:
 - Scaling up and down your services to see first-hand how the application behaves when things go wrong
 - Keeping state is hard and we will need dedicated components to do this efficiently
 - Having at least 2 replicas for our services maximizes the chances to avoid downtime. Making sure that the user-facing components are always up and running guarantees that even when things go wrong the user will be able to interact with parts of the application
 - Having fallbacks and built-in mechanisms to deal with problems when they arise makes your application a whole more resilient.

3 Service pipelines: Building cloud-native applications

This chapter covers

- The components needed to deliver Cloud Native applications to Kubernetes
- The advantages of creating and standardizing Service Pipelines for your services
- How projects like Tekton, Dagger, and tools like GitHub Actions can help us to build our Cloud-Native applications

In the previous chapter, you installed and interacted with a simple distributed Conference application composed of 4 services. This chapter covers what it takes to deliver each of these components continuously by using the concept of Pipelines as delivery mechanisms. This chapter describes and shows in practice how each of these services can be built, packaged, released, and published so they can run in your organization's environments.

This chapter introduces the concept of *Service Pipelines*. The Service Pipeline takes all the steps needed to build your software from source code until the artifacts are ready to run.

This chapter is divided into 2 main sections:

- What does it take to deliver a Cloud-Native application?
- Service Pipelines
 - What is a Service Pipeline?
 - Service pipelines in action using
- Tekton a Kubernetes Native Pipeline Engine
- Dagger, code your pipelines, and then run everywhere
- Should I use Tekton, Dagger, or GitHub actions?

3.1 What does it take to deliver a Cloud-Native Application continuously?

When working with Kubernetes, teams are now responsible for more moving pieces and tasks involving containers and how to run them in Kubernetes. These extra tasks don't come for free. Teams need to learn how to automate and optimize the steps required to keep each service running. Tasks that were the responsibility of the operations teams are now becoming more and more the responsibility of the teams in charge of developing each of the individual services. New tools and new approaches give developers the power to develop, run and maintain the services they produce. The tools that we will look at in this chapter are designed to automate all the tasks involved to go from source code to a service that is up and running inside a Kubernetes cluster.

This chapter describes the mechanisms needed to deliver software components (our application services) to multiple environments where these services will run. But before jumping into the tools, let's take a quick look at the challenges that we are facing.

Building and delivering cloud-native applications present significant challenges that teams must tackle:

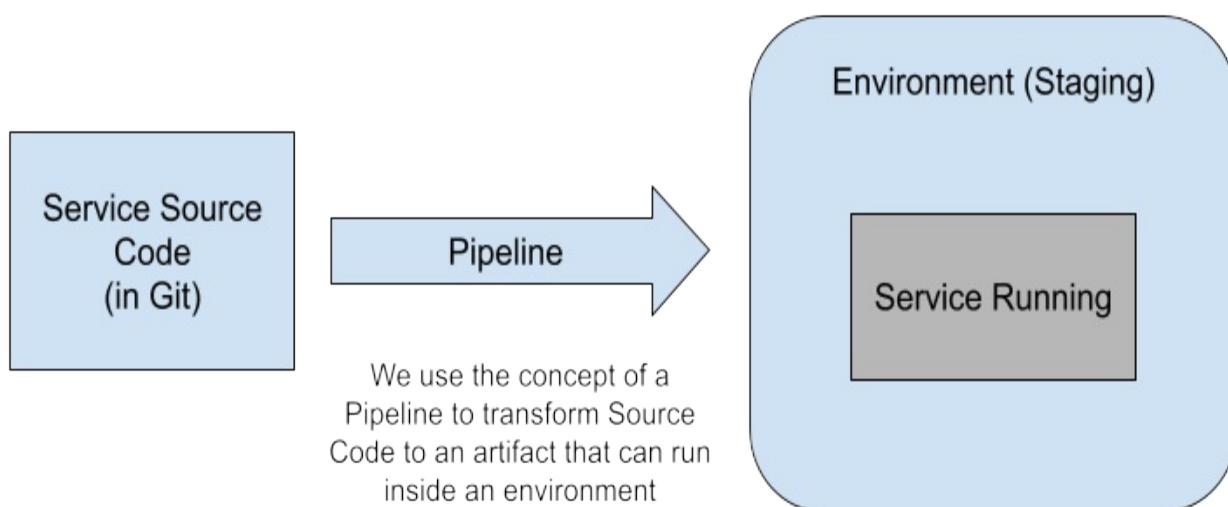
- **Dealing with different team interactions** when building different pieces of the application. This requires coordination between teams and making sure that services are designed in a way that the team responsible for a service is not blocking other teams' progress or their ability to keep improving their services.
- We need to **support upgrading a service without breaking or stopping all the other running services**. If we want to achieve continuous delivery, services should be able to be upgraded independently without the fear of bringing down the entire application.
- **Storing and publishing several artifacts per service that can be accessed/downloaded from different environments, which might be in different regions**. If we are working in a cloud environment, all servers are remote and all produced artifacts need to be accessible for each of these servers to fetch. If you are working on an On-premise

setup, all the repositories for storing these artifacts must be provisioned, configured, and maintained in-house.

- **Managing and provisioning different environments for various purposes such as Development, Testing, Q&A, and Production.** If you want to speed up your development and testing initiatives, developers and teams should be able to provision these environments on-demand. Having environments configured as close as possible to the real Production environment will save you a lot of time in catching errors before they hit your live users.

As we saw in the previous chapter, the main paradigm shift when working with Cloud-Native applications is that there is no single code base for our application. Teams can work independently on their services, but this requires new approaches to compensate for the complexities of working with a distributed system. If teams will worry and waste time every time a new Service needs to be added to the system, we are doing things wrong. End-to-end automation is necessary for teams to feel comfortable adding or refactoring services. This automation is usually performed by what is commonly known as **Pipelines**. As shown in figure 3.X, these pipelines describe what needs to be done to build and run our services, and usually, they can be executed without human intervention.

Figure 3.X from source to a running service using a Pipeline



You can even have pipelines to automate the creation of a new service or to

even add new users to your identity management solution.

But what are these Pipelines doing exactly? Do we need to create our own Pipelines from scratch? How do we implement these pipelines in our Projects? Do we need one or more pipelines to achieve this?

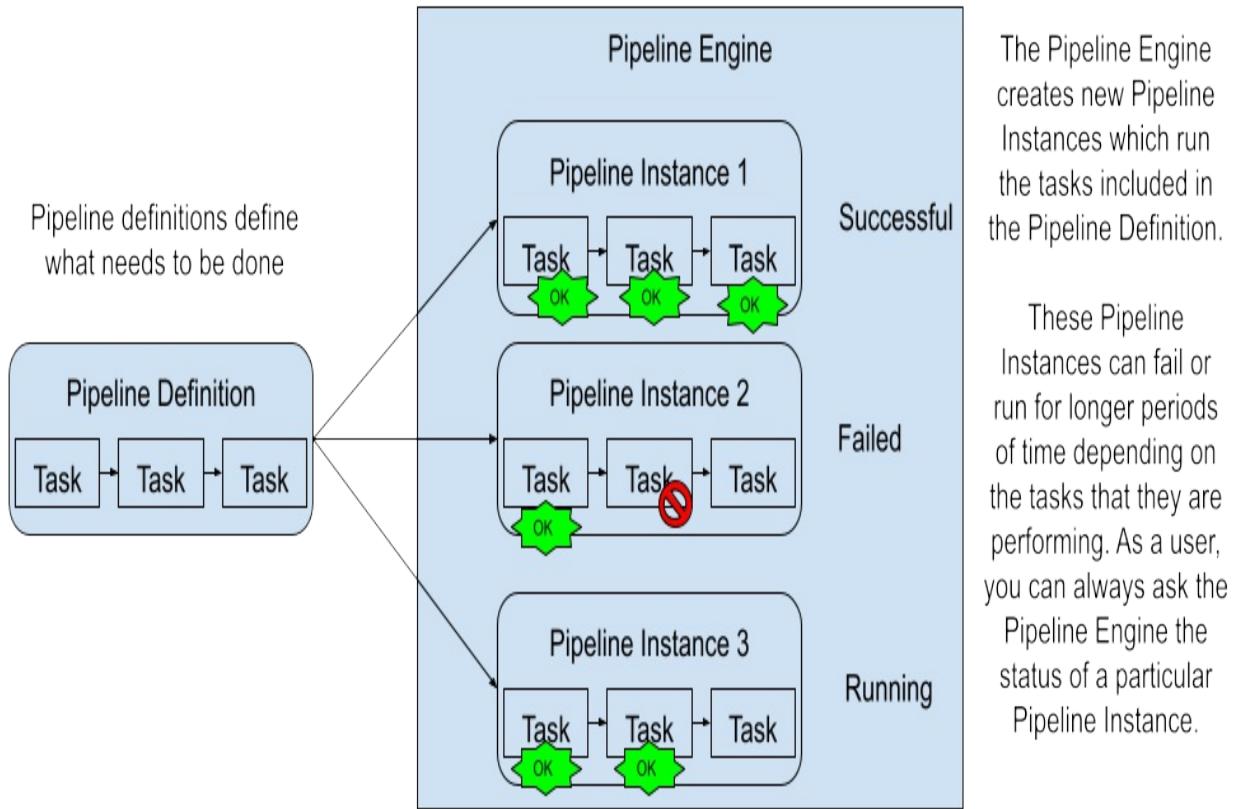
The following section (3.2) is focused on using Pipelines to build solutions that can be copied, shared, and executed multiple times to produce the same results. Pipelines can be created for different purposes and it is pretty common to define them as a set of steps (one after the other in sequence) that produce a set of expected outputs. Based on these outputs, these pipelines can be classified into different groups.

Most pipeline tools out there allow you to define pipelines as a collection of tasks (also known as steps or jobs) that will run a specific job or script to perform a concrete action. These steps can be anything, from running tests, copying code from one place to another, deploying software, provisioning virtual machines, creating users, etc.

Pipeline definitions can be executed by a component known as the Pipeline Engine, which is in charge of picking up the pipeline definition to create a new pipeline instance that runs each task. The tasks will be executed one after the other in sequence, and each task execution might generate data that can be shared with the following task. If there is an error in any of the steps involved with the pipeline, the pipeline stops, and the pipeline state will be marked as in error (failed). If there are no errors, the pipeline execution (also known as pipeline instance) can be marked as successful. Depending on the pipeline definition and if the execution was successful, we should verify that the expected outputs were generated or produced.

In figure 3.x, we can see the Pipeline engine picking up our Pipeline definition and creating different instances that can be parameterized differently for different outputs. For example, Pipeline Instance 1 finished correctly, while Pipeline Instance 2 is failing to finish executing all the tasks included in the definition. Pipeline Instance 3, in this case, is still running.

Figure 3.X A Pipeline definition can be instantiated by a Pipeline Engine multiple times



As expected, with these Pipeline definitions we can create loads of different automation solutions, and it is common to find tools that build more specific solutions on top of a pipeline engine or even hide the complexity of dealing with a pipeline engine to simplify the user experience. In the following sections, we will be looking for examples of different tools, some more low-level and very flexible, and some more high-level, more opinionated, and designed to solve a very concrete scenario.

But how do these concepts and tools apply to delivering Cloud-Native applications? For Cloud-Native applications, we have very concrete expectations about how to build, package, release and publish our software components (Services) and where these should be deployed. In the context of delivering cloud-native applications, we can define two main kinds of pipelines:

- **Service Pipelines:** take care of the building, unit testing, packaging, and distributing (usually to an artifact repository) of our software artifacts
- **Environment Pipelines:** take care of deploying and updating all the

services in a given environment such as staging, testing, production, etc. usually consuming what needs to be deployed from a source of truth.

The remaining of chapter 3 focuses on Service Pipelines, while chapter 4 focuses on tools that helps us to define Environment Pipelines using a more declarative approach known as GitOps.

By separating the build process (Service Pipeline) and the deployment process (Environment Pipeline) we give more control to the teams responsible for promoting new versions in front of our customers.

Service and Environment Pipelines are executed on top of different resources and with different expectations. The following section (3.2) goes into more detail about the steps that we commonly define in our service pipelines. Chapter 4 covers what is expected from environment pipelines.

3.2 Service Pipelines

A Service pipeline is in charge of defining and executing all the steps required to build, package, and distribute a service artifact so it can be deployed into an environment. A Service Pipeline is not responsible for deploying the newly created service artifact, but it can be responsible for notifying interested parties that there is a new version available for the service.

If you standardize how your services must be built, packaged, and released, you can share the same pipeline definition for different services. You should try to avoid pushing each of your teams to define a completely different pipeline for each service, as they will probably be reinventing something that has already been defined, tested, and improved by other teams.

There is a considerable amount of tasks that need to be performed, and a set of conventions that, when followed, can reduce the amount of time required to perform these tasks.

The name “Service Pipeline” makes reference to the fact that each of our application’s services will have a pipeline that describes the tasks required for

that particular service. If the services are similar and they are using a similar technology stack, it makes sense for the pipelines to look quite similar. One of the main objectives of these Service Pipelines is to contain enough detail so they can be run without any human intervention, automating all the tasks included in the pipeline end to end.

Service pipelines can be used as a mechanism to improve the communication between the development team that is creating a service and the operations team that is running that service in production. Development teams expect that these pipelines will run and notify them if there is any problem with the code that they are trying to build. If there are no errors, they will expect one or more artifacts to be produced as part of the pipeline execution. Operation teams can add all the checks to these pipelines to ensure that the produced artifacts are production ready. These checks can include policy and conformance checks, signing, security scanning, and other requirements that validate that the produced artifacts are up to the standards expected to run in the production environment.

Note: it is tempting to think about creating a single pipeline for the entire application (collection of services), as we did with Monolith applications. However, that defeats the purpose of independently updating each service at its own pace. You should avoid situations where you have a single pipeline defined for a set of services, as it will block your ability to release services independently.

3.2.1 Conventions that will save you time

Service Pipelines can be more opinionated on how they are structured and their reach, by following some of these strong opinions and conventions, you can avoid pushing your teams to define every little detail and discover these conventions by trial and error. The following approaches have been proven to work:

- **Trunk-Based Development:** the idea here is to make sure that what you have in the main branch of your source code repository is always ready to be released. You don't merge changes that break the build and release process of this branch. You only merge if the changes you are merging

are ready to be released. This approach also includes using feature branches, which allow developers to work on features without breaking the main branch. When the features are done and tested, developers can send Pull Requests (Change requests) for other developers to review and merge. This also means that when you merge something to the main branch, you can automatically create a new release of your service (and all the related artifacts). This creates a continuous stream of releases generated after each new feature is merged into the main branch. Because each release is consistent and has been tested, you can then deploy this new release to an environment that contains all the other services of your application. This approach enables the team behind the Service to move forward and keep releasing without worrying about other services.

- **Source Code and Configuration Management:** there are different approaches to dealing with software and the configuration that is needed to run the software that we are producing. When we talk about services and distributed applications, there are two different schools of thought:
 - **One Service/One Repository/One Pipeline:** you keep your service source code and all the configurations that need to be built, packaged, released, and deployed into the same repository. This allows the team behind the service to push changes at any pace they want without worrying about other services' source code. It is a common practice to have the source code in the same repository where you have the Dockerfile describing how the docker image should be created and the Kubernetes manifest required to deploy the service into a Kubernetes cluster. These configurations should include the pipeline definition that will be used to build and package your service.
 - **Mono Repository:** alternatively, use a Mono repository approach where a single repository is used, and different pipelines are configured for different directories inside the repository. While this approach can work, you need to ensure that your teams are not blocking each other by waiting for each others' pull requests to merge.
- **Consumer-Driven Contract Testing:** Your service uses contracts to run tests against other services. Unit testing an individual service shouldn't require having other services up and running. By creating

Consumer-driven contracts, each service can test its functionality against other services' APIs. If any of the downstream services is released, a new contract is shared with all the upstream services so they can run their tests against the new version.

I recommend you to find more bibliography about these topics as these conventions are well documented and compared. Most of the tools mentioned in this book allow you to implement these practices for efficient delivery.

If we take these practices and conventions into account, we can define the responsibility of a Service Pipeline as follows:

“A Service Pipeline Transform source code to one or a set of artifacts that can be deployed in an environment”.

3.2.2 Service Pipeline structure

With this definition in mind, let's take a look at what tasks are included in Service Pipelines for Cloud-Native applications that will run on Kubernetes:

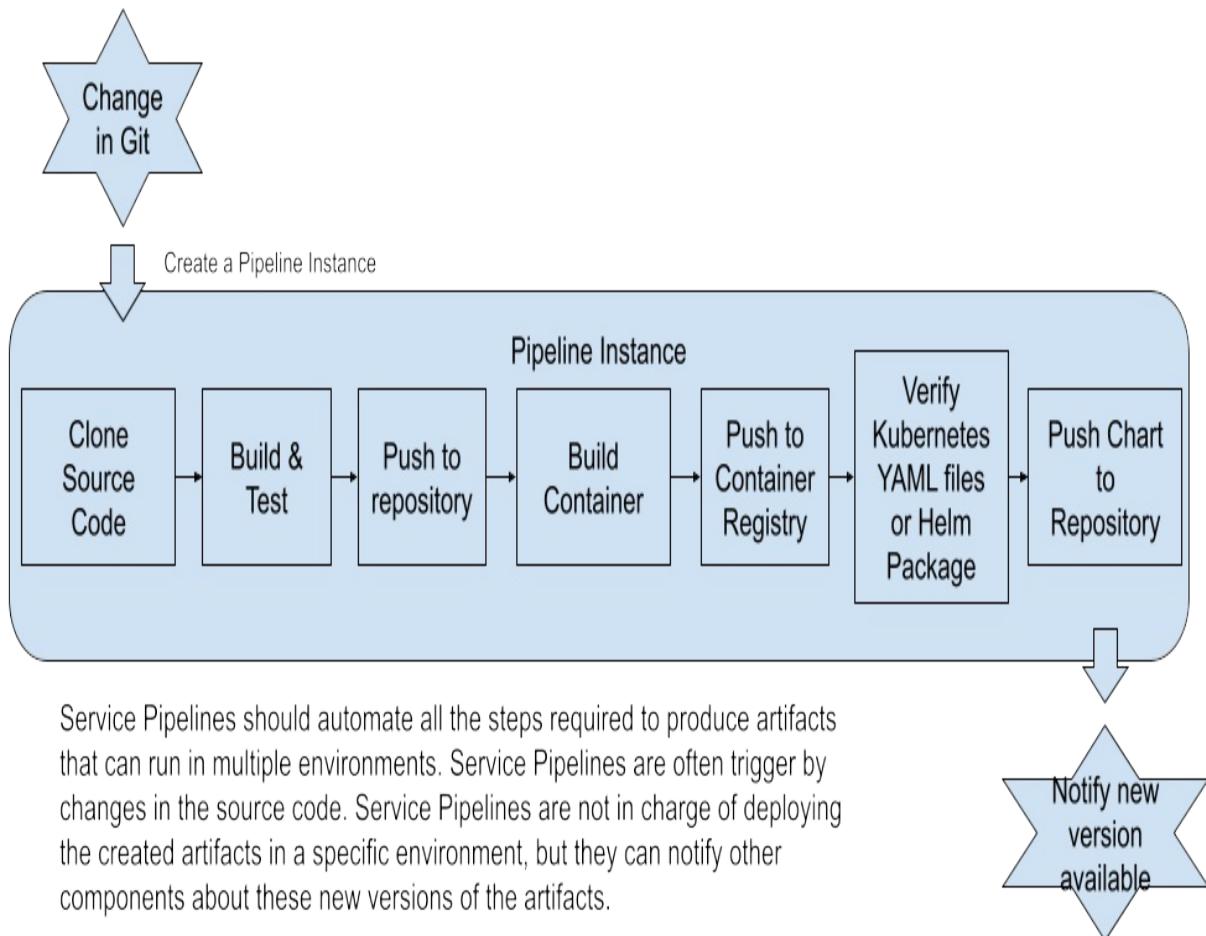
- **Register to receive notifications about changes in the source code repository main branch:** (source version control system, nowadays a Git repository): if the source code changes, we need to create a new release. We create a new release by triggering the Service Pipeline.
- **Clone the source code from the repository:** to build the service, we need to clone the source code into a machine that has the tools to build/compile the source code into a binary format that can be executed.
- **Create a new tag for the new version to be released:** based on trunk-based development, every time that a change happens, a new release can be created. This will help us to understand what is being deployed and what changes were included in each new release.
- **Build & Test the source code:**
 - As part of the build process, most projects will execute unit tests and break the build if there are any failures
 - Depending on the technology stack we are using, we will need to have a set of tools available for this step to happen, for example, compilers, dependencies, linters (static source code analyzers), etc.

- **Publish the binary artifacts into an artifact repository:** we need to make sure that these binaries are available for other systems to consume, including the next steps in the pipeline. This step involves copying the binary artifact to a different location over the network. This artifact will share the same version of the tag that was created in the repository, providing us with traceability from the binary to the source code that was used to produce it.
- **Building a container:** if we are building Cloud-Native services, we must build a container image. The most common way of doing this today is using Docker or other container alternatives. This step requires the source code repository to have, for example, a Dockerfile defining how this container image needs to be built and the mechanism to build ('builder') the container image. Some tools like CNCF Buildpacks (<https://buildpacks.io>) save us from having a Dockerfile and can automate the container-building process. Notice that nowadays, it is pretty essential to have the right tools for the job, as multiple container images might need to be generated for different platforms. For a Service version, we might end up having more than one container image, for example, one for `amd64` and one for `arm64`. All the examples in this book are built for these two platforms.
- **Publish the container into a container registry:** in the same way that we published the binary artifacts that were generated when building our service source code, we need to publish our container image into a centralized location where others can access it. This container image will have the same version as the tag created in the repository and the binary published. This helps us to clearly see which source code will run when you run the container image.
- **Lint, verify, and optionally package YAML files for Kubernetes Deployments (Helm can be used here):** if you are running these containers inside Kubernetes, you need to manage, store, and version Kubernetes manifest that defines how the containers are going to be deployed into a Kubernetes cluster. If you use a package manager such as Helm, you can version the package with the same version used for the binaries and the container image. My rule of thumb for packaging YAML files goes as follows: *"If you have enough people trying to install your services (Open Source project or very large globally distributed organization) you might want to package and version your*

YAML files. If you only have a few teams and few environments to handle, you can probably distribute the YAML files without using a packaging tool.”

- **(Optional) Publish these Kubernetes manifests to a centralized location:** if you are using Helm, it makes sense to push these Helm packages (called charts) to a centralized location. This will allow other tools to fetch these charts so they can be deployed in any number of Kubernetes clusters.
- **Notify interested parties about the new version of the service:** if we are trying to automate from source to a service running, the Service Pipeline should be able to send a notification to all the interested services which might be waiting for new versions to be deployed. These notifications can be in the form of Pull Requests to other repositories, events to an event bus, maybe an email to the teams interested in these releases, etc. A pull based approach can also work, where an agent is constantly monitoring the artifact repository (or container registry) to see if there are new versions available for a given artifact.

Figure 3.X Tasks expected for a service pipeline



The outcome of this pipeline is a set of artifacts that can be deployed to an environment to have the service up and running. The service needs to be built and packaged in a way that does not depend on any specific environment. The service can depend on other services to be present in the environment to work, for example, infrastructural components such as databases and message brokers or just other downstream services.

No matter the tool that you choose to use to implement these pipelines, you should be looking at the following characteristics:

- Pipelines run automatically based on changes (if you are following trunk-based development, one pipeline instance is created for every change in the repository's main branch).
- Pipelines executions will notify about the success or failure state with clear messages. This includes having easy ways to find, for example, the why and where the pipeline failed or how much time takes to execute

each step.

- Each pipeline execution has a unique `id` that we can use to access the log and the parameters that were used to run the pipeline, so we can reproduce the setup that was used to troubleshoot issues. Using this unique `id` we can also access the logs created by all the steps in the pipeline. By looking at the pipeline execution, we should also be able to find all the produced artifacts and where those were published.
- Pipelines can also be triggered manually and configured with different parameters for special situations. For example, to test a work-in-progress feature branch.

Let's now deep dive into the concrete details of how a Service Pipeline will look in real life.

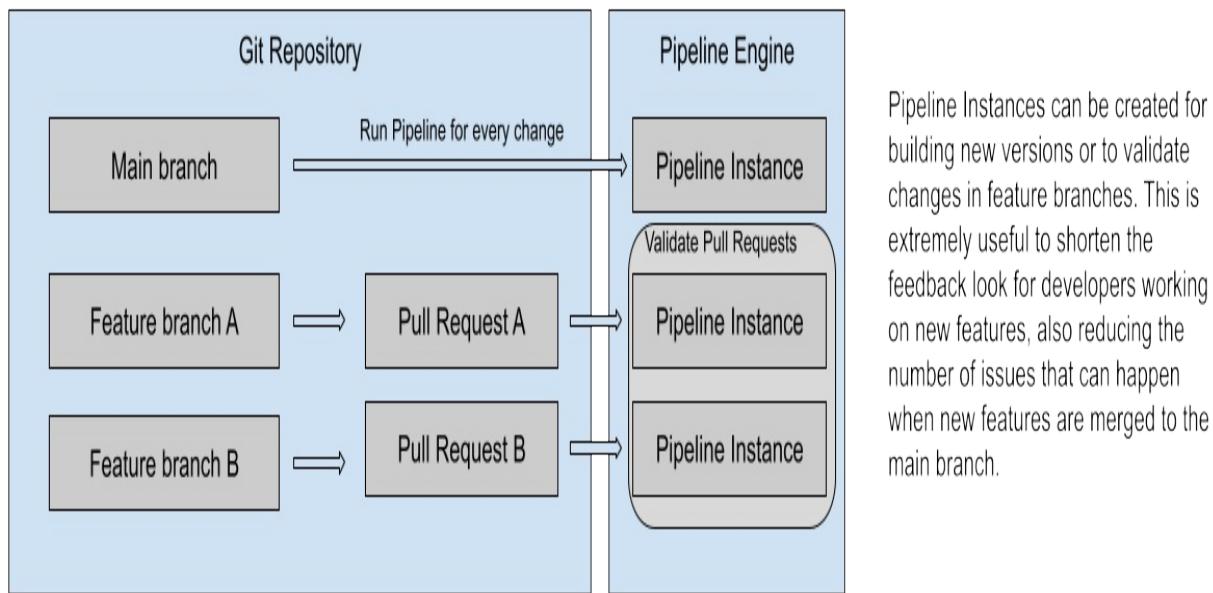
3.2.3 Service Pipeline in real life

In real life, there will be a Service pipeline that runs every time you merge changes to the main branch of your repository. This is how it should work if you follow a trunk-based development approach:

- When you merge changes to your main branch, this Service Pipeline runs and creates a set of artifacts using the latest code base. If the Service Pipeline succeeds, our artifacts will be in a releasable state. We want to make sure that our main branch is always in a releasable state, which means that the Service Pipeline which runs on top of the main branch must always succeed. If, for some reason, this pipeline is failing, the team behind the service needs to switch the focus to fixing the problem as soon as possible. In other words, teams shouldn't merge code into your main branch that breaks its Service Pipeline. To do that, we must also run a pipeline in our feature branches.
- For each of your feature branches, a very similar pipeline should run to verify that the changes in the branch can be built, tested, and released against the main branch. In modern environments, the concept of GitHub Pull Requests is used to run these pipelines to make sure that before merging any “Pull Request” a pipeline validates the changes.
- It is common that after merging a set of features to the main branch, and because we know that the main branch is releasable at all times the team

in charge of the service decides to tag a new release. In Git, this means creating a new tag (a pointer to a specific commit) based on the main branch. The tag name is commonly used to represent the version of the artifact that the pipeline will create.

Figure 3.X Service pipelines for main branch and feature branches



This Service Pipeline, shown in figure 3.X represents the most common steps that you will need to execute every time you merge something to the main branch. Still, there are also some variations of this pipeline that you might need to run under different circumstances. Different events can kick off a pipeline execution, we can have slightly different pipelines for different purposes, such as:

- **Validate a change in a feature branch:** this pipeline can execute the same steps as the pipeline in the main branch, but the artifacts generated should include the branch name, maybe as a version or as part of the artifact name. Running a pipeline after every change might be too expensive and not needed all the time, so you might need to decide based on your needs.
- **Validate a Pull Request(PR)/Change Request:** The pipeline will validate that the Pull Request/Change Request changes are valid and that artifacts can be produced with the recent changes. Usually, the result of

the pipeline can be notified back to the user in charge of merging the PR and also block the merging options if the pipeline is failing. This pipeline is used to validate that whatever is merged into the main branch is valid and can be released. Validating Pull Requests / Change Requests can be an excellent option to avoid running pipelines for every change in the feature branches. When the developer(s) is ready to get feedback from the build system, it can create a PR, and that will trigger the pipeline. The pipeline would be retriggered if developers made changes on top of the Pull Request.

Despite minor differences and optimizations that can be added to these pipelines, the behavior and produced artifacts are mostly the same. These conventions and approaches rely on the pipelines executing enough tests to validate that the service being produced can be deployed to an environment.

Service Pipelines requirements

This section covers the infrastructural requirements for service pipelines to work and the contents of the source repository required for the pipeline to do its work.

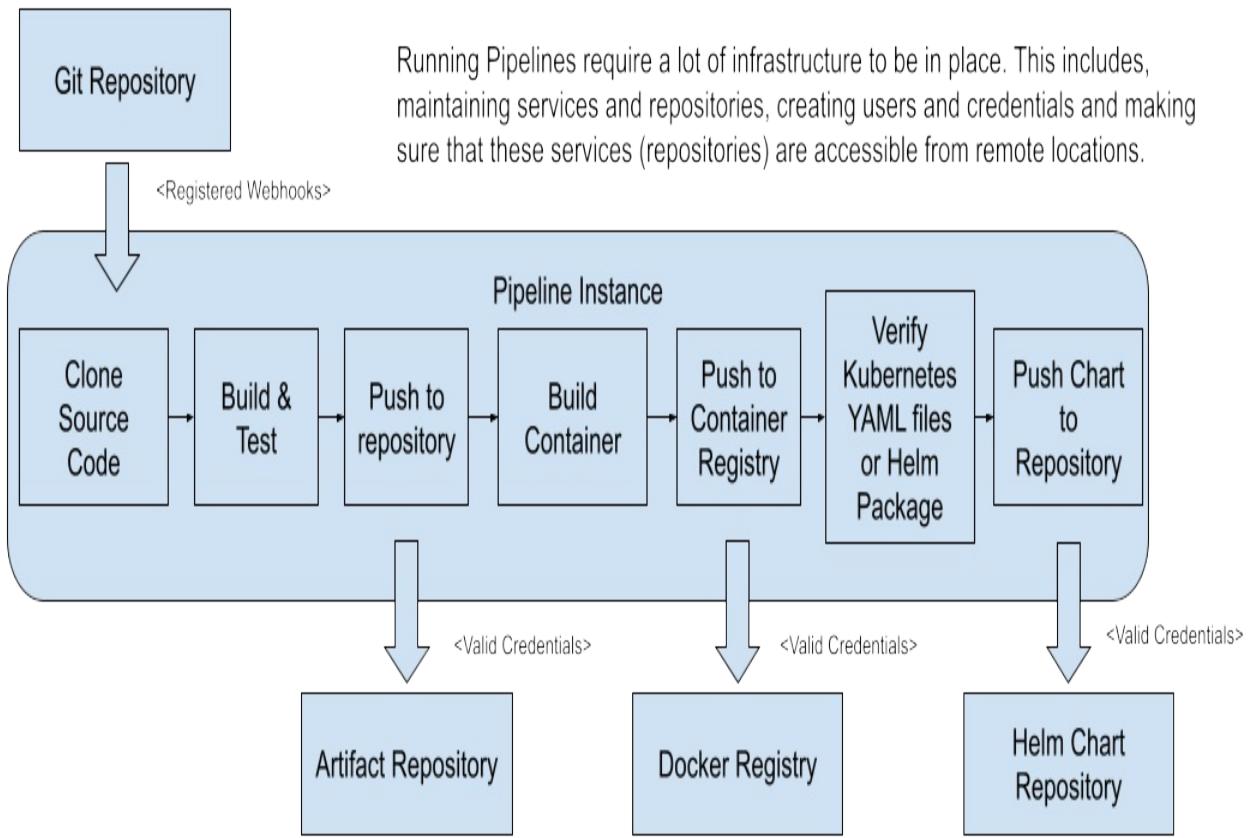
Let's start with the infrastructural requirements that a service pipeline needs to work:

- **Webhooks for source code change notifications:** First of all, it needs to have access to register webhooks to the Git repository that has the source code of the service, so a pipeline instance can be created when a new change is merged into the main branch.
- **Artifact Repository available and Valid credentials to push the binary artifacts:** once the source code is built, we need to push the newly created artifact to the artifact repository where all artifacts are stored. This requires having an artifact repository configured and valid credentials to be able to push new artifacts to it.
- **Container Registry and Valid credentials to push new container images:** in the same way as we need to push binary artifacts, we need to distribute our docker containers, so Kubernetes clusters can fetch the images when we want to provision a new instance of a service. A

container registry with valid credentials is needed to accomplish this step.

- **Helm Chart Repository and Valid Credentials:** Kubernetes manifest can be packaged and distributed as helm charts. If you are using Helm, you will need to have a Helm Chart repository and valid credentials to be able to push these packages.

Figure 3.X Service pipelines required infrastructure



For Service Pipelines to do their job, the repository containing the service's source code also needs to have a Dockerfile or the ways to produce a container image and the necessary Kubernetes manifest to be able to deploy the service into Kubernetes.

Figure 3.X The Service source code repository needs to have all the configurations for the Service pipeline to work

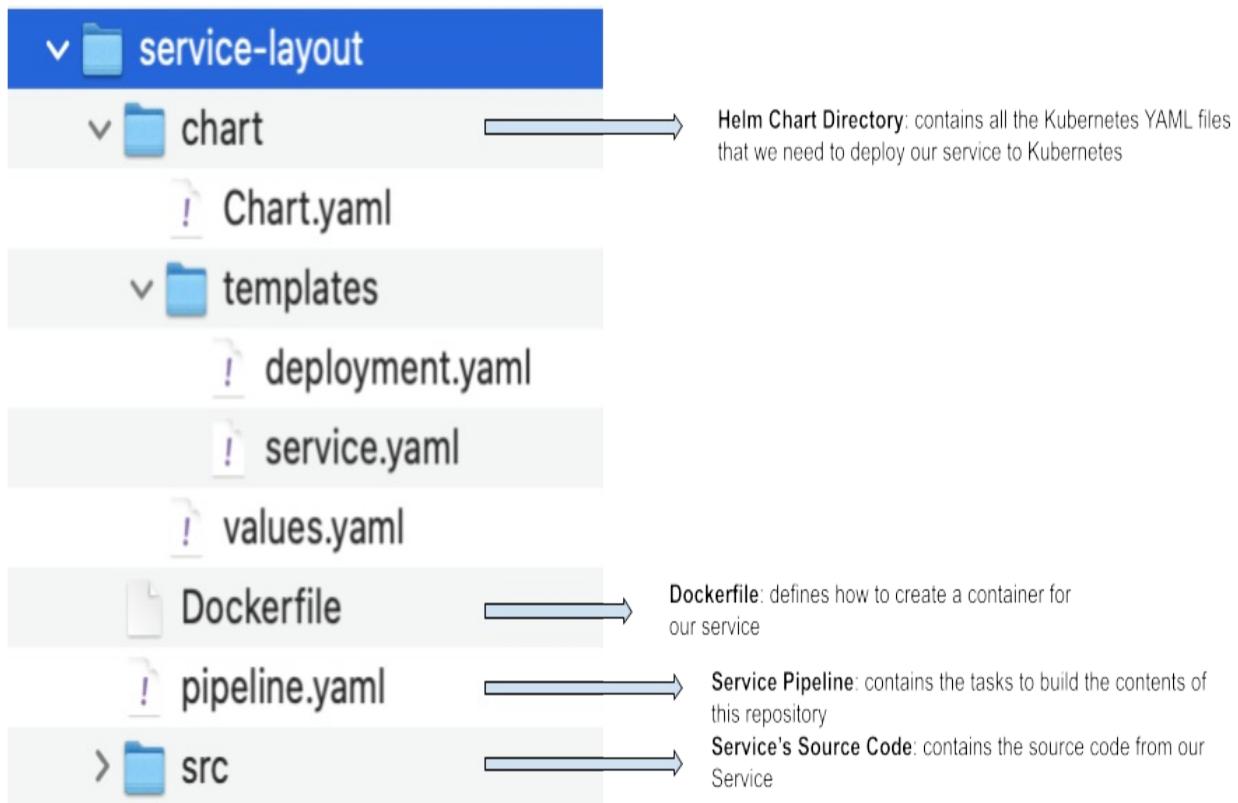


Figure 3.x shows a possible directory layout of our Service source code repository, which includes the source (src) directory containing all the files that will be compiled into binary format. The Dockerfile is used to build our container image for the service, and the Helm chart directory containing all the files to create a Helm chart that can be distributed to install the service into a Kubernetes Cluster. A common practice is to have a Helm chart definition of your service along with your service's source code, in other words, a Helm Chart per Service, as we will see in the following sections. In the same way having the Service Pipeline definition close to the service code helps to keep track of the steps needed for this service to be build.

If we include everything that is needed to build, package, and run our service into a Kubernetes Cluster, the Service pipeline just needs to run after every change in the main branch to create a new release of the service.

In summary, Service Pipelines are in charge of building our source and related artifacts to be deployed into an environment. As mentioned before, Service Pipelines are not responsible for deploying the produced Service into

a live environment, that is the responsibility of the Environment Pipeline is covered in the next chapter.

3.3 Service Pipelines in Action

There are several pipeline engines out there, even fully managed services like Github Actions (<https://github.com/features/actions>) and a number of well-known CI (continuous integration) managed services that will provide loads of integrations for you to build and package your application's services.

In the following sections, we will look at two projects: Tekton and Dagger. These projects excel at providing you with the tools to work with Cloud-Native applications and, as we will see in Chapter 6, enable platform teams to package, distribute and reuse the organization's specific knowledge that is built over time. Tekton (<https://tekton.dev>) was designed as a Pipeline Engine for Kubernetes. Because Tekton is a generic pipeline engine, you can create any pipeline with it. On the other hand, a much newer project called Dagger (<https://dagger.io>) was designed to run everywhere. We will be contrasting Tekton and Dagger with Github actions.

3.3.1 Tekton in Action

Tekton was initially created as part of the Knative project (<https://knative.dev>) from Google, initially called Knative Build, and later separated from Knative to be an independent project. Tekton's main characteristic is that it is a Cloud-Native Pipeline Engine designed for Kubernetes. In this section, we will look into how to use Tekton to define Service Pipelines.

In Tekton, you have two main concepts: Tasks and Pipelines. Tekton, the Pipeline Engine comprises a set of components that will understand how to execute `Tasks` and `Pipelines` Kubernetes Resources that we define. Tekton, as most of the Kubernetes projects covered in this report, can be installed into your Kubernetes cluster by running `kubectl` or using Helm Charts. I strongly recommend you check their official documentation page, which explains the value of using a tool like Tekton:
<https://tekton.dev/docs/concepts/overview/>

You can find a step-by-step tutorial for this section in this repository:
<https://github.com/salaboy/from-monolith-to-k8s/tree/main/tekton>

When you install Tekton, you are installing a set of Custom Resource Definitions, which are extensions to the Kubernetes APIs, which, in the case of Tekton, defines what Tasks and Pipelines are. Tekton also installs the Pipeline Engine that knows how to deal with Tasks and Pipelines resources.

You can install Tekton in your Kubernetes cluster by running the following command:

```
> kubectl apply -f https://storage.googleapis.com/tekton-releases
```

Optionally, you can also install the Tekton Dashboard by running:

```
> kubectl apply -f kubectl apply -f https://github.com/tektoncd/d
```

Once you install the Tekton release, you will see a new namespace called `tekton-pipelines` which contains the pipeline controller (the pipeline engine) and the pipeline webhook listener, which is used to listen for events coming from external sources, such as git repositories.

You can also install the `tkn` command-line tool, which helps greatly if you are working with multiple tasks and complex pipelines. You can follow the instructions for installations here: <https://github.com/tektoncd/cli>

Let's create a simple task definition in YAML. A Task in Tekton will look like a normal Kubernetes resource:

```
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: hello-world-task #A
spec:
  params:
    - name: name #B
      type: string
      description: who do you want to welcome?
      default: tekton user
  steps:
    - name: echo
      image: ubuntu #C
```

```
command:
  - echo #D
args:
  - "Hello World: $(params.name)" #E
```

You can find the Task Definition in this repository, along side a step-by-step tutorial to run it in your own cluster:

<https://github.com/salaboy/from-monolith-to-k8s/blob/main/tekton/resources/hello-world-task.yaml>

Derived from this example, you can create a task for whatever you want, as you have the flexibility to define which container to use and which commands to run. Once you have the task definition, you need to make that available to Tekton by applying this file to the cluster with `kubectl apply -f task.yaml`. By applying the file into Kubernetes, we are only making the definition available to the Tekton components in the cluster, but the task will not run.

If you want to run this task, a task can be executed multiple times. Tekton requires you to create a TaskRun resource like the following:

```
apiVersion: tekton.dev/v1
kind: TaskRun
metadata:
  name: hello-world-task-run-1
spec:
  params:
    - name: name
      value: "Building Platforms on top of Kubernetes reader!" #A
  taskRef:
    name: hello-world-task #B
```

The TaskRun resource can be found here: <https://github.com/salaboy/from-monolith-to-k8s/blob/main/tekton/task-run.yaml>

If you apply this TaskRun to the cluster (`kubectl apply -f taskrun.yaml`), the Pipeline Engine will execute this task. You can take a look at the Tekton Task in action by looking at the TaskRun resources:

```
> kubectl get taskrun
NAME          SUCCEEDED   REASON      START
```

```
hello-world-task-run-1    True          Succeeded   66s
```

If you list all the running pods you will notice that each Task creates a Pod:

```
> kubectl get pods
NAME                               READY   STATUS
hello-world-task-run-1-pod     0/1     Init:0/1   0
```

And because you have a Pod you can tail the logs to see what the Task is doing:

```
> kubectl logs -f hello-world-task-run-1-pod
Defaulted container "step-echo" out of: step-echo, prepare (init)
Hello World: Building Platforms on top of Kubernetes reader!
You just executed your first Tekton TaskRun. Congrats!
```

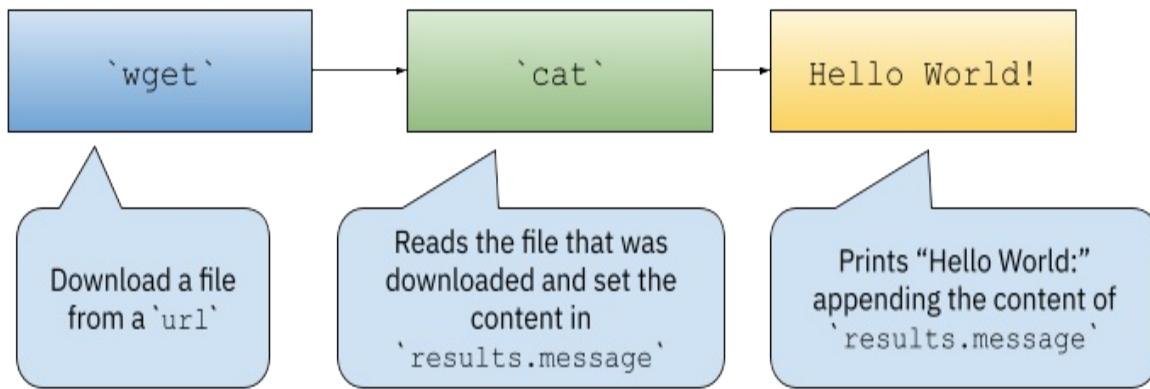
But a single Task is not interesting at all. If we can sequence multiple Tasks together we would be able to create our Service Pipelines. Let's take a look at how we can build Tekton Pipelines to build up on this simple Task example.

3.3.2 Pipelines in Tekton

A Task in itself can be helpful but Tekton becomes interesting when you create sequences of these tasks by using Pipelines.

A pipeline is a collection of these tasks in a concrete sequence. The following pipeline uses the Task definition that we have defined before it prints a message it fetches a file from a URL, then proceeds to read its content and the content is forwarded to our Hello World Task which prints a message.

Figure 3.X Simple Tekton Pipeline using our Hello World Task



In this simple pipeline we are using our previously defined Task (Hello World Task), we are fetching a Task definition (`wget`) from the Tekton Hub, which is a community repository hosting generic Tasks, and we are defining the `cat` Task inline inside the pipeline to showcase Tekton flexibility.

Let's take a look at a simple Service Pipeline defined in Tekton (`hello-world-pipeline.yaml`), don't be scared this is a lot of YAML, I warned you:

```

apiVersion: tekton.dev/v1
kind: Pipeline
metadata:
  name: hello-world-pipeline
  annotations:
    description: |
      Fetch resource from internet, cat content and then say hell
spec:
  results: #A
  - name: message
    type: string
    value: $(tasks.cat.results.messageFromFile)
  params: #B
  - name: url
    description: resource that we want to fetch
    type: string
    default: ""
  workspaces: #C
  - name: files
  tasks:
  - name: wget
    taskRef: #D

```

```

name: wget
params:
- name: url
  value: "$(params.url)"
- name: diroptions
  value:
    - "-P"
workspaces:
- name: wget-workspace
  workspace: files
- name: cat
runAfter: [wget]
workspaces:
- name: wget-workspace
  workspace: files
taskSpec: #E
workspaces:
- name: wget-workspace
results:
- name: messageFromFile
  description: the message obtained from the file
steps:
- name: cat
  image: bash:latest
  script: |
    #!/usr/bin/env bash
    cat $(workspaces.wget-workspace.path)/welcome.md | tee
- name: hello-world
runAfter: [cat]
taskRef:
  name: hello-world-task  #F
params:
- name: name
  value: "$(tasks.cat.results.messageFromFile)"  #G

```

You can find the full pipeline definition here:

<https://github.com/salaboy/from-monolith-to-k8s/blob/main/tekton/resources/hello-world-pipeline.yaml>

Before applying the Pipeline definition you need to install the `wget` Tekton Task that was created and maintained by the Tekton community:

```
kubectl apply -f https://raw.githubusercontent.com/tektoncd/catal
```

Once again, you will need to apply this pipeline resource to your cluster for

Tekton to know about: `kubectl apply -f hello-world-pipeline.yaml`.

As you can see in the pipeline definition, the `spec.tasks` field contains an array of tasks. These tasks need to be already deployed into the cluster, and the Pipeline definition is in charge of defining the sequence in which these tasks will be executed. These Task references can be your tasks, or as in the example, they can come from the Tekton Catalog, which is a repository that contains community-maintained task definitions that you can reuse.

In the same way, as Tasks need TaskRuns for the executions, you will need to create a PipelineRun for every time you want to execute your Pipeline.

```
apiVersion: tekton.dev/v1
kind: PipelineRun
metadata:
  name: hello-world-pipeline-run-1
spec:
  workspaces: #A
    - name: files
      volumeClaimTemplate:
        spec:
          accessModes:
            - ReadWriteOnce
          resources:
            requests:
              storage: 1M
  params:
    - name: url #B
      value: "https://raw.githubusercontent.com/salaboy/salaboy/main/hello-world-pipeline"
  pipelineRef:
    name: hello-world-pipeline #C
```

You can find the PipelineRun resource here:

<https://github.com/salaboy/from-monolith-to-k8s/blob/main/tekton/pipeline-run.yaml>

When you apply this file to the cluster `kubectl apply -f pipeline-run.yaml` Tekton will execute the pipeline by running all the tasks defined in the pipeline definition.

When running this Pipeline Tekton will create one Pod per Task as well as three TaskRun resources. At the end of the day, a Pipeline is just

orchestrating Tasks, or in other words creating TaskRuns.

Check that the TaskRuns were created and that the Pipeline executed successfully:

```
> kubectl get taskrun
NAME                               SUCCEEDED   REASON
hello-world-pipeline-run-1-cat    True        Succeeded
hello-world-pipeline-run-1-hello-world  True        Succeeded
hello-world-pipeline-run-1-wget   True        Succeeded
```

For each TaskRun Tekton created a Pod:

```
> kubectl get pods
NAME                               READY   STATUS
hello-world-pipeline-run-1-cat-pod 0/1    Completed
hello-world-pipeline-run-1-hello-world-pod 0/1    Completed
hello-world-pipeline-run-1-wget-pod   0/1    Completed
```

Tail the logs from the `hello-world-pipeline-run-1-hello-world-pod` to see what the task printed:

```
> kubectl logs hello-world-pipeline-run-1-hello-world-pod
Defaulted container "step-echo" out of: step-echo, prepare (init)
Hello World: Welcome internet traveller!
```

You can always look at all Task, TaskRun, Pipeline and PipelineRuns into the Tekton Dashboard.

Figure 3.X Our PipelineRun execution in Tekton Dashboard

← → C ⓘ localhost:9097/#/namespaces/default/pipelineruns/hello-world-pipeline-run-1?pipelineTask=wget&step=wget

Update

Tekton Dashboard

default X v

Tekton resources

Pipelines

PipelineRuns

Tasks

ClusterTasks

TaskRuns

CustomRuns

hello-world-pipeline-run-1 Last updated 7 minutes ago

Succeeded Tasks Completed: 3 (Failed: 0, Cancelled 0), Skipped: 0

wget

wget Completed

Duration: 0s

cat

Logs Details

hello-world

Logs Details

```
--2023-03-18 11:32:02-- https://raw.githubusercontent.com/salaboy/salaboy/main/welcome.md
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.109.133, 185.199.110.133, 185.199.111.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)(185.199.109.133):443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 28 [text/plain]
Saving to: '/workspace/wget-workspace/welcome.md'

OK          100% 380K=0s

2023-03-18 11:32:02 (380 KB/s) - '/workspace/wget-workspace/welcome.md' saved [28/28]

Step completed successfully
```

If required, you can find a step-by-step tutorial on how to install Tekton in your Kubernetes Cluster and how to run the Service pipeline at the following repository: <https://github.com/salaboy/from-monolith-to-k8s/tree/main/tekton>

At this end of the tutorial, you will find link to more complex pipelines that I've defined for each of the Conference application services. These Pipelines are way more complex, because they require to have access to external services, credentials to publish artifacts and container images and the rights to do some privileged actions inside the cluster. Check this section of the tutorial if you are interested in more details: <https://github.com/salaboy/from-monolith-to-k8s/tree/main/tekton#tekton-for-service-pipelines>

3.3.3 Tekton advantages and extras

As we have seen, Tekton is super flexible and allows you to create pretty advanced pipelines, and it includes other features, such as

- Input and Output mappings to share data between tasks
- Event triggers that allow you to listen for events that will trigger Pipelines or Tasks
- A Command-Line tool to easily interact with Tasks and Pipelines from your terminal
- A Simple Dashboard to monitor your pipelines and task executions.

Figure 3.X Tekton Dashboard a user interface to monitor your pipelines

Tekton Dashboard

Tekton resources

- Pipelines
- PipelineRuns**
- PipelineResources
- Tasks
- ClusterTasks
- TaskRuns
- Conditions
- EventListeners
- TriggerBindings
- ClusterTriggerBindings
- TriggerTemplates
- About

dashboard-release-nightly-drt9f 2 days ago

Succeeded Tasks Completed: 2 (Failed: 0, Cancelled 0, Skipped: 0) []

Task	Status
build	Completed
publish-images	Completed
create-dir-builtdashb...	Completed
create-dir-bucket-for...	Completed
create-dir-dashboard...	Completed
source-copy-dashboa...	Completed
create-dir-bucket-for...	Completed
fetch-bucket-for-das...	Completed
link-input-bucket-to...	Completed
ensure-release-dirs-...	Completed
dashboard-run-ko	Completed
copy-to-latest-bucket	Completed
tag-images	Completed
image-digest-exporte...	Completed
source-mkdir-bucket...	Completed
source-copy-bucket-f...	Completed
upload-bucket-for-da...	Completed

Logs **Status** **Details**

```

+ gcloud auth activate-service-account --key-file=/secret/release.json'
Activated service account credentials for: [release-right-meow@tekton-releases.iam.gserviceaccount.com]
+ gcloud auth configure-docker
Adding credentials for all GCR repositories.
WARNING: A long list of credential helpers may cause delays running 'docker build'. We recommend passing the registry name to -
After update, the following will be written to your Docker config file
located at [/tekton/home/.docker/config.json]:
{
  "credHelpers": {
    "gcr.io": "gcloud",
    "us.gcr.io": "gcloud",
    "eu.gcr.io": "gcloud",
    "asia.gcr.io": "gcloud",
    "staging-k8s.gcr.io": "gcloud",
    "marketplace.gcr.io": "gcloud"
  }
}

Do you want to continue (Y/n)?
Docker configuration file updated.
+ date '+%s'
+ export 'SOURCE_DATE_EPOCH=1611112427'
+ cd /workspace/go/src/github.com/tektoncd/dashboard
+ sed -i s/devel/v20210128-00551a3b76/g /workspace/go/src/github.com/tektoncd/dashboard/base/300-deployment.yaml
+ sed -i s/devel/v20210128-00551a3b76/g /workspace/go/src/github.com/tektoncd/dashboard/base/300-service.yaml
+ which ko
/usr/local/bin/ko
+ ko version
2021/01/28 03:13:47 NOTICE:
-----
Please install ko from github.com/google/ko.
For more information see:
https://github.com/google/ko/issues/258
-----
0.7.0
+ kustomize version
{Version:unknown GitCommit:$Format:RH$ BuildDate:1970-01-01T00:00:00Z GoOs:linux GoArch:amd64}
-----
```

Figure 3.x shows the community-driven Tekton Dashboard, which you can

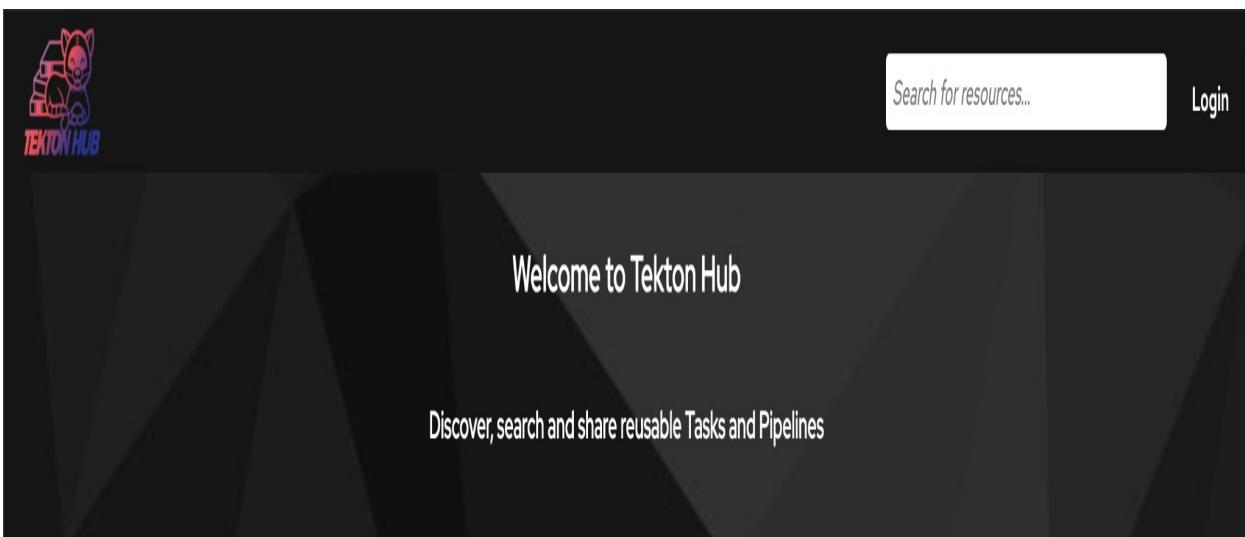
use to visualize the execution of your pipelines. Remember that because Tekton was built to work on top of Kubernetes, you can monitor your pipelines using `kubectl` as with any other Kubernetes resource. Still, nothing beats a User Interface for less technical users.

But now, if you want to implement a Service Pipeline with Tekton, you will spend quite a bit of time defining Tasks, the Pipeline, how to map inputs and outputs, defining the right events listener for your Git repositories, and then going more low-level into defining which docker images you will use for each Task. Creating and maintaining these pipelines and their associated resources can become a full-time job, and for that, Tekton launched an initiative to define a Catalog where Tasks (Pipelines and Resources are planned for future releases) can be shared, the Tekton Catalog:

<https://github.com/tektoncd/catalog>

With the help of the Tekton Catalog, we can create pipelines that reference Tasks defined in the catalog. Hence we don't need to worry about defining them. You can also visit <https://hub.tekton.dev>, which allows you to search for Task definitions and provides detailed documentation about how to install and use these tasks in your pipelines.

Figure 3.X Tekton Hub a portal to share and reuse Tasks and Pipeline definitions



Sort By ▾		Kind	Catalog	Category
<input type="checkbox"/>	Task	Buildpacks (phases) v0.2	The Buildpacks-Phases task builds source into a container image and pushes it to a registry, using Cloud Native Buildpacks. This task separately calls the aspects of the	Automation
<input type="checkbox"/>	Pipeline	curl	This task performs curl operation to transfer data from internet.	Build Tools
<input type="checkbox"/>	Tekton	EKS Cluster Teardown	Teardown an EKS cluster. This Task can be used to teardown an EKS cluster in an AWS account.	Cli
<input type="checkbox"/>		git cli	This task can be used to perform git operations. Git command that needs to be run can be passed as a script to the task. This task needs authentication to git	Cloud
<input type="checkbox"/>		v0.1	v0.1	Deploy
<input type="checkbox"/>		git cli	This task can be used to perform git operations. Git command that needs to be run can be passed as a script to the task. This task needs authentication to git	Editor
<input type="checkbox"/>		v0.1	v0.1	Git
<input type="checkbox"/>		git cli	This task can be used to perform git operations. Git command that needs to be run can be passed as a script to the task. This task needs authentication to git	Image Build
<input type="checkbox"/>		v0.1	v0.1	Language
<input type="checkbox"/>		git cli	This task can be used to perform git operations. Git command that needs to be run can be passed as a script to the task. This task needs authentication to git	Messaging
<input type="checkbox"/>		v0.1	v0.1	Monitoring
<input type="checkbox"/>		git cli	This task can be used to perform git operations. Git command that needs to be run can be passed as a script to the task. This task needs authentication to git	Notification
Sort By ▾		Buildpacks (phases) v0.2	The Buildpacks-Phases task builds source into a container image and pushes it to a registry, using Cloud Native Buildpacks. This task separately calls the aspects of the	Automation
Sort By ▾		curl	This task performs curl operation to transfer data from internet.	Build Tools
Sort By ▾		EKS Cluster Teardown	Teardown an EKS cluster. This Task can be used to teardown an EKS cluster in an AWS account.	Cli
Sort By ▾		git cli	This task can be used to perform git operations. Git command that needs to be run can be passed as a script to the task. This task needs authentication to git	Cloud
Sort By ▾		v0.1	v0.1	Deploy
Sort By ▾		git cli	This task can be used to perform git operations. Git command that needs to be run can be passed as a script to the task. This task needs authentication to git	Editor
Sort By ▾		v0.1	v0.1	Git
Sort By ▾		git cli	This task can be used to perform git operations. Git command that needs to be run can be passed as a script to the task. This task needs authentication to git	Image Build
Sort By ▾		v0.1	v0.1	Language
Sort By ▾		git cli	This task can be used to perform git operations. Git command that needs to be run can be passed as a script to the task. This task needs authentication to git	Messaging
Sort By ▾		v0.1	v0.1	Monitoring
Sort By ▾		git cli	This task can be used to perform git operations. Git command that needs to be run can be passed as a script to the task. This task needs authentication to git	Notification

Tekton Hub and the Tekton Catalog allow you to reuse Tasks and Pipelines created by a large community of users and companies. I strongly recommend you to check out their Tekton Overview page, which summarizes the advantages of using Tekton, who they recommend to use Tekton, and why: <https://tekton.dev/docs/concepts/overview/>

Tekton is quite a mature project in the Cloud-Native space, but it also presents some challenges:

- You need to install and maintain Tekton running inside a Kubernetes Cluster. You don't want your pipelines running right beside your application workloads. Hence you might need a separate cluster.
- There is no easy way to run a Tekton pipeline locally. For development purposes, you rely on having access to a Kubernetes Cluster to run a pipeline manually.
- You need to know Kubernetes to define and create tasks and pipelines.
- While Tekton provides some conditional logic, it is limited by what you can do in YAML and using a declarative approach of Kubernetes.

We will now jump into a project called Dagger that was created to mitigate some of these points, not to replace Tekton but to provide a different approach to solving everyday challenges when building complex pipelines.

3.3.4 Dagger in Action

Dagger (<https://dagger.io>) was born with one objective in mind—“**to enable developers to build pipelines using their favorite programming language that they can run everywhere**”. Dagger only relies on a container runtime to run pipelines that can be defined using code that every developer can write. Dagger currently supports Go, Python, Typescript, and Javascript SDKs, but they are quickly expanding to new languages.

Dagger is not focused on Kubernetes only; We want to ensure that while we leverage the powerful and declarative nature of Kubernetes, we also help development teams to be productive and use the appropriate tool for the job at hand. In this short section, we will examine how Dagger compares with Tekton, where it can fit better, and where it can complement other tools.

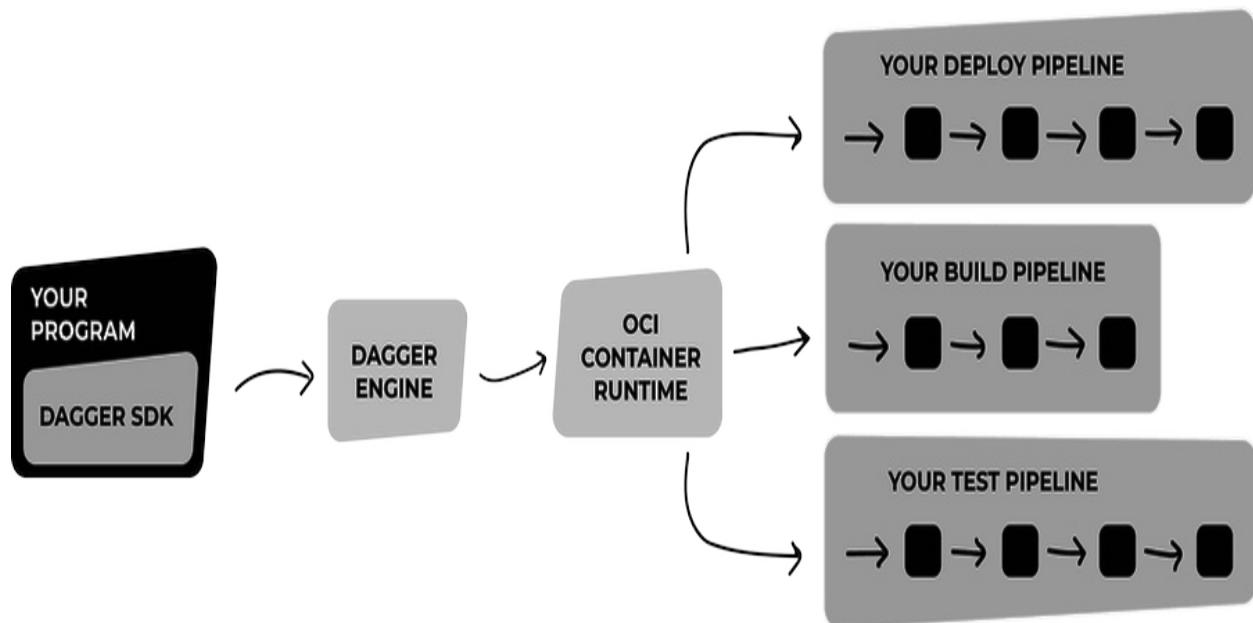
If you are interested in getting started with Dagger, you can check these resources:

- Dagger docs <https://docs.dagger.io>
- Dagger Quickstart <https://docs.dagger.io/648215/quickstart/>
- Dagger GraphQL playground <https://play.dagger.cloud>

Dagger, like Tekton, also has a Pipeline Engine, but this engine can work both locally and remotely, providing a unified runtime across environments. Dagger doesn't directly integrate with Kubernetes. This means that there are no Kubernetes CRDs or YAML involved. This can be important depending on the skills and preferences of the teams in charge of creating and maintaining these pipelines.

In Dagger, we define pipelines by writing code. Because Pipelines are just code, these pipelines can be distributed using any code packaging tools. For example, if our pipelines are written in Go, we can use Go modules to import pipelines or tasks written by other teams. If we use Java, we can use Maven or Gradle to package and distribute our pipeline libraries to promote reuse.

Figure 3.X Using your preferred programming language and its tools to write pipelines



The Dagger Pipeline Engine is then in charge of orchestrating the tasks

defined in the pipelines and optimizing what is requested by the container runtime used to execute each task.

A significant advantage of the Dagger Pipeline Engine is that it was designed from the ground up to optimize how pipelines run. Imagine that you are building tons of services multiple times a day. You will not only keep your CPUs hot but the amount of traffic downloading artifacts, again and again, becomes expensive—more if you are running on top of a Cloud Provider, which charges you based on consumption.

Dagger, similar to Tekton, uses containers to execute each task (step) in the pipeline. The pipeline engine optimizes the resource consumption by caching the results of previous executions, preventing you from re-executing tasks that were already executed using the same inputs. In addition, you can run the Dagger engine, both locally on your laptop/workstation or remotely, even inside a Kubernetes cluster.

When I compare Dagger to something like Tekton, with my developer background, I tend to like the flexibility of coding pipelines using a programming language I am familiar with. For developers to create, version, and share code is easy, as I don't need to learn any new tools.

Instead of looking at a Hello World example, I wanted to show how a Service Pipeline would look like in Dagger. So let's look at how a service pipeline defined using the Dagger Go SDK:

Listing 3.X Service pipeline implemented in Go (dagger.go)

```
func main() {
    var err error
    switch os.Args[1] {
    case "build":
        bc := buildContainer(ctx)
        _, err = bc.Directory("target").Export(ctx, "./target")
    case "publish-image":
        if len(os.Args) < 3 {
            err = fmt.Errorf("invalid number of arguments: expecte
                           break
        }
        err = buildImage(ctx, os.Args[2])
    }
}
```

```

case "publish-chart":
    if len(os.Args) < 7 {
        err = fmt.Errorf("invalid number of arguments: expecte
                           owner, repository, branch")
        break
    }
    err = publishChart(ctx, os.Args[2], os.Args[3], os.Args[4]
                       os.Args[5], os.Args[6])
case "helm-package":
    err = helmPackage(ctx)
case "full":
    ... do all previous tasks ...
default:
    log.Fatalln("Invalid task, available tasks are: build, pub
}
if err != nil {
    panic(err)
}
}

```

This simple Go program uses the Dagger Go SDK to define the different steps that can be executed to build, package, and publish the services artifacts.

By running `go run dagger.go build` Dagger will use Maven to build our application source code and then build a container ready to be pushed to a container registry.

Listing 3.X Building code using programming language-specific tools

```

func buildContainer(ctx context.Context) *dagger.Container {
    c := getDaggerClient(ctx)

    javaContainer := java.WithMaven(c)

    return javaContainer.WithExec([]string{"mvn", "package"})
}

func getDaggerClient(ctx context.Context) *dagger.Client {
    c, err := dagger.Connect(ctx, dagger.WithLogOutput(os.Stderr))
    if err != nil {
        panic(err)
    }
}

```

```
    return c
}
```

As you can see, we are using another library for dealing with Java and Maven builds in Go. While Dagger and the Open Source community will create all the basic building blocks, each organization will most likely have to create its domain-specific libraries to integrate with 3rd party or in-house/legacy systems. By focusing on enabling developers, Dagger lets you choose the right tool(s) to create these integrations. There is no need to write plugins, just code that can be distributed as any other library.

Try running the pipeline for one of the services or follow the step-by-step tutorial that you can find here:

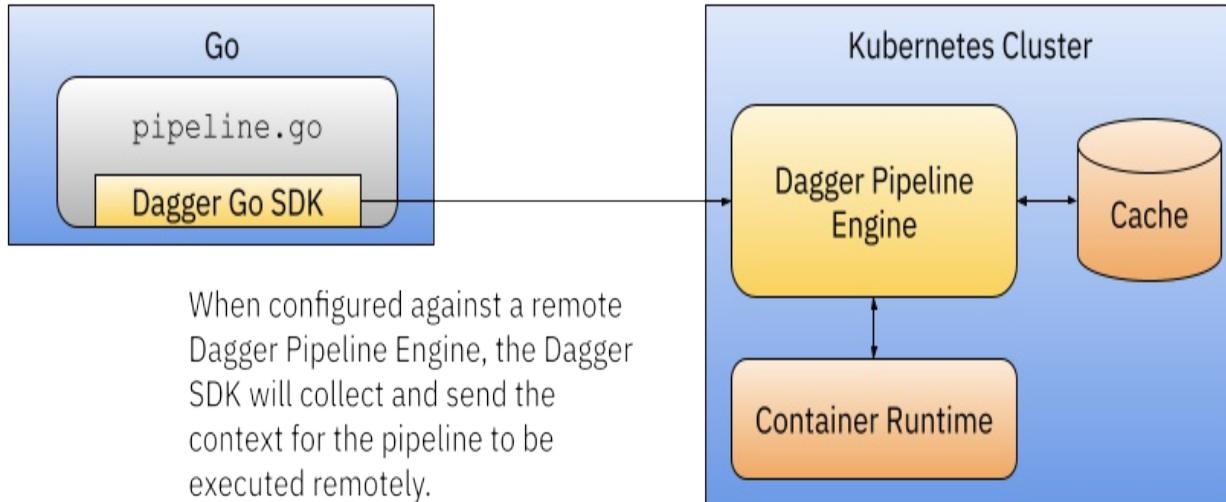
<https://github.com/salaboy/from-monolith-to-k8s/blob/main/dagger/README.md>

You will notice that if you run the pipeline twice, the second run will be almost instant since most of the steps are cached.

In contrast with Tekton, we are running the Dagger pipeline locally, not in a Kubernetes Cluster, which has it's advantages. For example, we don't need to have a Kubernetes Cluster to run and test this pipeline, but also we don't need to wait for remote feedback.

But now, how does this translate to a remote environment? What if we really want to run this pipeline remotely on a Kubernetes Cluster? The good news is that it works exactly similarly: it is just a remote Dagger Pipeline Engine that will execute our pipelines. No matter where this remote Pipeline Engine is, maybe running inside Kubernetes or as a Managed Service, our pipeline behavior and the caching mechanisms provided by the Pipeline Engine will behave the same way.

Figure 3.X Running your pipelines against a remote Dagger Engine running on Kubernetes



When we have the Dagger Pipeline Engine installed in a remote environment such as a Kubernetes Cluster, virtual machine, or any other computing resource, we can connect and run our pipelines against it.

The Dagger Go SDK takes all the context needed from the local environment and sends it over to the Dagger Pipeline Engine so the tasks can be executed remotely. We don't need to worry about publishing our application source code online for the pipeline to use.

Check this step by step tutorial on how to run your Dagger Pipeline on Kubernetes: <https://github.com/salaboy/from-monolith-to-k8s/blob/main/dagger/README.md#running-your-pipelines-remotely-on-kubernetes>

As you can see, Dagger will use persistent storage (Cache) to cache all the builds and tasks to optimize performance and reduce pipeline running times. The operations team in charge of deploying and running Dagger inside Kubernetes will need to track how much storage is needed based on the pipelines the organization is running.

In this short section, we have seen how to use Dagger to create our Service Pipelines. We have seen that Dagger is very different from Tekton: you don't need to write your pipelines using YAML, you can write your pipelines in any supported programming language, you can run your pipelines locally or remotely using the same code, and you can distribute your pipelines using the

same tools that you are using for your applications.

From a Kubernetes point of view, when you use a tool like Dagger you lose the Kubernetes native approach of managing your pipelines as you manage your other Kubernetes resources. I can see the Dagger community expanding in that direction at some point if they get enough feedback and requests for that.

From a platform engineering perspective, you can create and distribute complex pipelines (and tasks) for your teams to use and extend by relying on tools they already know. These pipelines will run the same way no matter where they are executed, making it an extremely flexible solution. Platform teams can take this flexibility to decide where to run these pipelines more efficiently (based on costs and resources), without complicating developers' lives, as they will always be able to run their pipelines locally for development purposes.

3.3.5 Should I use Tekton, Dagger or GitHub actions?

As you have seen, Tekton and Dagger provide us with the basic building blocks to construct unopinionated pipelines. In other words, we can use Tekton and Dagger to build Service pipelines and almost every imaginable pipeline. With Tekton, we leverage the Kubernetes resource-based approach, scalability, and self-healing features. Using Kubernetes-native resource can be very helpful if we want to integrate tekton with other Kubernetes tools, for example for managing and monitoring Kubernetes resources. By using the Kubernetes resource model, you can treat your Tekton Pipelines and PipelineRuns as any other Kubernetes resource and reuse all the existing tooling for that.

With Dagger, we can define our pipelines using well-known programming languages and tools, and we can run these pipelines everywhere (locally in our workstations in the same way as if we were running them remotely). This makes Tekton and Dagger perfect tools that Platform Builders can use to build more opinionated pipelines that development teams can use.

On the other hand, you can use a managed service such as Github actions.

You can look at how the Service Pipelines are configured using Github actions for all the projects mentioned here. For example, you can check the Frontend Service Pipeline here: https://github.com/salaboy/fmtok8s-frontend/blob/main/.github/workflows/ci_workflow.yml

The advantage of using GitHub actions is that you don't need to maintain the infrastructure running them or pay for the machines that run these pipelines (if your volume is small enough). But if you are running loads of pipelines and these pipelines are data-intensive, GitHub actions will be costly.

For cost-related reasons or because you cannot run your pipelines in the cloud due to industry regulations, Tekton and Dagger shine in providing you with all the building blocks to compose and run complex pipelines. While Dagger is already focused on cost and runtime optimization, I see this coming for Tekton and other pipeline engines.

It is quite important to notice that you can integrate both, Tekton and Dagger with GitHub, for example, use Tekton Triggers (<https://github.com/tektoncd/triggers/blob/main/docs/getting-started/README.md>) to react to commits into a Github repository. You can also run Dagger inside a GitHub action, enabling developers to run the same pipeline locally that what is being executed in GitHub actions, something that cannot be done easily out-of-the box.

Now that we have our artifacts and configurations ready to be deployed to multiple environments, let's look at what is commonly known as the GitOps approach for continuous deployment through environment pipelines.

3.4 Summary

- Service pipelines define how to go from source code to artifacts that can be deployed in multiple environments. In general, following Trunk-Based Development and One Service = One repository practices help your teams to standardize how to build and release software artifacts in a more efficient way.
- Tekton is a Pipeline Engine designed for Kubernetes, you can use Tekton to design your custom Pipelines and leverage all the shared

Tasks and Pipelines openly available in the Tekton Catalog. You can now install Tekton in your cluster and start creating Pipelines.

- Dagger allows you to write and distribute pipelines using your favorite programming language. These pipelines then can be executed in any environment, including your developer's laptops.

4 Environment pipelines: Deploying cloud-native applications

This chapter covers

- What it takes to deploy all the produced artifacts from Service Pipelines into different Environments
- The advantages of using the concept of Environment Pipelines and the GitOps approach to describe and manage environments
- How projects like ArgoCD/FluxCD in combination with Helm can help you to deliver software more efficiently

This chapter introduces the concept of *Environment Pipelines*. We cover the steps required to deploy the artifacts created by Service Pipelines into concrete running environments all the way to production. We will look into a common practice that has emerged in the Cloud-Native space called GitOps, which allows us to define and configure our environments using a Git repository. Finally, we will look at a project called ArgoCD which implements a GitOps approach for managing applications on top of Kubernetes.

This chapter is divided into 3 main sections:

- Environment Pipelines
 - What is an Environment Pipeline?
 - Environment Pipelines in action using ArgoCD
- Service + Environment Pipelines working together
- Jenkins X: a one-stop shop for CI/CD in Kubernetes

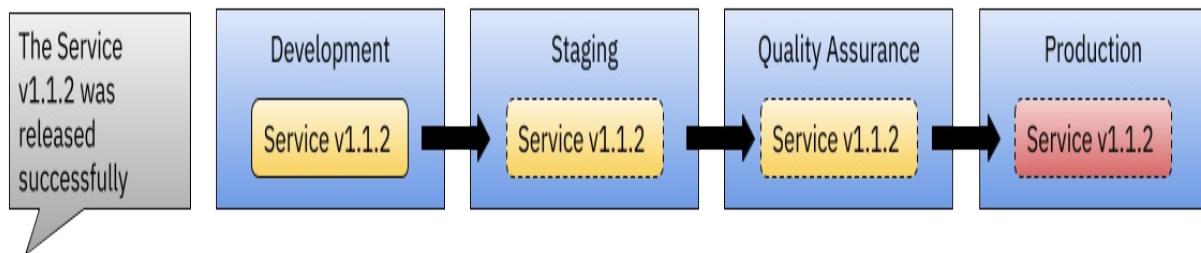
4.1 Environment Pipelines

We can build as many services as we want and produce new versions, but if these versions cannot flow freely across different environments to be tested,

and finally used by our customers, our organization will struggle to have a smooth end to end software delivery practice. Environment Pipelines are in charge of configuring and maintaining up to date our environments.

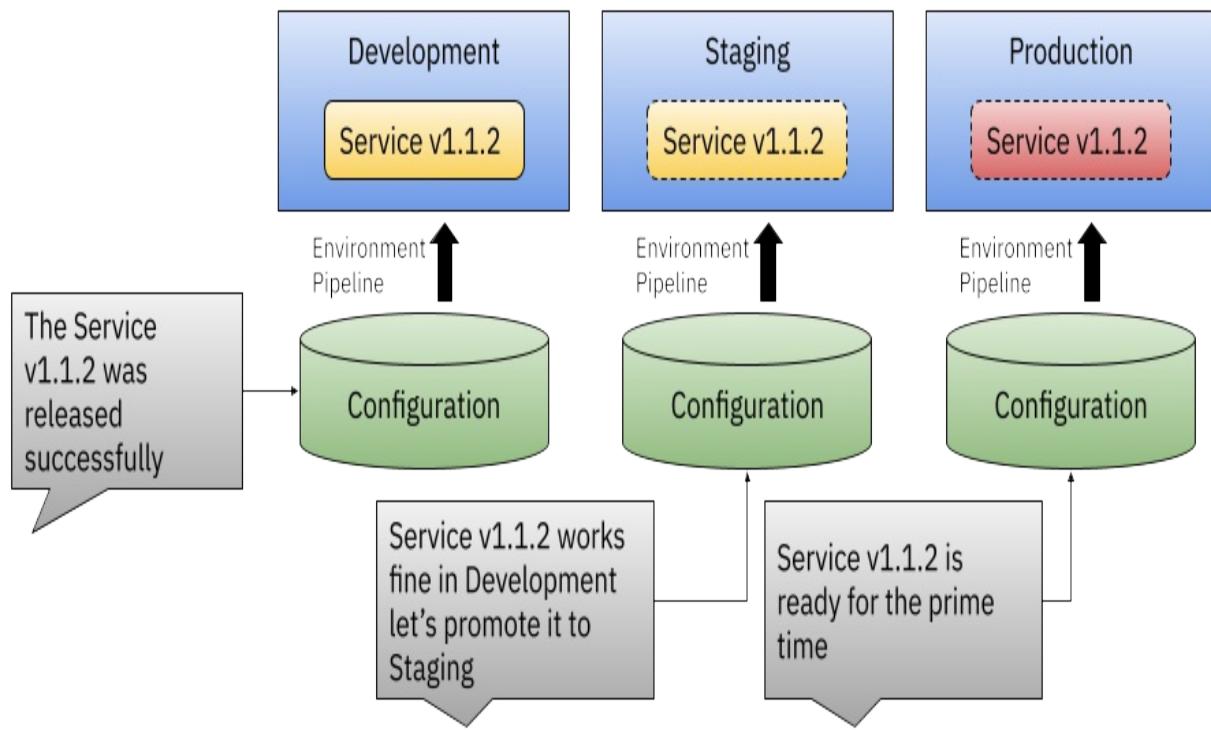
It is quite common for companies to have different environments for different purposes, for example, a Staging environment where developers can deploy their latest versions of the services or a Quality Assurance(QA) environment where manual testing happens, and one or more Production environments which are where the real users interact with our applications. These (Staging, QA, and Production) are just examples. There shouldn't be any hard limit on how many environments we can have.

Figure 4.X Released service moving throughout different environments



Each environment (Development, Staging, Quality Assurance, and Production) will have one Environment Pipeline. These pipelines will be responsible for keeping the environment configuration in sync with the hardware running the live version of the environment. These environment pipelines use as the source of truth a repository that contains the environment configurations, including which services and which version of each service needs to be deployed.

Figure 4.X Promoting services to different environments mean updating environment configurations



If you are using this approach, each environment will have its own configuration repository. Promoting a new released version means changing the environment configuration repository to add a new service or update the configuration to the new version that was released.

These configuration changes can be all automated or require manual intervention. For more sensitive environments, such as the Production environment, you might require different stakeholders to sign off before adding or updating a service.

But, where do Environment Pipelines come from? And why mightn't you have heard of them before? Before jumping into the details about how an Environment Pipeline would look like, we need to get a bit of background on why this matters in the first place.

4.1.1 How does this use to work, and what has changed lately?

Traditionally, creating new environments was hard and costly. Creating new environments on demand wasn't a thing for these two reasons. The differences between the environment that a developer used to create an

application and where the application ran for end users were completely different. These differences, not only in computing power, caused huge stress on operations teams responsible for running these applications. Depending on the environment's capabilities, they needed to fine-tune the application's configurations (that they didn't design).

Most of the time, deploying a new application or a new version of an application requires shutting down the server, running some scripts, copying some binaries, and then starting the server again with the new version running. After the server starts again, the application could fail to start. Hence more configuration tuning might be needed. Most of these configurations were done manually in the server itself, making it difficult to remember and keep track of what was changed and why.

As part of automating these processes, tools like Jenkins (<https://www.jenkins.io/>, a very popular pipeline engine) and/or scripts were used to simplify deploying new binaries. So instead of manually stopping servers and copying binaries, an operator can run a Jenkins Job defining which versions of the artifacts they wanted to deploy and Jenkins will run the job notifying the operator about the output. This approach had two main advantages:

- Tools like Jenkins can have access to the environment's credentials avoiding manual access to the servers by the operators
- Tools like Jenkins log every job execution and the parameters allowing us to keep track of what was done and the result of the execution

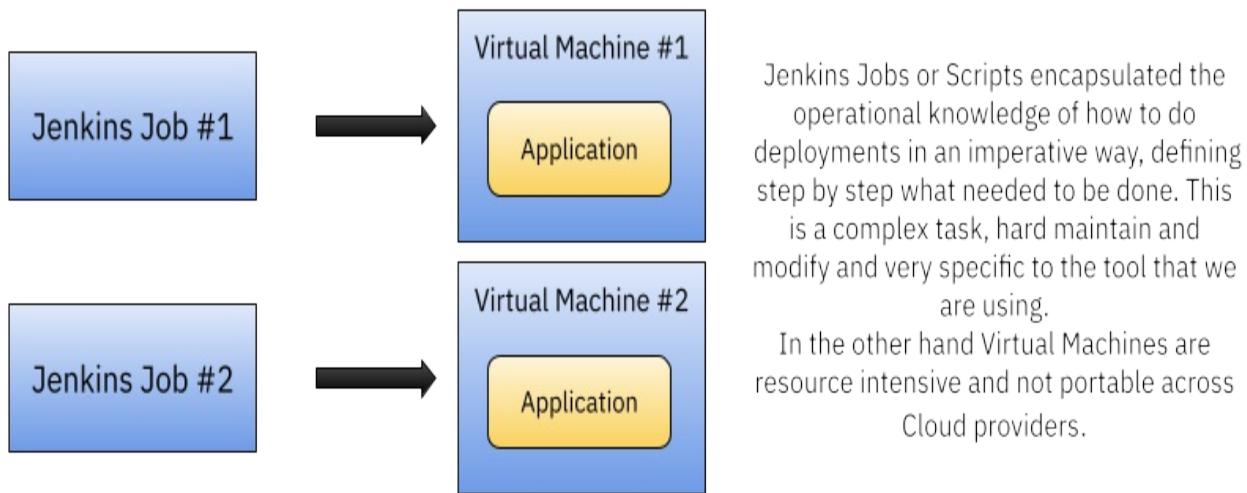
While automating with tools like Jenkins was a big improvement compared with manually deploying new versions, there were still some issues, like for example, having fixed environments that were completely different from where the software was being developed and tested. To reduce the difference between different environments even further, we needed to have a specification of how the environment is created and configured down to the operating system's version and the software installed into the machines or virtual machines. Virtual Machines helped greatly with this task, as we can easily create two or more virtual machines configured similarly.

We can even give our developers these virtual machines to work. But now we

have a new problem. We will need new tools to manage, run, maintain, and store our virtual machines. If we have multiple physical machines where we want to run virtual machines, we don't want our operations team to start these VMs in each server manually. Hence we will need a hypervisor to monitor and run VMs in a cluster of physical computers.

Using tools Jenkins and Virtual Machines (with hypervisors) was a huge improvement as we implemented some automation, operators didn't need to access servers or VMs to change configurations manually, and our environments were created using a configuration that was predefined in a fixed virtual machine configuration.

Figure 4.X Jenkins and Virtual Machines



While this approach is still common in the industry, there is a lot of room for improvement, for example, in the following areas:

- **Jenkins Jobs and scripts are imperative by nature.** Which means that they specify step by step what needs to be done. This has a great disadvantage: if something changes, let's say a server is no longer there or requires more data to authenticate against a service, the logic of the pipeline will fail, and it will need to be manually updated.
- **Virtual machines are heavy.** Every time you start a virtual machine, you start a complete instance of an Operating System. Running the Operating System processes does not add any business value; the larger

the cluster, the bigger the Operating System overhead. Running VMs in developers' environments might not be possible, depending on the VM's requirements.

- **Environments configurations are hidden and not versioned.** Most of the environment configurations and how the deployments are done are encoded inside tools like Jenkins where complex pipelines tend to grow out of control, making the changes very risky and migration to newer tools and stacks very difficult.
- **Each Cloud Provider has a non-standard way of creating Virtual Machines.** Pushing us into a vendor lock-in situation. If we created VMs for Amazon Web Services, we could not run these VMs into the Google Cloud Platform or Microsoft Azure.

How are teams approaching this with modern tooling? That is an easy question, we now have Kubernetes and Containers that aim to solve the overhead caused by VMs and the Cloud-Provider portability by relying on containers and the widely adopted Kubernetes APIs. Kubernetes also provides us with the base building blocks to ensure we don't need to shut down our servers to deploy new applications or change their configurations. If we do things in the Kubernetes way we shouldn't have any downtime in our applications.

But Kubernetes alone doesn't solve the process of configuring the clusters themselves, how we apply changes to their configurations, or how we deploy applications to these clusters. That's why you might have heard about Gitops.

What is GitOps, and how does it relate to our Environment Pipelines? We'll answer that question next.

4.1.2 What is GitOps, and how does it relate to Environment Pipelines?

If we don't want to encode all of our operational knowledge in a tool like Jenkins where it is difficult to maintain, change and keep track of it, we need a different approach.

The term GitOps, defined by the OpenGitOps group (<https://opengitops.dev/>),

defines the process of creating, maintaining, and applying the configuration of our environments and applications declaratively using Git as the source of truth. The OpenGitOps group defines four core principles that we need to consider when we talk about GitOps:

1. **Declarative:** A [system](#) managed by GitOps must have its desired state expressed [declaratively](#). We have this pretty much covered if we are using Kubernetes manifest, as we are defining what needs to be deployed and how that needs to be configured using declarative resources that Kubernetes will reconcile.
2. **Versioned and Immutable:** The desired state is [stored](#) in a way that enforces immutability, versioning and retains a complete version history. The OpenGitOps initiative doesn't enforce the use of Git, as soon as our definitions are stored, versioned and immutable we can consider it as GitOps. This opens the door to store files in for example S3 buckets, which are also versioned and immutable.
3. **Pulled Automatically:** Software agents automatically pull the desired state declarations from the source. The GitOps software is in charge of pulling the changes from the source periodically in an automated way. Users shouldn't worry about when the changes are pulled.
4. **Continuously Reconciled:** Software agents [continuously](#) observe actual system state and [attempt to apply](#) the desired state. This continuous reconciliation helps us to build resilience in our environments and the entire delivery process, as we have components that are in charge of applying the desired state and monitor our environments from configuration drifts. If the reconciliation fails, GitOps tools will notify us about the issues and keep trying to apply the changes until the desired state is achieved.

By storing the configuration of our environments and applications in a Git repository, we can track and version the changes we make. By relying on Git, we can easily roll back changes if these changes don't work as expected. GitOps not only covers the storing of the configuration but also the process of applying these configurations to the computing resources where the applications will run.

GitOps was coined in the context of Kubernetes, but this approach is not new

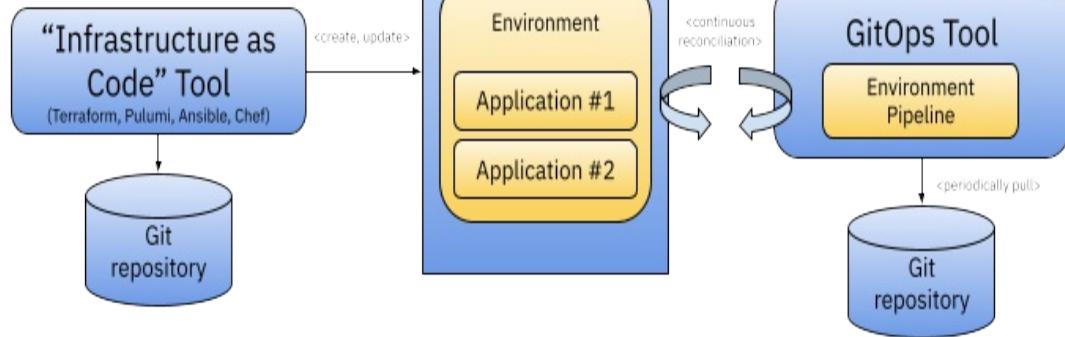
as configuration management tools have existed for a long time. With the rise of cloud providers' tools for managing Infrastructure as Code became really popular, tools like Chef, Ansible, Terraform, and Pulumi are loved by operation teams, as these tools allow them to define how to configure cloud resources and configure them together in a reproducible way. If you need a new environment you just run this Terraform Script or Pulumi app and then voila, the environment is up and running. These tools are also equipped to communicate with the Cloud Provider APIs to create Kubernetes Clusters, so we can automate the creation of these clusters.

With GitOps we are doing configuration management and relying on the Kubernetes APIs as the standard way for deploying our applications to Kubernetes Clusters. With GitOps we use a Git repository as the source of truth for our environment's internal configurations (Kubernetes YAML files) while removing the need to manually interact with the Kubernetes clusters to avoid configuration drifts and security issues. When using GitOps tools we can expect to have software agents in charge of pulling from the source of truth (git repository in this example) periodically and constantly monitoring the environment to provide a continuous reconciliation loop. This ensures that the GitOps tool will do its best to make sure that the desired state expressed in the repository is what we have in our live environments.

We can reconfigure any Kubernetes Cluster to have the same configuration that is stored in our Git repository by running an Environment Pipeline.

Figure 4.X Infrastructure as Code, GitOps and Environment Pipelines working together

Infrastructure as Code tools run scripts to create Cloud Resources in a reproducible way. We can create our Kubernetes Clusters to be all the same using these tools.

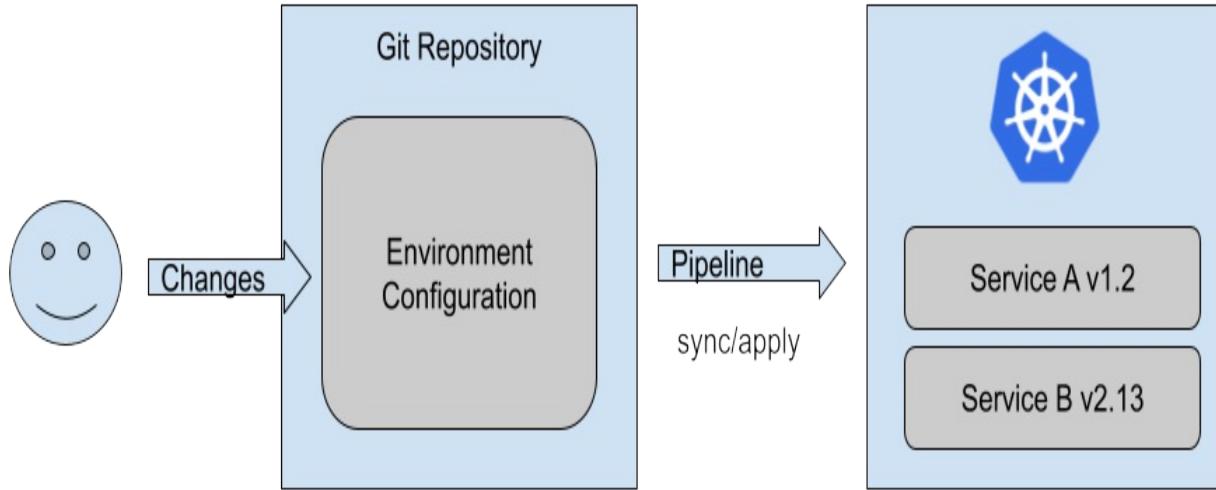


GitOps tools run environment pipelines to continuously reconcile declarative configuration which is stored in a versioned and immutable repository.

By separating the infrastructure and application concerns, our Environment Pipelines allow us to make sure that our environments are easy to reproduce and to update whenever needed. By relying on Git as the source of truth, we can roll back both our infrastructural changes and our application changes as needed. It is also important to understand that because we are working with the Kubernetes APIs, our environment definitions are now expressed in a declarative way, supporting changes in the context where these configurations are applied and letting Kubernetes deal with how to achieve the desired state expressed by these configurations.

Figure 4.x shows these interactions, where operation teams only make changes to the Git Repository that contains our environment configuration, and then a pipeline (a set of steps) is executed to make sure that this configuration is in sync with the target environment.

Figure 4.X Defining the state of the cluster using the configuration in Git (GitOps)



Environment Pipelines have the goal of monitor configurations changes from a Git Repository and apply those changes to the infrastructure, whenever a new change is detected. Following this approach allows us to rollback changes in the infrastructure by reverting commits and it also allow us to replicate the exact environment configuration by just running the same pipeline against another cluster.

When you start using Environment Pipelines, you aim to stop interacting, changing, or modifying the environment's configuration manually and all interactions are done exclusively by these pipelines. To give a very concrete example, instead of executing `kubectl apply -f` or `helm install` into our Kubernetes Cluster, a component will be in charge of running these commands based on the contents of a Git repository that has the definitions and configurations of what needs to be installed in the cluster.

In theory, a component that monitors a Git repository and reacts to changes is all you need, but in practice, a set of steps are needed to make sure that we have full control of what is deployed to our environments. Hence, thinking about GitOps as a pipeline helps us to understand that for some scenarios we will need to add extra steps to these pipelines that are triggered every time an environment configuration is changed.

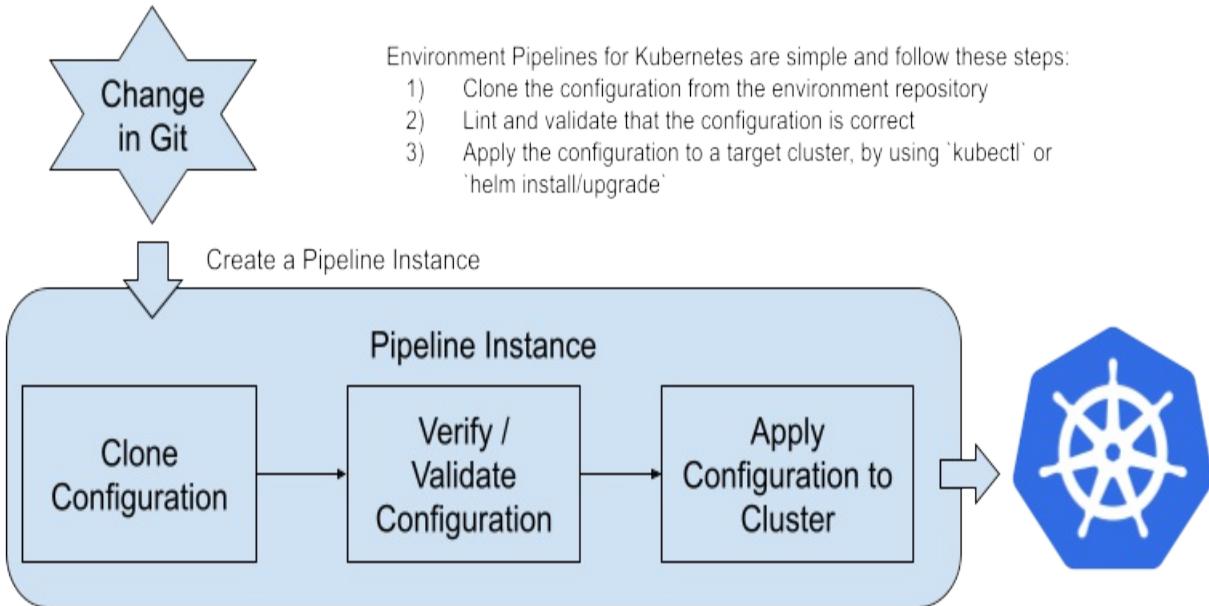
Let's look at what these steps look like with more concrete tools that we will commonly find in real-life scenarios.

4.1.3 Steps involved in an Environment Pipeline

Environment pipelines usually include the following steps:

- **Reacting to changes in the configuration:** This can be done in two different ways polling vs pushing:
 - **Polling for changes:** A component can be pulling the repository and checking if there were new commits since the last time it checked. If new changes are detected, a new environment pipeline instance is created
 - **Pushing changes using webhooks:** If the repository supports webhooks, the repository can notify our environment pipelines that there are new changes to sync.
- **Clone the source code from the repository, which contains the desired state for our environment:** this step fetches the configuration from a remote Git repository that contains the environment configurations. Tools like Git fetches only the delta between the remote repository and what we have locally.
- **Apply the desired state to a live environment:** This usually includes doing a `kubectl apply -f` or a `helm install` command to install new versions of the artifacts. Notice that with both, `kubectl` or `helm`, Kubernetes is smart enough to recognize where the changes are and only apply the differences. Once the pipeline has all the configurations locally accessible, it will use a set of credentials to apply these changes to a Kubernetes Cluster. Notice that we can fine-tune the access rights that the pipelines have to the cluster to make sure that they are not exploited from a security point of view. This also allows you to remove access from individual team members to the clusters where the services are deployed.
- **Verify that the changes are applied and that the state is matching what is described inside the git repository (deal with configuration drift):** once the changes are applied to the live cluster, checking that the new versions of services are up and running is needed to identify if we need to revert back to a previous version. In the case that we need to revert back, it is quite simple as all the history is stored in git, applying the previous version is just looking at the previous commit in the repository.

Figure 4.X Environment pipeline for a Kubernetes environment



For the Environment Pipeline to work, a component that can apply the changes to the environment is needed, and it needs to be configured accordingly with the right access credentials. The main idea behind this component is to make sure that nobody will change the environment configuration by manually interacting with the cluster. This component is the only one allowed to change the environment configuration, deploy new services, upgrade versions, or remove services from the environment.

For an Environment Pipeline to work, the following two considerations need to be met:

- The repository containing the desired state for the environment needs to have all the necessary configurations to create and configure the environment successfully.
- The Kubernetes Cluster where the environment will run needs to be configured with the correct credentials for allowing the state to be changed by the pipelines.

The term Environment Pipeline refers to the fact that each Environment will have a pipeline associated with it. As multiple environments are usually required (development, staging, production) for delivering applications, each

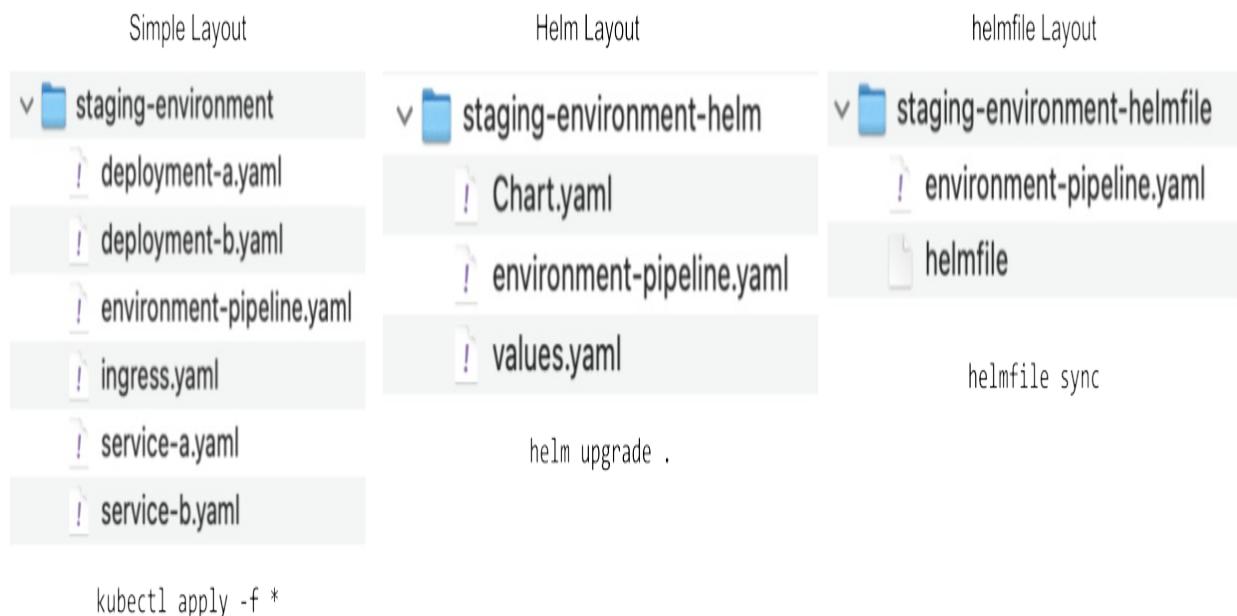
will have a pipeline in charge of deploying and upgrading the components running in them. By using this approach, promoting services between different environments is achieved by sending Pull Requests/Change Requests to the environment's repository. The pipeline will reflect the changes into the target cluster.

4.1.4 Environment Pipeline requirements and different approaches

So what are the contents of these Environment's repositories? In the Kubernetes world, an environment can be a namespace inside a Kubernetes cluster or a Kubernetes cluster itself. Let's start with the most straightforward option, a "Kubernetes namespace". As you will see in the figure that follows, the contents of the Environment Repository are just the definition of which services need to be present in the environment, the pipeline then can just apply these Kubernetes manifests to the target namespace.

The following figure shows 3 different approaches that you can use to apply configuration files to a Kubernetes Cluster. Notice that the three options all include an `environment-pipeline.yaml` file with the definition of the tasks that need to be executed.

Figure 4.X Three different approaches for defining environments' configurations



The first option (Simple layout) is just to store all the Kubernetes YAML files into a Git repository and then the Environment Pipeline will just use `'kubectl apply -f *'` against the configured cluster. While this approach is simple, there is one big drawback, if you have your Kubernetes YAML files for each service in the service repository, then the environment repository will have these files duplicated and they can go out of sync. Imagine if you have several environments you will need to maintain all the copies in sync and it might become really challenging.

The second option (Helm layout) is a bit more elaborate, now we are using Helm to define the state of the cluster. You can use Helm dependencies to create a parent chart that will include as dependencies all the services that should be present in the environment. If you do so, the environment pipeline can use `'helm update .'` to apply the chart into a cluster. Something that I don't like about this approach is that you create one Helm release per change and there are no separate releases for each service. The prerequisite for this approach is to have every service package as a Helm chart available for the environment to fetch.

The third option is to use a project called `'helmfile'` (<https://github.com/roboll/helmfile>) which was designed for this very specific purpose, to define environment configurations. A `'helmfile'`

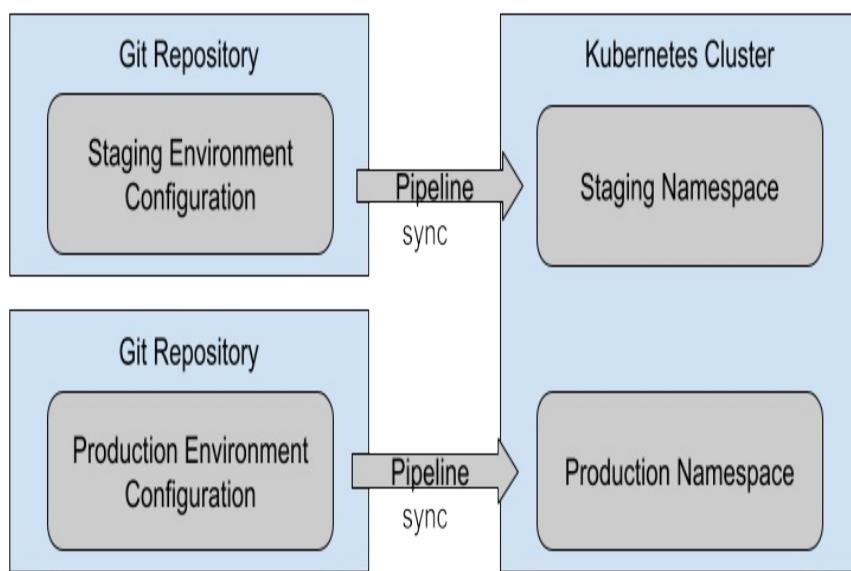
allows you to declaratively define what Helm releases need to be present in our cluster. This Helm releases will be created when we run `helmfile sync` having defined a `helmfile` containing the helm releases that we want to have in the cluster.

No matter if you use any of these approaches or other tools to do this, the expectation is clear. You have a repository with the configuration (usually one repository per environment) and a pipeline will be in charge of picking up the configuration and using a tool to apply it to a cluster.

It is common to have several environments (staging, qa, production), even allowing teams to create their own environments on-demand for running tests or day-to-day development tasks.

If you use the “one environment per namespace” approach, as shown in figure 4.x, it is common to have a separate git repository for each environment, as it helps to keep access to environments isolated and secure. This approach is simple, but it doesn’t provide enough isolation on the Kubernetes Cluster, as Kubernetes Namespaces were designed for logical partitioning of the cluster, and in this case, the Staging Environment will be sharing with the Production environment the cluster resources.

Figure 4.X One Environment per Kubernetes Namespace

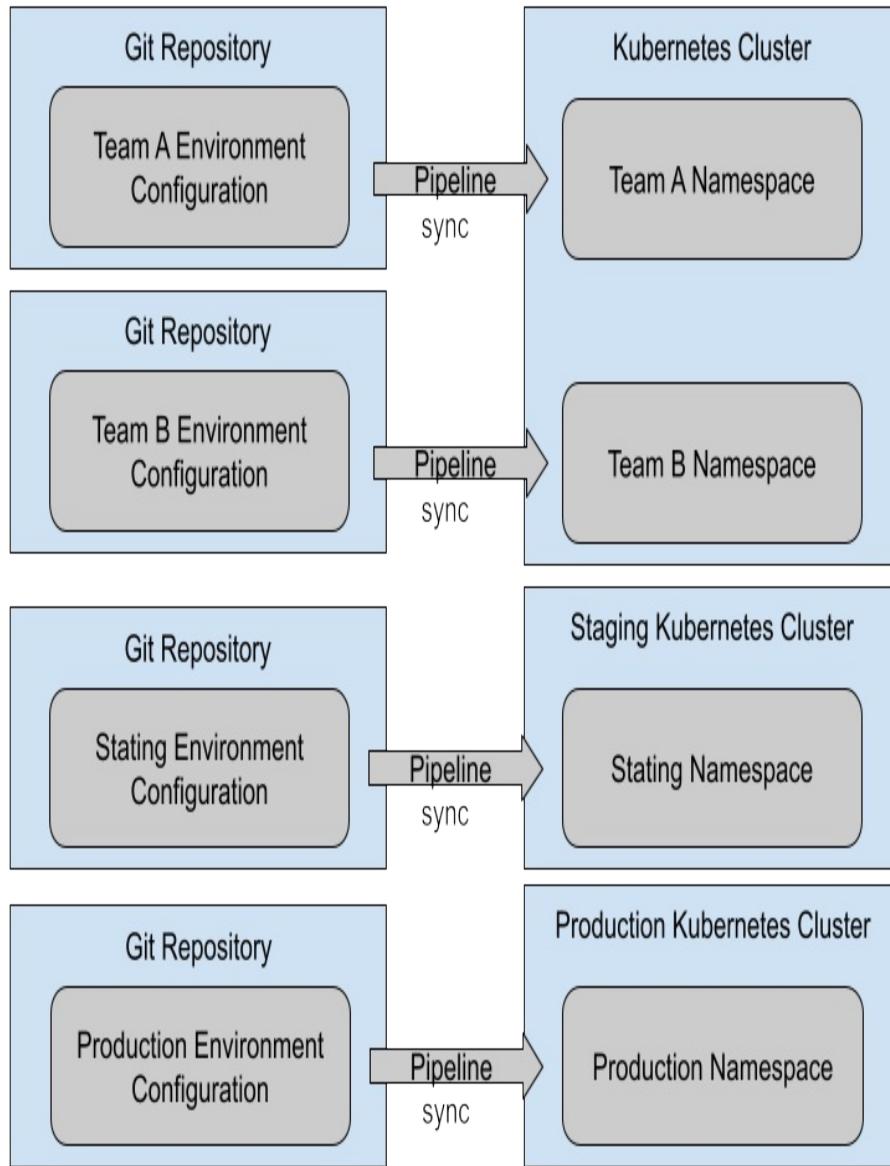


One strategy is to use namespaces for different environments. While this simplifies the configurations required for the pipelines to deploy services to different environments, namespaces doesn't have strong isolation guarantees.

An alternative approach can be to use an entirely new cluster for each environment. The main difference is isolation and access control. By having a cluster per environment you can be more strict in defining who and which components can deploy and upgrade things in these environments and have different hardware configurations for each cluster, such as multi-region setups and other scalability concerns that might not make sense to have in your Staging and Testing environments. By using different clusters you can also aim for a multi-cloud setup, where different environments can be hosted by different cloud providers.

Figure 4.x shows how you can use the namespace approach for Development Environments which will be created by different teams and then have separated clusters for Staging and Production. The idea here is to have the Staging and Production cluster configured as similarly as possible, so applications deployed behave in a similar way.

Figure 4.X One Environment per Kubernetes Cluster



A more realistic approach can be to use the same cluster for different teams day to day work and have separate clusters for Staging and Production Environments.

New Teams can be added by just copying another's team repository and creating a new namespace.

For a service to be promoted to the Staging or Production environment, Pull Requests/Merge Requests can be sent to the specific environment git repository.

Okay, but how can we implement these pipelines? Should we implement these pipelines using Tekton? In the next section, we will look at ArgoCD (<https://argo-cd.readthedocs.io/en/stable/>), a tool that has encoded the environment pipeline logic and best practices into a very specific tool for Continuous Deployment.

4.2 Environment Pipelines in Action

You can definitely go ahead and implement an Environment Pipeline as

described in the previous section using Tekton or Dagger. This has been done in projects like Jenkins X (<https://jenkins-x.io>) but nowadays, the steps for an Environment Pipeline are encoded in specialized tools for Continuous Deployment like ArgoCD (<https://argo-cd.readthedocs.io/en/stable/>).

In contrast with Service Pipelines, where we might need specialized tools to build our artifacts depending on which technology stack we use, Environment Pipelines for Kubernetes are well-standardized today under the GitOps umbrella.

Considering that we have all our artifacts being built and published by our Service Pipelines, the first thing that we need to do is to create our Environment Git repository, which will contain the environment configuration, including the services that will be deployed to that environment.

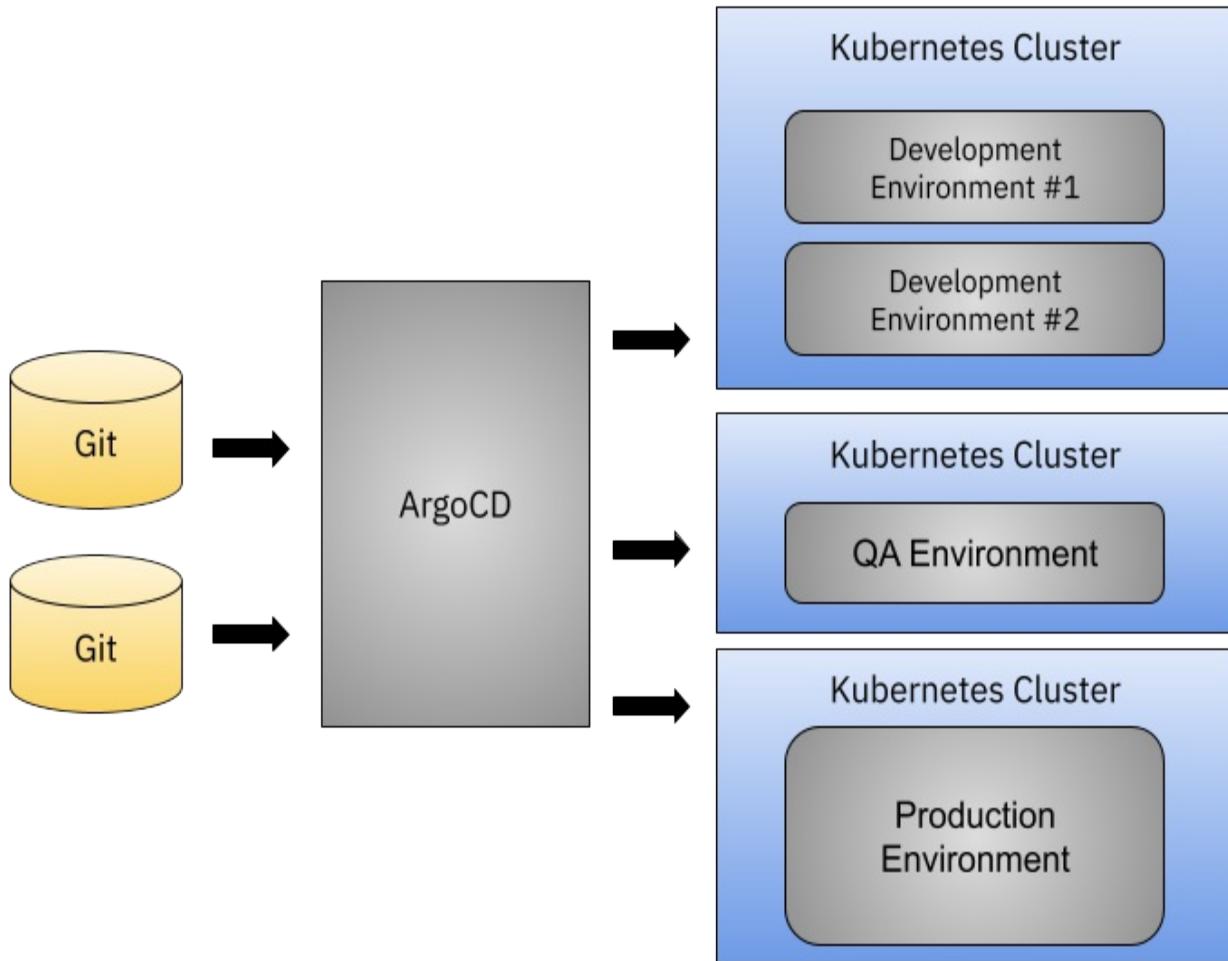
4.2.1 ArgoCD

ArgoCD provides a very opinionated but flexible GitOps implementation. When using ArgoCD we will be delegating all the steps required to deploy software into our environments continuously. ArgoCD can out-of-the-box monitor a git repository that contains our environment(s) configuration and periodically apply the configuration to a live cluster. This enables us to remove manual interactions with the target clusters, which reduces configuration drifts as git becomes our source of truth.

Using tools like ArgoCD allows us to declaratively define what we want to install in our environments, while ArgoCD is in charge of notifying us when something goes wrong, or our clusters are out of sync.

ArgoCD is not limited to a single cluster, meaning that we can have our environment living in separate clusters, even in different cloud providers.

Figure 4.X ArgoCD will sync environments configurations from git to live clusters



In the same way that we now have separate Service Pipelines for each service, we can have separate repositories, branches, or directories to configure our environments. ArgoCD can monitor repositories or directories inside repositories for changes to sync our environments configurations.

For this example, we will install ArgoCD in our Kubernetes Cluster and we will configure our Staging environment using a GitOps approach. For that, we need a git repository that serves as our source of truth.

You can follow a step-by-step tutorial located at:

<https://github.com/salaboy/from-monolith-to-k8s/tree/main/argocd>

For installing ArgoCD I recommend you to check their getting started guide that you can find here: https://argocd.readthedocs.io/en/stable/getting_started/ This guide installs all the

components required for ArgoCD to work hence after finishing this guide we should have all we need to get our Staging environment going.

The installation also guides you to install the `argocd` CLI (Command-Line Interface), which sometimes is very handy. In the following sections, we will focus on the User Interface, but you can access pretty much the same functionality by using the CLI.

ArgoCD comes with a very useful User Interface that lets you monitor at all times how your environments and applications are doing and quickly find out if there are any problems.

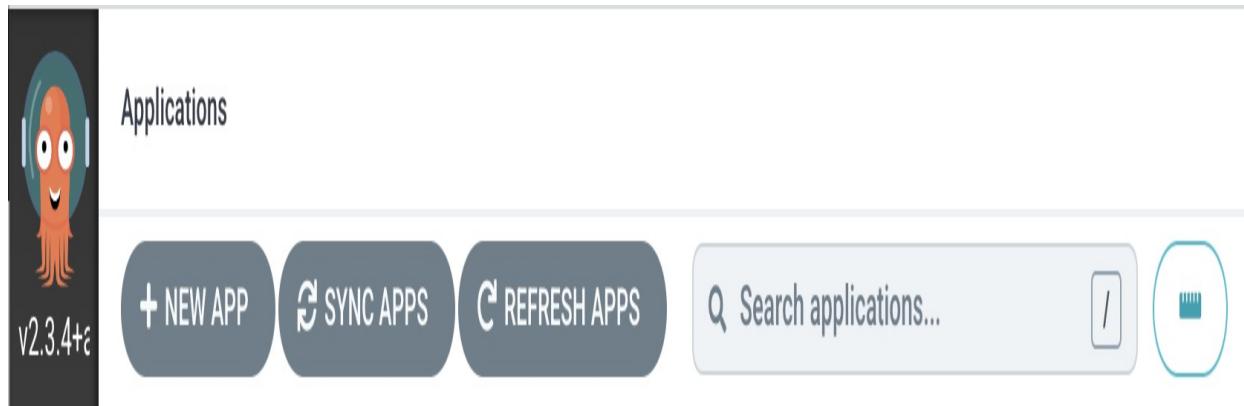
The main objective of this section is to replicate what we did in section 1.3 where we installed and interacted with the application, but here we are aiming to fully automate the process for an environment that will be configured using a git repository. Once again, we will use Helm to define the environment configuration as ArgoCD provides an out-of-the-box Helm integration.

Note: ArgoCD used a different nomenclature than the one that we have been using here. In ArgoCD you configure Applications instead of Environments. In the following screenshots, you will see that we will be configuring an ArgoCD application to represent our staging environment. As there are no restrictions on what you can include in a Helm Chart, we will be using a Helm Chart to configure our Conference application into this environment.

4.2.2 Creating an ArgoCD application

If you access the ArgoCD user interface, you will see right in the top left corner of the screen the “+ New App” button:

Figure 4.X ArgoCD User Interface - New Application Creation



Go ahead and hit that button to see the Application creation form. Besides adding a name and selecting a Project where our ArgoCD application will live (we will select the `default` project) and we will check the `Auto-Create Namespace` option.

Figure 4.X New Application parameters, manual sync, and auto-create namespace

CREATE **CANCEL** X

GENERAL

Application Name

staging

Project

default

SYNC POLICY

Manual ▾

SYNC OPTIONS

SKIP SCHEMA VALIDATION

AUTO-CREATE NAMESPACE

PRUNE LAST

APPLY OUT OF SYNC ONLY

RESPECT IGNORE DIFFERENCES

PRUNE PROPAGATION POLICY: foreground ▾

REPLACE ⚠

RETRY

By associating our environment to a new namespace in our cluster, we can use the Kubernetes RBAC mechanism only to allow administrators to modify the Kubernetes resources located in that namespace. Remember that by using ArgoCD, we want to ensure that developers don't accidentally change the application configuration or manually apply configuration changes to the cluster. ArgoCD will take care of syncing the resources defined in a Git repository. So where is that Git repository? That's exactly what we need to configure next:

Figure 4.X ArgoCD Application's configuration repository, revision and path

SOURCE

Repository URL

<https://github.com/salaboy/from-monolith-to-k8s/>

GIT ▾

Revision

HEAD

Branches ▾



Path

argocd/staging/

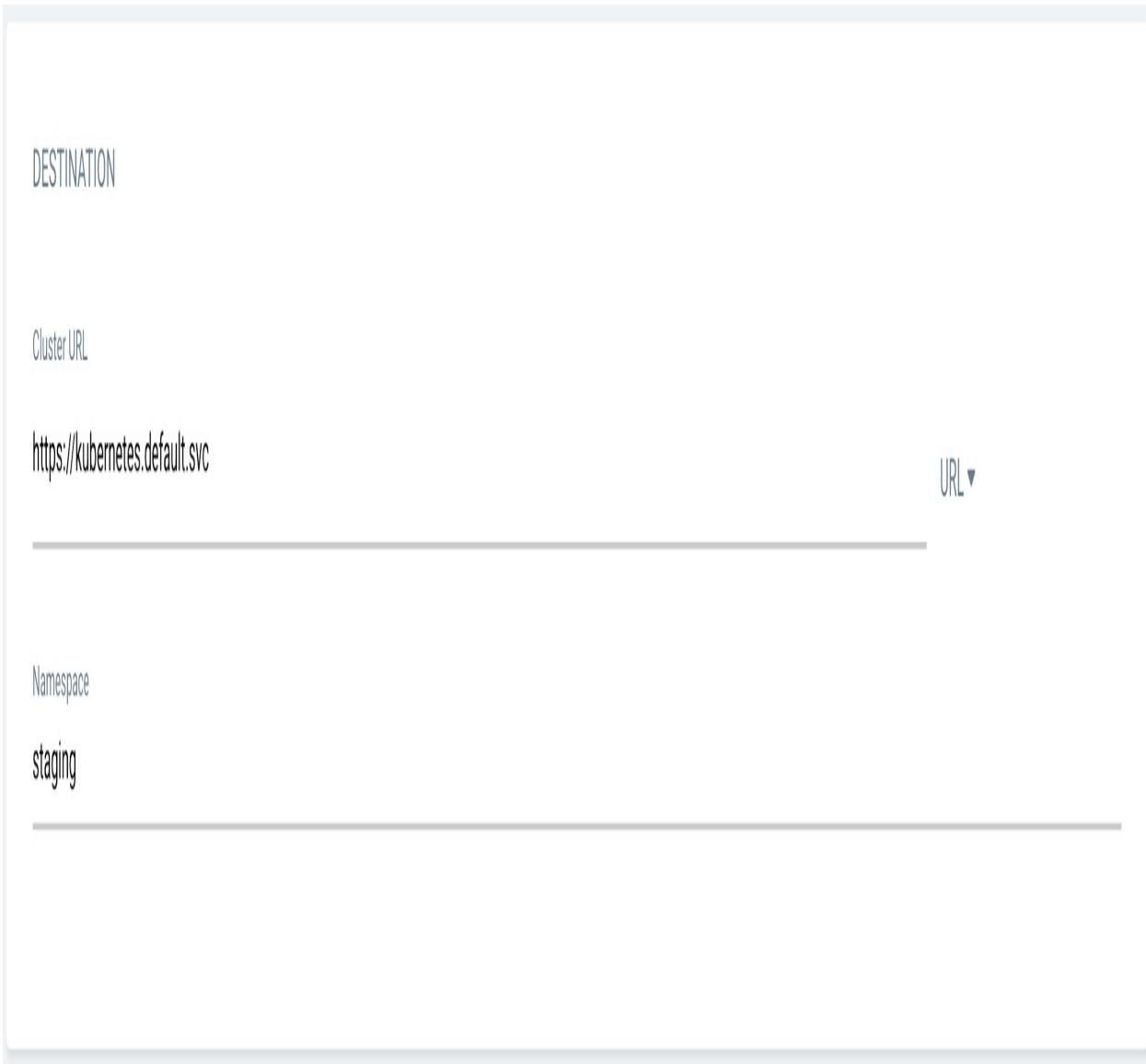
As mentioned before, we will be using a directory inside the
<https://github.com/salaboy/from-monolith-to-k8s/> repository to define our

staging environment. My recommendation is for you to fork this repository (and then use your fork URL) so you can make any changes that you want to the environment configuration.

The directory that contains the environment configuration can be found under `argocd/staging/`. As you can see you can also select between different branches and tags, allowing you to have fine-grain control of where the configuration is coming from and how that configuration evolves.

The next step is to define where this environment configuration will be applied by ArgoCD. As mentioned before we can use ArgoCD to install and sync environments in different clusters, but for this example we will be using the same Kubernetes Cluster where we installed ArgoCD, because the namespace will be automatically created, make sure to enter `staging` as the namespace so ArgoCD creates it for you.

Figure 4.X Configuration destination, for this example, is the cluster where ArgoCD is installed.



Finally, because it makes sense to reuse the same configuration for similar environments, ArgoCD enables us to configure different parameters specific to this installation. Since we are using Helm and the ArgoCD User Interface is smart enough to scan the content of the repository/path that we have entered, it knows that it is dealing with a Helm Chart. If we were not using a Helm Chart, ArgoCD allows us to set up Environment Variables that we can use as parameters for our configuration scripts.

Figure 4.X Helm configuration parameters for the Staging Environment



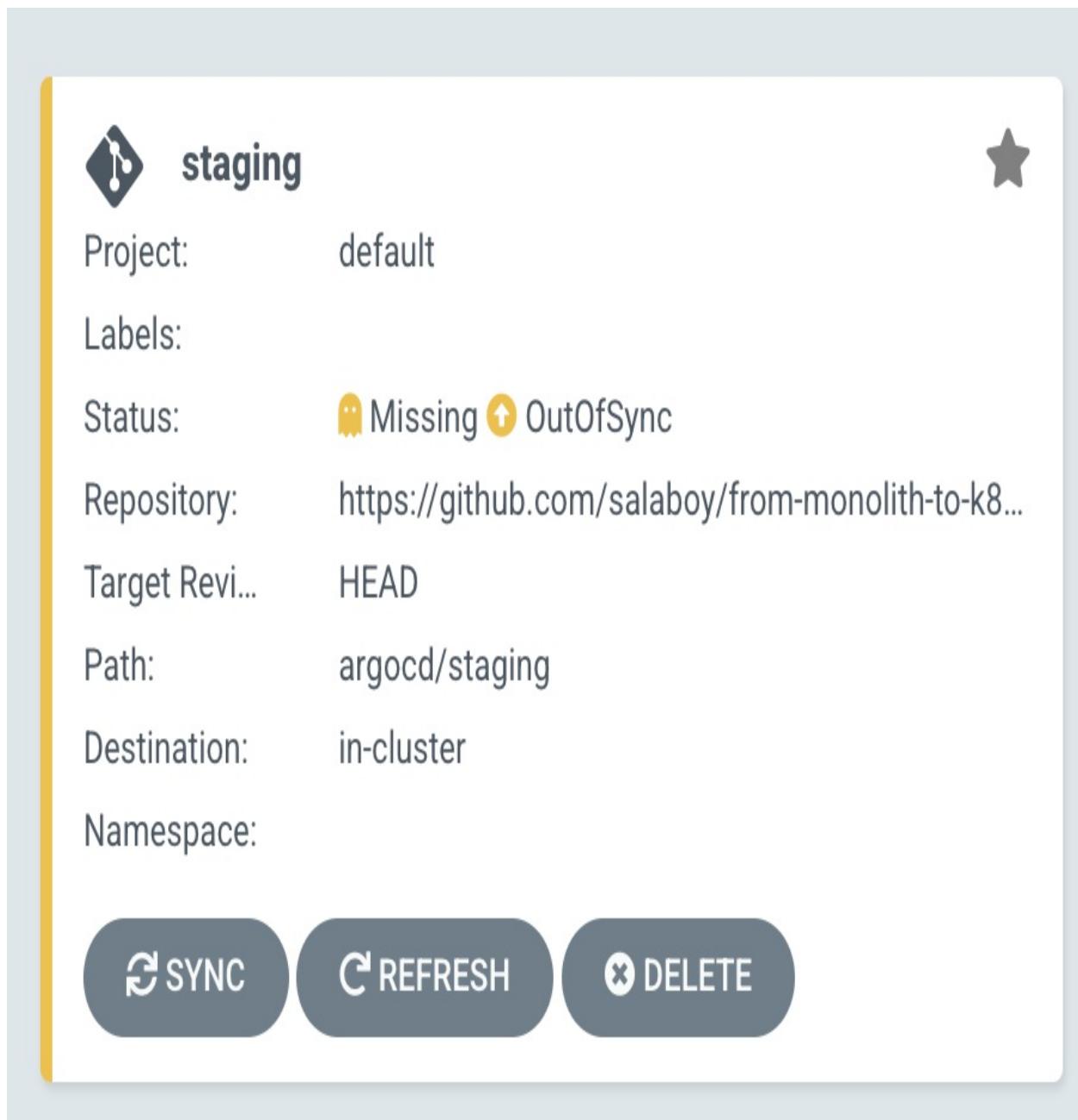
As you can see in the previous image, ArgoCD also identified that there are

non-empty `values.yaml` file inside the repository path that we have provided and it is automatically parsing the parameters. We can add more parameters to the `VALUES` text box to override any other chart (or sub-charts) configurations.

After we provide all this configuration, we are ready to hit the “create” button at the top of the form.

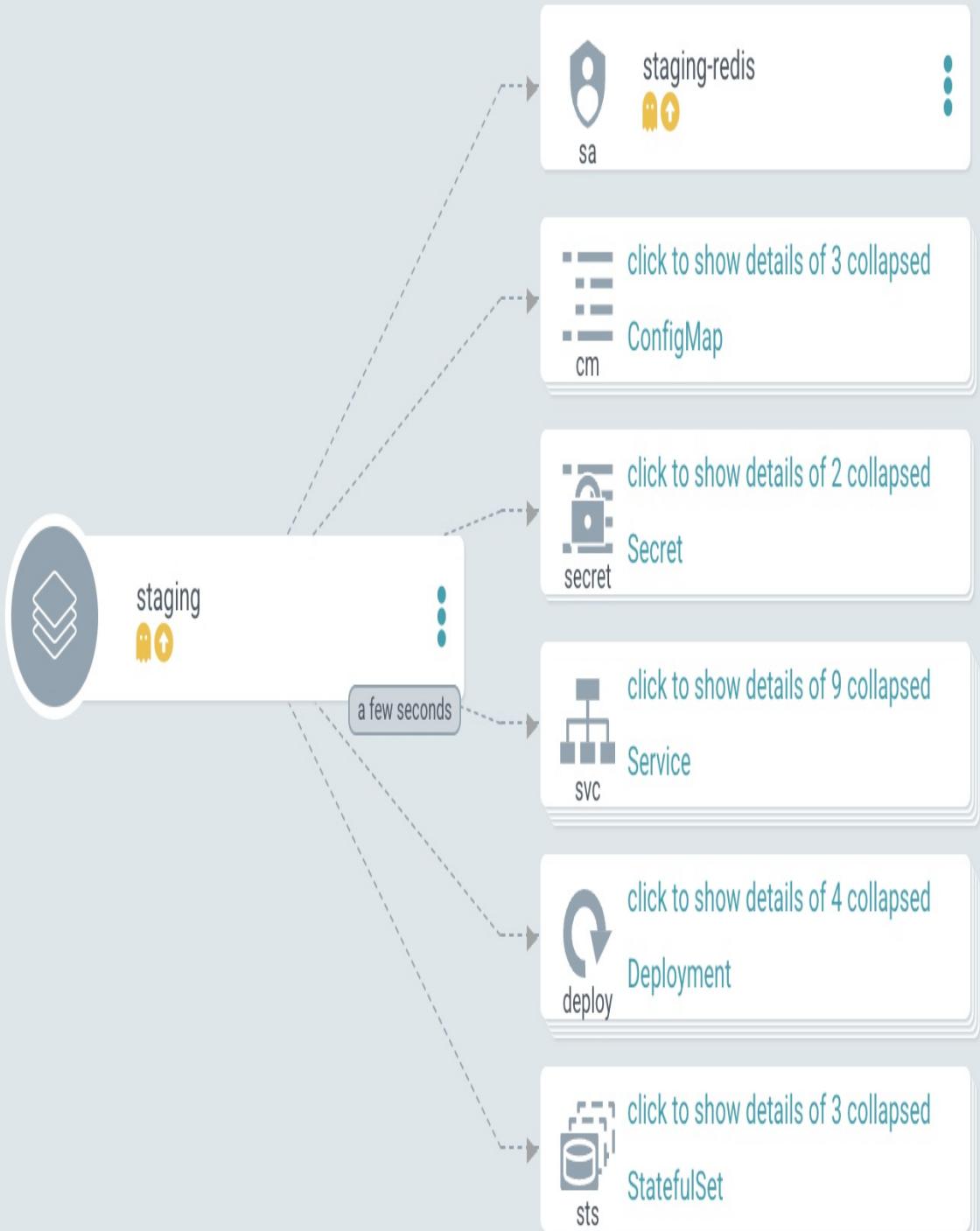
ArgoCD will create the application, but because we have selected Manual Sync it will not automatically apply the configuration to the cluster.

Figure 4.X Application created but not synced



If you click into the application you will drill down to the Application's full view which shows you the state of all the resources associated with the application.

Figure 4.X Application resources before sync



On the top menu, you will find the `Sync` button which allows you to parameterize which resources to sync and some other parameters that can influence how the resources are applied to the target namespace.

Figure 4.X Application Sync parameters

SYNCHRONIZE

CANCEL

X

Synchronizing application manifests from <https://github.com/salaboy/from-monolith-to-k8s/>

Revision

HEAD

PRUNE DRY RUN APPLY ONLY FORCE

SYNC OPTIONS

SKIP SCHEMA VALIDATION AUTO-CREATE NAMESPACE
 PRUNE LAST APPLY OUT OF SYNC ONLY
 RESPECT IGNORE DIFFERENCES

PRUNE PROPAGATION POLICY: **foreground** ▾

REPLACE ⚠

RETRY

SYNCHRONIZE RESOURCES:

[all](#) / [out of sync](#) / [none](#)

- /CONFIGMAP/STAGING/STAGING-REDIS-CONFIGURATION 
- /CONFIGMAP/STAGING/STAGING-REDIS-HEALTH 
- /CONFIGMAP/STAGING/STAGING-REDIS-SCRIPTS 
- /SECRET/STAGING/STAGING-POSTGRESQL 
- /SECRET/STAGING/STAGING-REDIS 
- /SERVICE/STAGING/FMTOK8S-AGENDA 
- /SERVICE/STAGING/FMTOK8S-C4P 
- /SERVICE/STAGING/FMTOK8S-EMAIL 
- /SERVICE/STAGING/FMTOK8S-FRONTEND 
- /SERVICE/STAGING/STAGING-POSTGRESQL 
- /SERVICE/STAGING/STAGING-POSTGRESOL-HL 

As we mentioned before, if we want to use Git as our source of truth, we should be syncing all the resources every time that we sync our configuration to our live cluster. For this reason, the `all` selection makes a lot of sense and it is the default and selected option.

After a few seconds, you will see the resources created and monitored by ArgoCD.

Figure 4.X Our Staging environment is Healthy, and all the services are up and running.

Applications / Q staging

APPLICATION DETAILS TREE

v2.3.4c

APP DETAILS **APP DIFF** **SYNC** **SYNC STATUS** **HISTORY AND ROLLBACK** **DELETE** **REFRESH**

Log out

APP HEALTH **Healthy** **CURRENT SYNC STATUS** **Synced** **LAST SYNC RESULT** **Sync OK**

To HEAD (cb51daf) To cb51daf

Author: salaboy <Salaboy@gmail.com> - Comment: updating subchart value

Author: salaboy <Salaboy@gmail.com> - Comment: updating subchart value

FILTERS

- NAME**:
- KINDS**:
- SYNC STATUS**: Synced OutOfSync
- HEALTH STATUS**: Healthy Progressing Degraded Suspended Missing Unknown

```

graph TD
    staging[staging] --> stagingRedisHeadless[staging-redis-headless]
    staging --> stagingRedisMaster[staging-redis-master]
    staging --> stagingRedisReplicas[staging-redis-replicas]
    staging --> stagingRedis[staging-redis]
    staging --> stagingFmtok8sAgendaService[staging-fmtok8s-agenda-service]
    staging --> stagingFmtok8sC4PService[staging-fmtok8s-c4p-service]
    staging --> stagingFmtok8sEmailService[staging-fmtok8s-email-service]
    staging --> stagingFmtok8sFrontend[staging-fmtok8s-frontend]

    stagingRedisHeadless --> stagingRedisHeadlessEp[staging-redis-headless-ep]
    stagingRedisMaster --> stagingRedisMasterEp[staging-redis-master-ep]
    stagingRedisReplicas --> stagingRedisReplicasEp[staging-redis-replicas-ep]
    stagingRedis --> stagingRedisSa[staging-redis-sa]
    stagingFmtok8sAgendaService --> stagingFmtok8sAgendaServiceDeploy[staging-fmtok8s-agenda-service-deploy]
    stagingFmtok8sC4PService --> stagingFmtok8sC4PServiceDeploy[staging-fmtok8s-c4p-service-deploy]
    stagingFmtok8sEmailService --> stagingFmtok8sEmailServiceDeploy[staging-fmtok8s-email-service-deploy]
    stagingFmtok8sFrontend --> stagingFmtok8sFrontendDeploy[staging-fmtok8s-frontend-deploy]

    stagingRedisHeadlessEp --> stagingRedisHeadlessEs[staging-redis-headless-es]
    stagingRedisMasterEp --> stagingRedisMasterEs[staging-redis-master-es]
    stagingRedisReplicasEp --> stagingRedisReplicasEs[staging-redis-replicas-es]
    stagingRedisSa --> stagingRedisTokenCnnvx[staging-redis-token-cnnvx]
    stagingFmtok8sAgendaServiceDeploy --> stagingFmtok8sAgendaServiceRs[staging-fmtok8s-agenda-service-rs]
    stagingFmtok8sC4PServiceDeploy --> stagingFmtok8sC4PServiceRs[staging-fmtok8s-c4p-service-rs]
    stagingFmtok8sEmailServiceDeploy --> stagingFmtok8sEmailServiceRs[staging-fmtok8s-email-service-rs]
    stagingFmtok8sFrontendDeploy --> stagingFmtok8sFrontendRs[staging-fmtok8s-frontend-rs]

    stagingRedisHeadlessEs --> stagingRedisHeadlessPod[staging-redis-headless-pod]
    stagingRedisMasterEs --> stagingRedisMasterPod[staging-redis-master-pod]
    stagingRedisReplicasEs --> stagingRedisReplicasPod[staging-redis-replicas-pod]
    stagingRedisTokenCnnvx --> stagingRedisTokenCnnvxPod[staging-redis-token-cnnvx-pod]
    stagingFmtok8sAgendaServiceRs --> stagingFmtok8sAgendaServicePod[staging-fmtok8s-agenda-service-pod]
    stagingFmtok8sC4PServiceRs --> stagingFmtok8sC4PServicePod[staging-fmtok8s-c4p-service-pod]
    stagingFmtok8sEmailServiceRs --> stagingFmtok8sEmailServicePod[staging-fmtok8s-email-service-pod]
    stagingFmtok8sFrontendRs --> stagingFmtok8sFrontendPod[staging-fmtok8s-frontend-pod]
  
```

Depending on if you are creating the environment in a local cluster or in a real Kubernetes Cluster, you should access the application and interact with it.

Let's recap a bit on what we have achieved so far:

- We have installed ArgoCD into our Kubernetes Cluster. Using the provided ArgoCD UI we have created a new ArgoCD application for our Staging Environment.
- We have created our Staging Environment configuration in a Git repository hosted in GitHub, which uses a Helm Chart definition to configure our Conference Application services and their dependencies (Redis and PostgreSQL)
- We have synced the configuration to a namespace (`staging`) in the same cluster where we installed ArgoCD.
- Most importantly, we have removed the need for manual interaction against the target cluster. In theory, there will be no need to execute `kubectl` against the `staging` namespace.

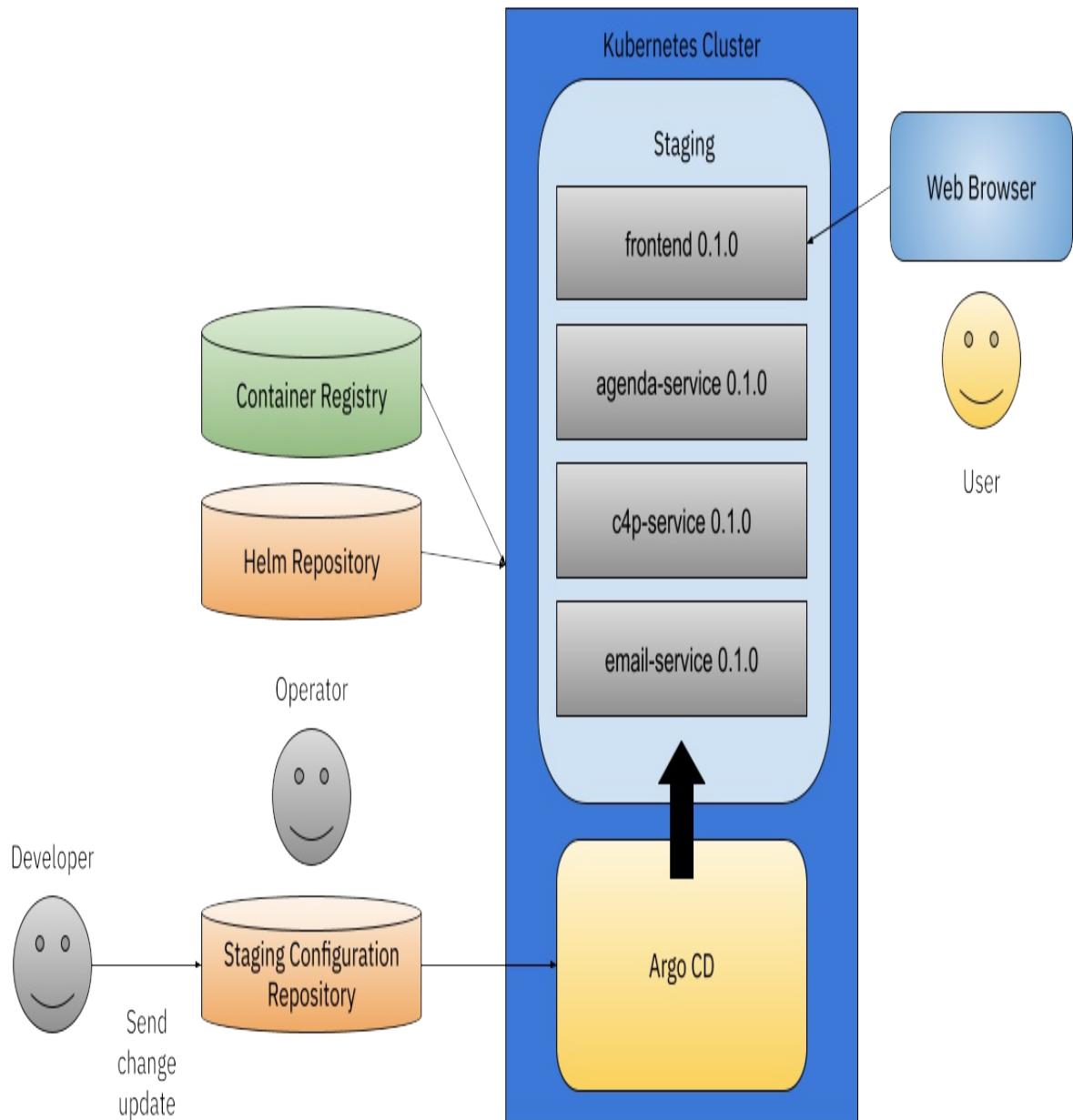
For this setup to work, we need to make sure that the artifacts that the Helm Charts (and the Kubernetes resources inside them) are available for the target cluster to pull.

I strongly recommend you to follow you the step-by-step tutorial to get hands on with tools like ArgoCD and when using them

4.2.3 Dealing with changes, the GitOps way

Imagine now that the team in charge of developing the user interface (`frontend`) decides to introduce a new feature. Hence they create a Pull Request to the `frontend` repository. Once this Pull Request is merged to the `main` the team can decide to create a new release for the service. The release process should include the creation of tagged artifacts using the release number. The creation of these artifacts is the responsibility of the service pipeline, as we saw in previous sections.

Figure 4.X Components to set up the staging environment with ArgoCD



Once we have the released artifacts we can now update the environment. We can update the Staging environment by submitting a Pull Request to our GitHub repository that can be reviewed before merging to the main branch, which is the branch we used to configure our ArgoCD application. The changes in the environment configuration repository are going to be usually about:

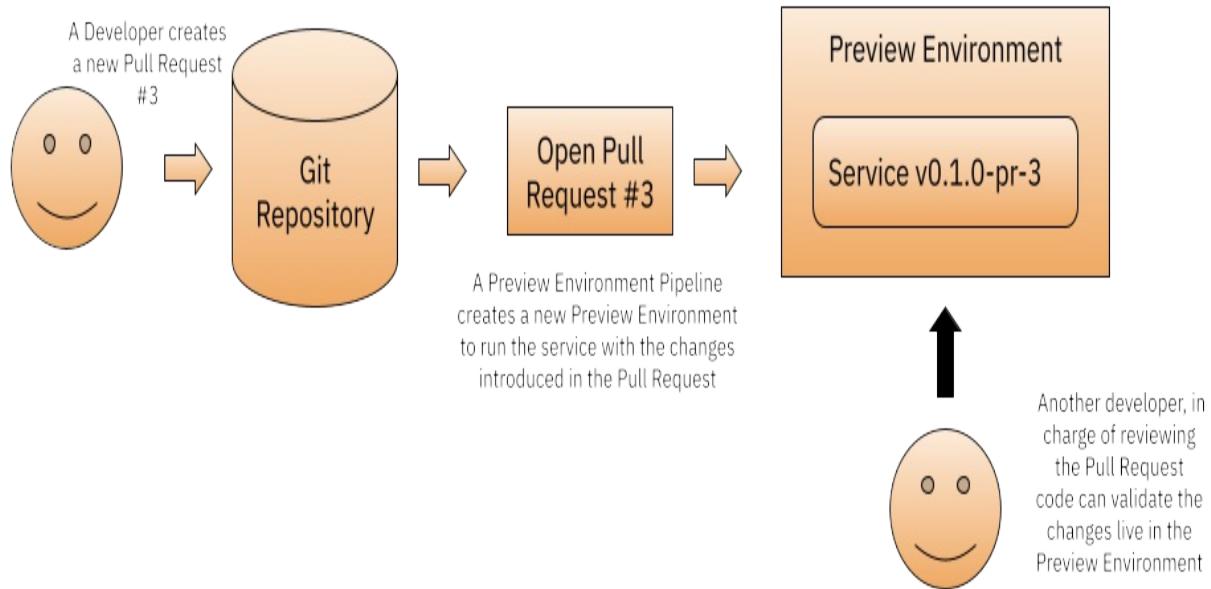
- **Bumping up or reverting a service version:** for our example, this is as simple as changing the version of the chart of one or more services. Rolling back one of the services to the previous is as simple as reverting the version number in the environment chart or even reverting the commit that increased the version in the first place. Notice that reverting commits is always recommended, as rolling back to a previous version might also include configuration changes to the services that, if they are not applied, old versions might not work.
- **Adding or removing a service:** adding a new service is a bit more complicated, as you will need to add both the chart reference and the service configuration parameters. For this to work, the chart definition needs to be reachable by the ArgoCD installation. Suppose the service(s)' chart(s) are available and the configuration parameters are valid. In that case, the next time that we sync our ArgoCD application, the new service(s) will be deployed to the environment. Removing services is more straightforward, as the moment that you remove the dependency from the environment Helm chart the service will be removed from the environment.
- **Tweaking charts parameters:** sometimes, we don't want to change any service version, and we might be trying to finetune the application parameters to accommodate performance or scalability requirements, monitoring configurations, or the log level for a set of services. These kinds of changes are also versioned and should be treated as new features and bug fixes.

If we compare this with manually using Helm to install the application into the cluster, we will quickly notice the differences. First, a developer might have the environment configuration on her/his laptop, making the environment very difficult to replicate from a different location. Changes to the environment configuration that are not tracking using a version control system will be lost, and we will not have any way to verify if these changes are working in a live cluster or not. Configuration drifts are much more difficult to track down and troubleshoot.

Following this automated approach with ArgoCD can open the door to more advanced scenarios. For example, we can create Preview Environments (figure 4.x) for our Pull requests to test changes before they get merged, and

artifacts are released.

Figure 4.X Preview Environments for faster iterations



Using Preview Environments can help to iterate faster and enable teams to validate changes before merging them into the main branch of the project. Preview Environments can also be notified when the Pull Request is merged hence an automated clean-up mechanism is straightforward to implement.

Note:

Another important detail to mention when using ArgoCD and Helm is that compared with using Helm Charts manually, where Helm will create Release resources every time that we update a chart in our cluster, ArgoCD will not use this Helm feature. ArgoCD takes the approach of using `helm template` to render the Kubernetes resources YAML, and then it does apply the output using `kubectl apply`. This approach relies on the fact that everything is versioned in Git and also allows to unify of different templating engines for YAML. At the same time, it also allows ArgoCD to decorate the resulting resources with ArgoCD custom annotations.

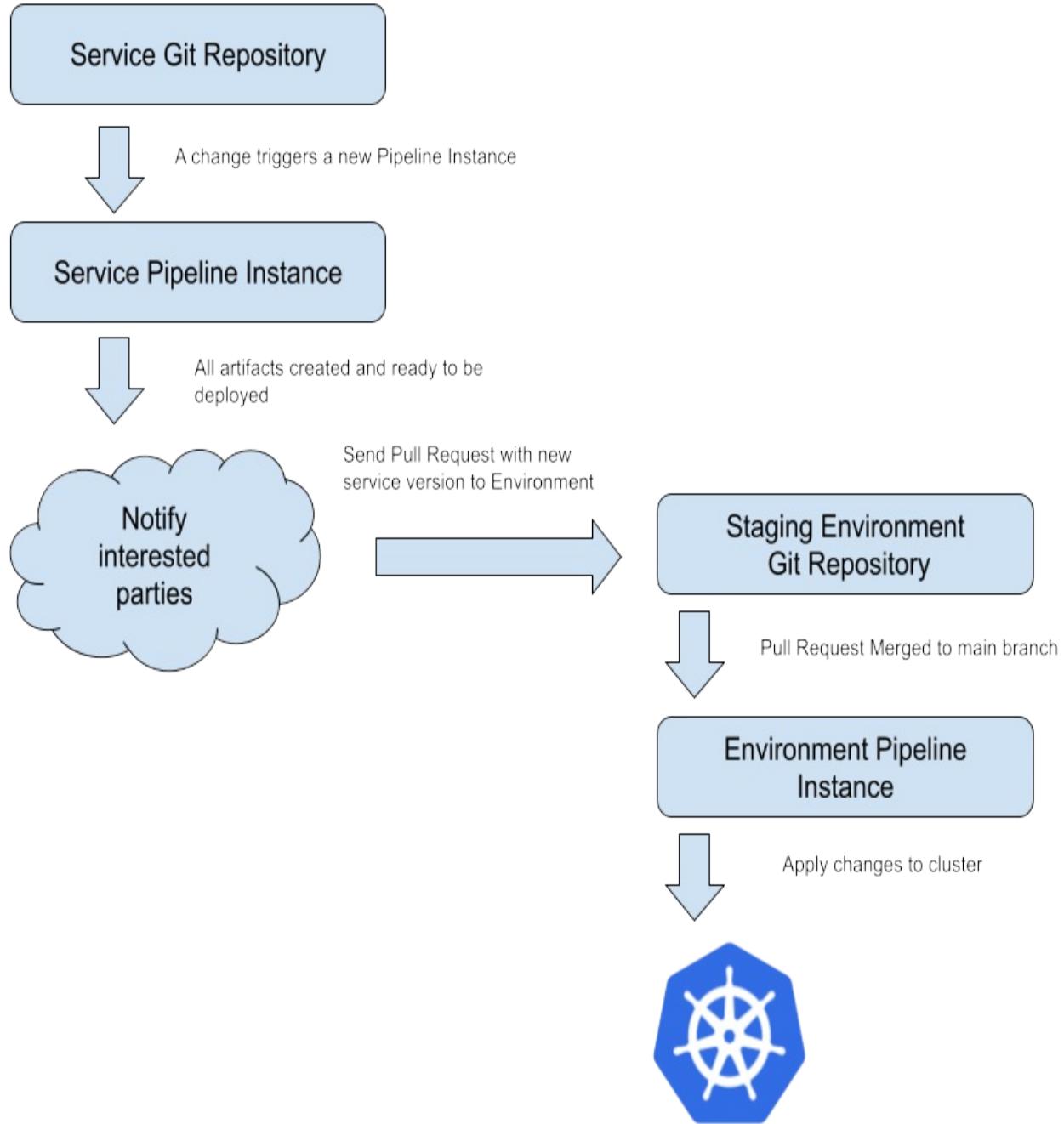
Finally, to tie things together, let's see how Service and Environment

pipelines interact to provide end-to-end automation from code changes to deploying new versions into multiple environments.

4.3 Service + Environment Pipelines

Let's look at how Service Pipelines and Environment Pipeline connect. The connection between these two pipelines happens via Pull/Change Requests to git repositories, as the pipelines will be triggered when changes are submitted and merged:

Figure 4.X A Service Pipeline can trigger an Environment Pipeline via a Pull Request



Developers when they finish a new feature create a Pull/Change request to the repository's `main` branch. This Pull/Change request can be reviewed and built by a specialized Service Pipeline. When this new feature is merged into the repository's `main` branch, a new instance of the Service Pipeline is triggered. This instance creates a new release and all the artifacts needed to deploy the Service's new version into a Kubernetes Cluster. As we saw in chapter 3, this includes a binary with the compiled source code, a container

image, and Kubernetes Manifests that can be packaged using tools like Helm.

As the last step of the Service Pipeline, you can include a notification step that can notify the interested environments that there is a new version of a service that they are running available. This notification is usually an automated Pull/Change request into the Environment's repository.

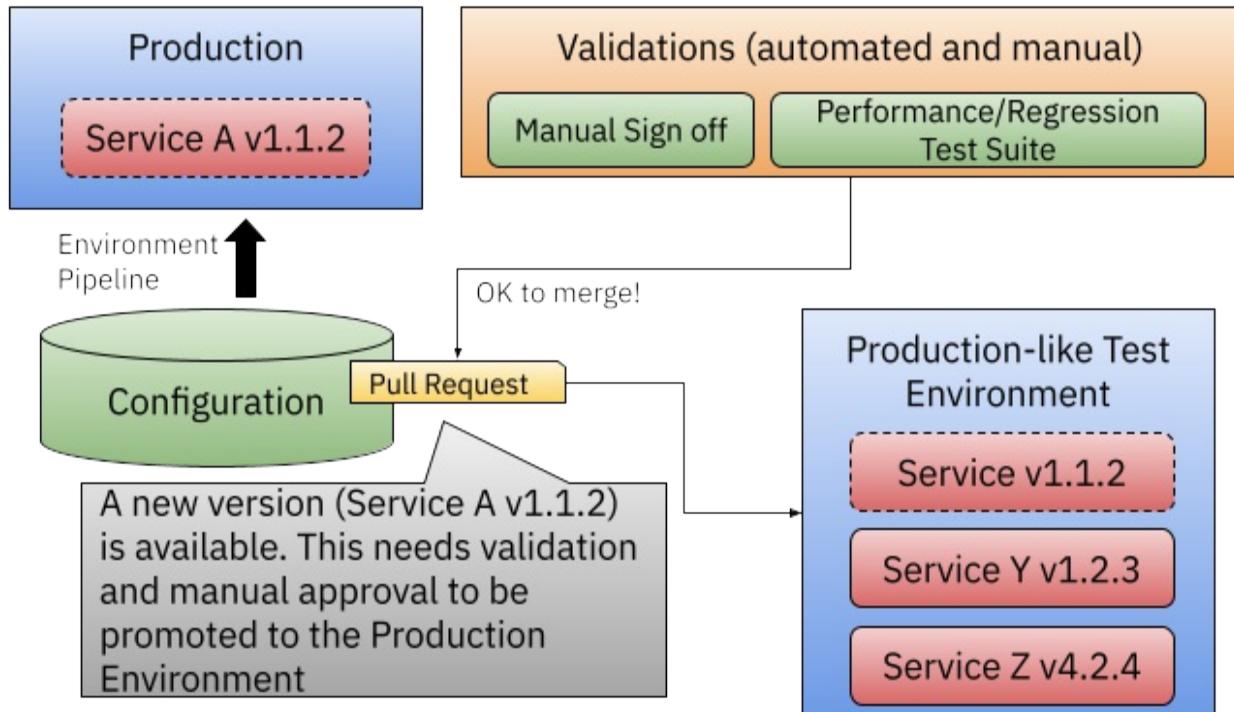
Alternatively, you can be monitoring (or you can be subscribed to notifications) your artifact repositories, and when a new version is detected a Pull/Change request is created to the configured environments.

The Pull/Change requests created to Environment repositories can be automatically tested by a specialized Environment Pipeline, in the same way as we did with Service Pipelines, and for low-risk environments, these Pull/Change requests can be automatically merged without any human intervention.

By implementing this flow we can enable developers to focus on fixing bugs and creating new features that will be automatically released and promoted to low-risk environments.

Once the new versions are tested in environments like Staging, and we know that these new versions or configurations are not causing any issues, a Pull/Change request can be created for the repository that contains the Production environment configuration.

Figure 4.X Promoting changes to the Production Environment



The more sensitive the environments are, the more checks and validations they will require. In this case, as shown in figure 4.x, to promote a new service version to the Production Environment, a new Test environment will be created to validate and test the changes introduced in the Pull/Change request submitted. Once those validations are done, a Manual sign-off is required to merge the Pull Request and trigger the Environment Pipeline synchronization.

At the end of the day, Environment Pipelines are the mechanism you use to encode your organization's requirements to release and promote software to different environments. We have seen in this chapter what a tool like ArgoCD can do for us. Next, we need to evaluate if a single ArgoCD installation would be enough and who will manage it and keep it secure. Do you need to extend ArgoCD with custom hook points? Do you need to integrate it with other tools? These are all valid questions we will explore in Chapter 6, but now it is time to tackle one more important topic: Application Infrastructure.

4.4 Summary

- Environment Pipelines are responsible for deploying software artifacts to live environments. Environment Pipelines avoid teams interacting directly with the cluster where the applications run, reducing errors and misconfigurations.
- Using tools like ArgoCD, you can define the content of each environment into a git repository that is used as the source of truth for what the environment configuration should look like. ArgoCD will keep track of the state of the cluster where the environment is running and make sure that there is no drift in the configuration that is applied in the cluster.
- Teams can upgrade or downgrade the versions of the services running in an environment by submitting Pull/Change requests to the repository where the environment configuration is stored. A team or an automated process can validate these changes, and when approved and merged, these changes will be reflected in the live environment. If things go wrong, changes can be rolled back by reverting commits to the git repository.

5 Multi-cloud (app) infrastructure

This chapter covers

- Define and manage the infrastructure required by your applications
- The challenges of managing your application infrastructure components
- The Kubernetes way to deal with infrastructure with Crossplane

In the previous chapters, we installed a walking skeleton, we understood how to build each separate component using Service Pipelines and how to deploy them into different environments using Environment pipelines. We are now faced with a big challenge: dealing with our application infrastructure, meaning running and maintaining not only our application services but the components that these services need to run. These applications are expected to require other components to work correctly, such as Databases, Message Brokers, Identity Management solutions, Email Servers, etc. While several tools are out there focused on automating the installation or provisioning of these components for On-Premises setups and in different cloud providers, in this chapter, we will focus on just one that does it in a Kubernetes way. This chapter is divided into three main sections:

- The challenges of dealing with infrastructure
- How to deal with infrastructure leveraging Kubernetes constructs
- How to provision infrastructure for our walking skeleton using Crossplane

Let's get started. Why is it so difficult to manage our application infrastructure?

5.1 The challenges of managing Infrastructure in Kubernetes

When you design applications like the walking skeleton introduced in Chapter 1, you face specific challenges that are not core to achieving your

business goals. Installing, configuring and maintaining Application Infrastructure components that support our application's services is a big task that needs to be planned carefully by the right teams with the right expertise.

These components can be classified as Application Infrastructure, which usually involves third-party components not being developed in-house, such as Databases, Message Brokers, Identity Management solutions, etc. A big reason behind the success of modern Cloud Providers is that they are great at providing and maintaining these components and allowing your development teams to focus on building the core features of our applications, which bring value to the business.

It is important to distinguish between Application Infrastructure and Hardware Infrastructure, which is also needed. In general, I assume that for Public Clouds offerings, the provider solves all hardware-related topics. For On-Prem scenarios, you likely have a specialized team taking care of the Hardware (removing, adding and maintaining hardware as needed).

It is common to rely on Cloud Provider services to provision Application Infrastructure. There are a lot of advantages to doing so, such as pay-as-you-use services, easy provisioning at scale, and automated maintenance. But at that point, you heavily rely on provider-specific ways of doing things and their tools. The moment you create a database or a message broker in a cloud provider, you are jumping outside of the realms of Kubernetes. Now you depend on their tools and their automation mechanisms, and you are creating a strong dependency of your business with the specific Cloud Provider.

Let's take a look at the challenges associated with provisioning and maintaining application infrastructure, so your teams can plan and choose the right tool for the job:

- **Configuring components to scale:** each component will require different expertise to be configured (Database Administrators for databases and Message Broker experts) and a deep understanding of how our application's services will use it, as well as the hardware available. These configurations need to be versioned and monitored closely, so new environments can be created quickly to reproduce issues or test new versions of our application.

- **Maintaining components in the long run:** components such as databases and message brokers are constantly released and patched to improve performance and security. This pushes the operations teams to ensure they can upgrade to newer versions and keep all the data safe without bringing down the entire application. This requires a lot of coordination and impact analysis between the teams involved with these components and services.
- **Cloud Provider services affect our multi-cloud strategy:** if we rely on cloud-specific application infrastructure and tools, we need to find a way to enable developers to create and provision their components for developing and testing their services. We need a way to abstract how infrastructure is provisioned to enable applications to define what infrastructure they need without relying directly on cloud-specific tools.

Interestingly, we had these challenges even before having distributed applications, and configuration and provisioning architectural components have always been hard and usually far away from developers. Cloud Providers are doing a fantastic job by bringing these topics closer to developers so they can be more autonomous and iterate faster. Unfortunately, when working with Kubernetes, we have more options that we need to consider carefully to make sure that we understand the trade-offs. The next section covers how we can manage our Application Infrastructure inside Kubernetes; while this is usually not recommended, it can be practical and cheaper for some scenarios.

5.1.1 Managing your Application Infrastructure

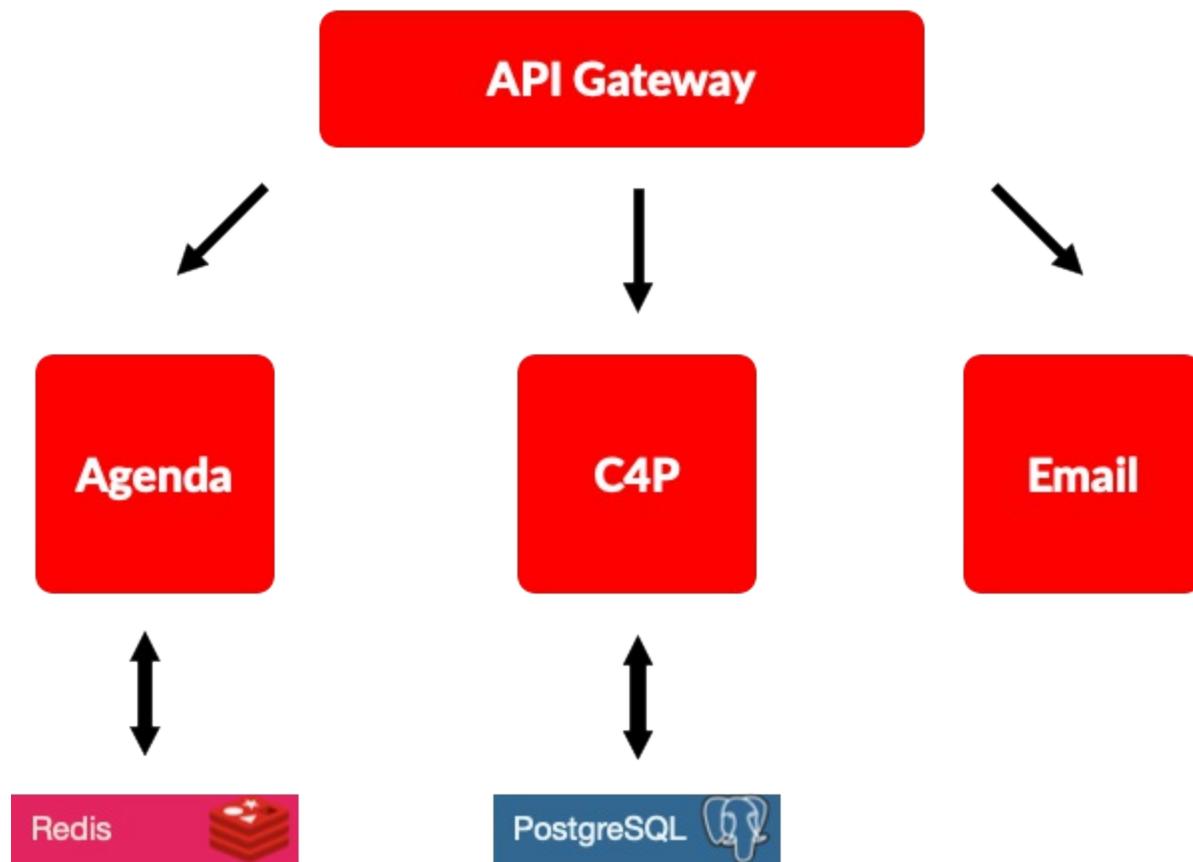
Application infrastructure has become an exciting arena. With the rise of containers, every developer can bootstrap a database or message broker with a couple of commands, which is usually enough for development purposes. In the Kubernetes world, this translates to Helm Charts, which uses containers to configure and provision Databases (relational and NoSQL), message brokers, identity management solutions, etc. As we saw in Chapter 2 you installed the walking skeleton application containing 4 services and 2 databases with a single command.

For our walking skeleton, we are provisioning a Redis NoSQL database for

the Agenda Service and a PostgreSQL database for the Call for Proposals (C4P) Service using Helm Charts. The amount of Helm charts available today is amazing, and it is quite easy to think that installing a Helm Chart will be the way to go.

As discussed in Chapter 2 (Dealing with application state), if we want to scale our services that keep state, we must provision specialized components such as databases. Application developers will define which kind of database will suit them best depending on the data they need to store and how that data will be structured.

Figure 5.x Keeping service state using application infrastructure



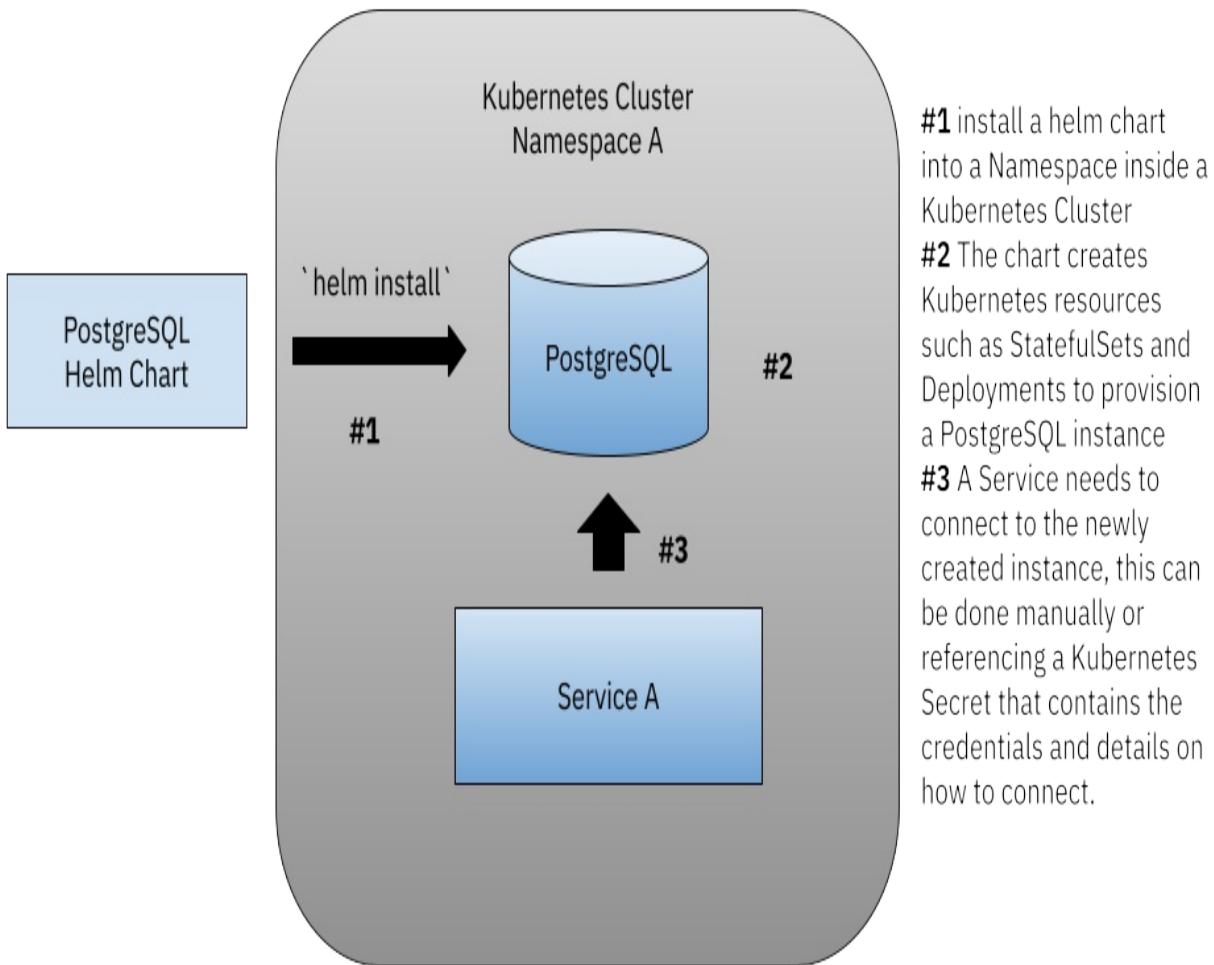
The process of setting up these components inside your Kubernetes Cluster involves the following steps:

- Finding or creating the right Helm Chart for the component you want to bootstrap: for this case, PostgreSQL

(<https://github.com/bitnami/charts/tree/master/bitnami/postgresql>) and Redis (<https://github.com/bitnami/charts/tree/master/bitnami/redis>) can be found in the Bitnami Helm Chart repository. If you cannot find a Helm Chart, but you have a Docker Container for the component that you want to provision, you can create your chart after you define the basic Kubernetes constructs needed for the deployment.

- Research the Chart configurations and parameters you will need to set up to accommodate your requirements. Each chart exposes a set of parameters that can be tuned for different use cases, check the chart website to understand what is available. Here you might want to include your Operations teams and DBAs to check how the Databases need to be configured. This will also require Kubernetes expertise to make sure that the components can work in HA (High Availability) mode inside Kubernetes.
- Install the chart into your Kubernetes Cluster using `helm install`. By running `helm install` you are downloading a set of Kubernetes manifest (YAML files) that describe how your applications/services need to be deployed. Helm will then proceed to apply these YAML files to your cluster. Alternatively, you can define a dependency on your Application's services as we did for the walking skeleton.
- Configure your service to connect to the newly provisioned components, this is done by giving the service the right URL and credentials to connect to the newly created resource, for a database, it will be the database URL serving requests and possibly a username and password. An interesting detail to notice here is that your application will need some kind of driver to connect to the target database.
- Maintain these components in the long run, doing backups and ensuring the fail-over mechanisms work as expected.

Figure 5.x Provisioning a new PostgreSQL instance using the PostgreSQL Helm Chart



If you are working with Helm Charts, there are a couple of caveats and tricks that you need to be aware of:

- If the chart doesn't allow you to configure a parameter that you are interested in changing, you can always use `helm template`, then modify the output to add or change the parameters that you need to finally install the components using `kubectl apply -f`. Alternatively, you can submit a Pull Request to the chart repository. It is a common practice not to expose all possible parameters and wait for community members to suggest more parameters to be exposed by the chart. Don't be shy and contact the maintainers if that is the case. Whatever modification you do, the chart content must be maintained and documented. By using `helm template`, you lose the Helm release management features, which allow you to upgrade a chart when a new chart version is available.

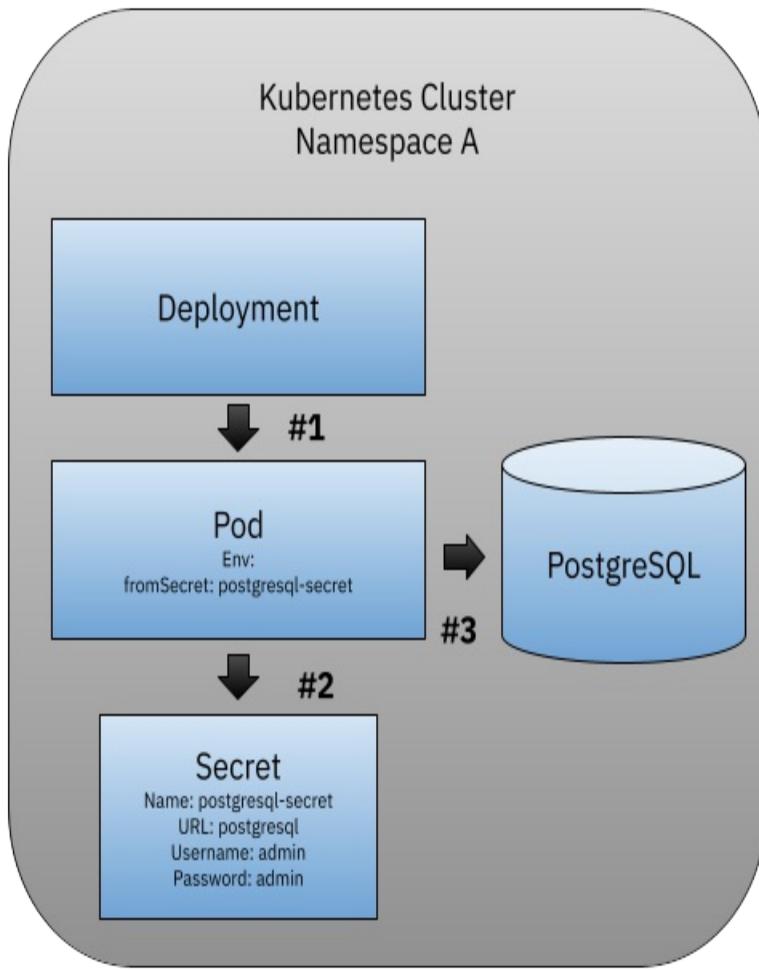
- Most charts come with a default configuration designed to scale, meaning that the default deployment will target high-availability scenarios. This results in Charts that, when installed, consume a lot of resources that might not be available if you use Kubernetes KinD or Minikube on your laptop. Once again, chart documentation usually includes special configurations for development and resource-constrained environments.
- If you are installing a Database inside your Kubernetes Cluster, each Database container (pod) must have access to storage from the underlying Kubernetes Node. For databases, you might need a special kind of storage to enable the database to scale elastically, which might require advanced configurations outside of Kubernetes.

For our walking skeleton, for example, we set up the Redis chart to use the `architecture` parameter to `standalone`, (as you can see in the Environment Pipeline configurations, but also in the Agenda Service Helm Chart `values.yaml` file) to make it easier to run on environments where you might have limited resources, such as your laptop/workstation. This affects Redis's availability to tolerate failure, as it will only run a single replica in contrast with the default setup where a master and two slaves are created.

Connecting our Services to the newly provisioned Infrastructure

Installing the charts will not make our application services automatically connect to the Redis and PostgreSQL instances. We need to provide these services the configurations needed to connect, but also be conscious about the time needed by these components, such as databases, to start.

Figure 5.x Connecting a Service to a provisioned resource using Secrets



A common practice is to use Kubernetes Services to store the credentials for these application infrastructure components. The Helm Chart for Redis and PostgreSQL that we are using for our walking skeleton both create a new Kubernetes Secret containing the details required to connect. These Helm Charts also create a Kubernetes Service to be used as the location (URL) where the instance will be running.

To connect the Call for Proposals (C4P) Service to the PostgreSQL instance, you need to make sure that the Kubernetes Deployment for the C4P service (`conference-fmtok8s-c4p`) has the right environment variables:

- name: SPRING_DATASOURCE_DRIVERCLASSNAME
value: org.postgresql.Driver
- name: SPRING_DATASOURCE_PLATFORM
value: org.hibernate.dialect.PostgreSQLDialect
- name: SPRING_DATASOURCE_URL

```
    value: jdbc:postgresql://${DB_ENDPOINT}:${DB_PORT}/post
- name: SPRING_DATASOURCE_USERNAME
  value: postgres
- name: SPRING_DATASOURCE_PASSWORD
  valueFrom:
    secretKeyRef:
      key: postgres-password
      name: postgresql
- name: DB_ENDPOINT
  value: postgresql
- name: DB_PORT
  value: "5432"
```

In bold are highlighted how we can consume the dynamically generated password when we install the chart and the DB endpoint URL, which in this case is the PostgreSQL Kubernetes Service, also created by the chart. The DB Endpoint would be different if you used a different chart release name.

A similar configuration applies to the Agenda Service and Redis.

```
- name: SPRING_REDIS_HOST
  value: redis-master
- name: SPRING_REDIS_PASSWORD
  valueFrom:
    secretKeyRef:
      key: redis-password
      name: redis
```

As before, we extract the password from a secret called `redis`. The REDIS_HOST is obtained from the name of the Kubernetes Service that is created by the chart, this depends on the `helm release` name that you have used.

Being able to configure different instances for the application infrastructure components opens the door for us to delegate the provisioning and maintenance to other teams and even Cloud Providers.

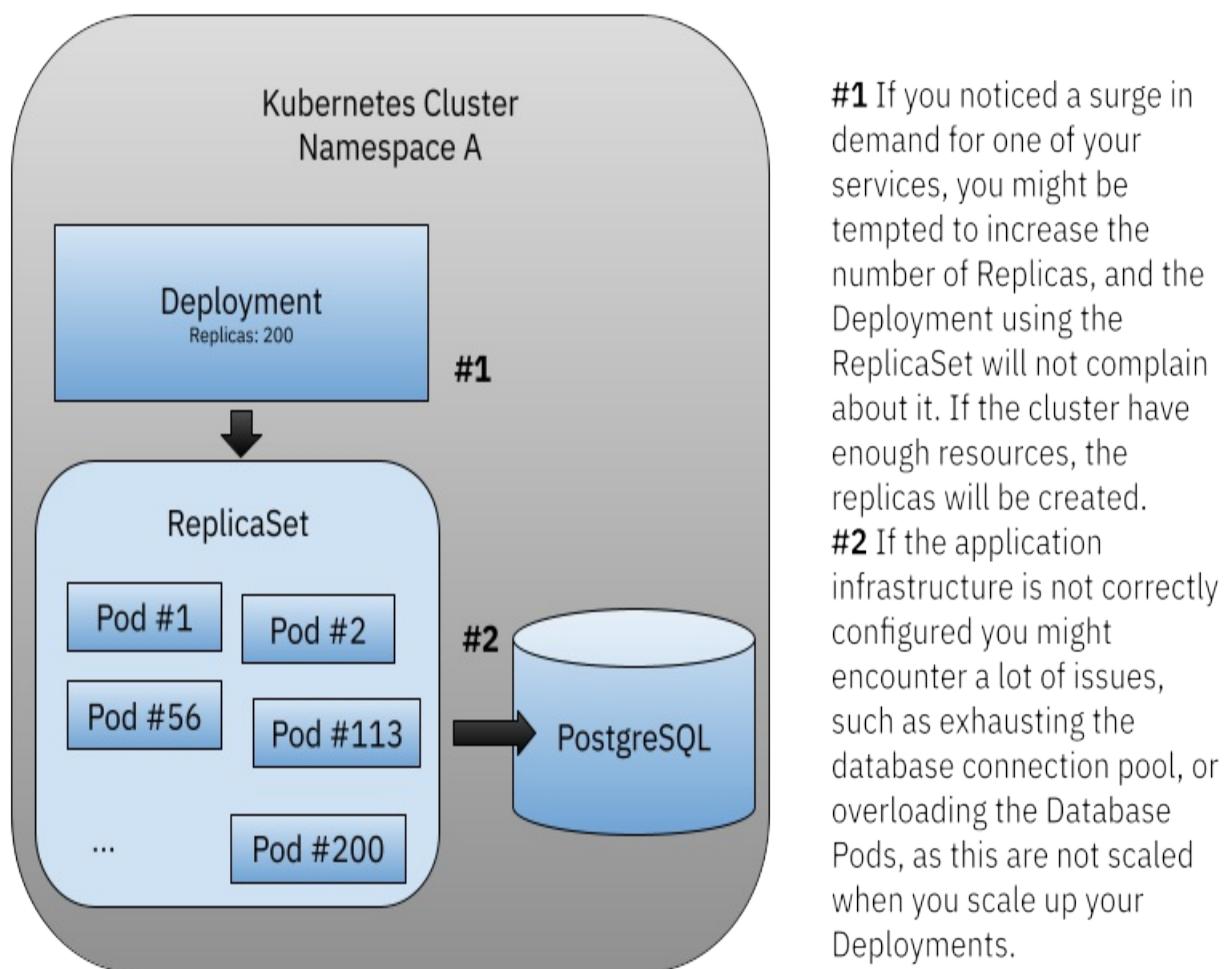
5.1.2 I've heard about Kubernetes Operators. Should I use them?

Now you have 4 application services and two databases inside your Kubernetes cluster. Believe it or not, now you are in charge of 6 components

to maintain and scale depending on the application's needs. The team that built the services will know exactly how to maintain and upgrade each service, but they are not experts in maintaining and scaling databases.

You might need help with these databases depending on how demanding the services are. Imagine if you have too many requests on the Agenda Service, so you decide to scale up the number of replicas of the Agenda Deployment to 200. At that point, Redis must have enough resources to deal with 200 pods connecting to the Redis Cluster. The advantage of using Redis for this scenario, where we might get a lot of reads while the conference is ongoing, is that the Redis Cluster allows us to read data from the replicas so the load can be distributed.

Figure 5.x Application infrastructure needs to be configured according how our services will be scaled



If you are installing Application Infrastructure with Helm, notice that Helm will not check for the health of these components, it is just doing the installation. It is quite common nowadays to find another alternative to install components in a Kubernetes cluster called Operators. Usually associated with Application Infrastructure, you can find more active components that will install and monitor the installed components. One example of these Operators is the Zalando PostgreSQL Operator, which you can find here:

<https://github.com/zalando/postgres-operator>. While these operators are also focused on allowing you to provision new instances of PostgreSQL databases, they also implement other features focused on maintenance, for example:

- Rolling updates on Postgres cluster changes, incl. quick minor version updates
- Live volume resize without pod restarts (AWS EBS, PVC)
- Database connection pooling with PGBouncer
- Support fast in-place major version upgrades.

In general, Kubernetes Operators try to encapsulate the operational tasks associated with a specific component, in this case, PostgreSQL. While using Operators might add more features on top of installing a given component, you still need to maintain the component and the operator itself now. Each Operator comes with a very opinionated flow that your teams will need to research and learn to manage these components. Take this into consideration when researching and deciding which Operator to use.

Regarding the Application Infrastructure that you and your teams decide to use if you plan to run these components inside your cluster, plan accordingly to have the right in-house expertise to manage, maintain and scale these extra components.

In the following section, we will look at how we can tackle these challenges by looking at an Open Source project that aims to simplify the provisioning of Cloud and On-prem resources for application infrastructure components by using a declarative approach.

5.2 Declarative Infrastructure using Crossplane

Using Helm to install application infrastructure components inside Kubernetes is far from ideal for large applications and user-facing environments, as the complexity of maintaining these components and their requirements, such as advanced storage configurations, might become too complex to handle for your teams.

Cloud Providers do a fantastic job at allowing us to provision infrastructure, but they all rely on cloud provider-specific tools which are outside of the realms of Kubernetes.

In this section, we will look at an alternative tool; a CNCF project called Crossplane (<https://crossplane.io>), which uses the Kubernetes APIs and extension points to enable users to provision real infrastructure in a declarative way, using the Kubernetes APIs. Crossplane relies on the Kubernetes APIs to support multiple Cloud Providers, this also means that it integrates nicely with all the existing Kubernetes tooling.

By understanding how Crossplane works and how it can be extended, you can build a multi-cloud approach to build and deploy your Cloud Native applications into different providers without worrying about getting locked in on a single vendor. Because Crossplane uses the same declarative approach as Kubernetes, you will be able to create your high-level abstractions about the applications that you are trying to deploy and maintain.

To use Crossplane, you first need to install its control plane in a Kubernetes Cluster. You can do this by following their official documentation (<https://docs.crossplane.io/>) or the step-by-step tutorial introduced in section 5.3.

The core Crossplane components alone will not do much for you. Depending on your cloud provider(s) of choice, you will install and configure one or more Crossplane Providers. Let's take a look at what Crossplane Providers have to offer us.

5.2.1 Crossplane Providers

Crossplane extends Kubernetes by installing a set of components called “Providers” (<https://docs.crossplane.io/v1.11/concepts/providers/>) in charge

of understanding and interacting with cloud provider-specific services to provision these components for us.

Figure 5.x Crossplane installed with GCP and AWS Providers



By installing Crossplane Providers, you are extending the Kubernetes APIs functionality to provision external resources such as Databases, Message Brokers, Buckets, and other Cloud Resources that will live outside your Kubernetes cluster but inside the Cloud Provider realm. There are several Crossplane providers covering the major cloud providers such as GCP, AWS, and Azure. You can find these Crossplane providers in the Crossplane GitHub's organization: <https://github.com/crossplane/>.

Once a Crossplane Provider is installed, you can create provider-specific resources in a declarative way, which means that you can create a Kubernetes Resource, apply it with `kubectl apply -f`, package these definitions in

Helm Charts or use Environment Pipelines storing these resources in a Git repository.

For example creating a Bucket in Google Cloud using the Crossplane GCP provider looks like this:

```
cat <#A
kind: Bucket #B
metadata:
  generateName: crossplane-bucket-
  labels:
    docs.crossplane.io/example: provider-gcp
spec: #C
  forProvider:
    location: US
  providerConfigRef:
    name: default
EOF
```

Provisioning cloud-specific resources relying on the Kubernetes APIs is a big step forward, but Crossplane doesn't stop there. If you look at what it takes to provision a database in any major cloud provider, you will realize that provisioning the component is just one of the tasks involved in getting the component ready to be used. You will need network and security configurations and user credentials to connect to these provisioned components.

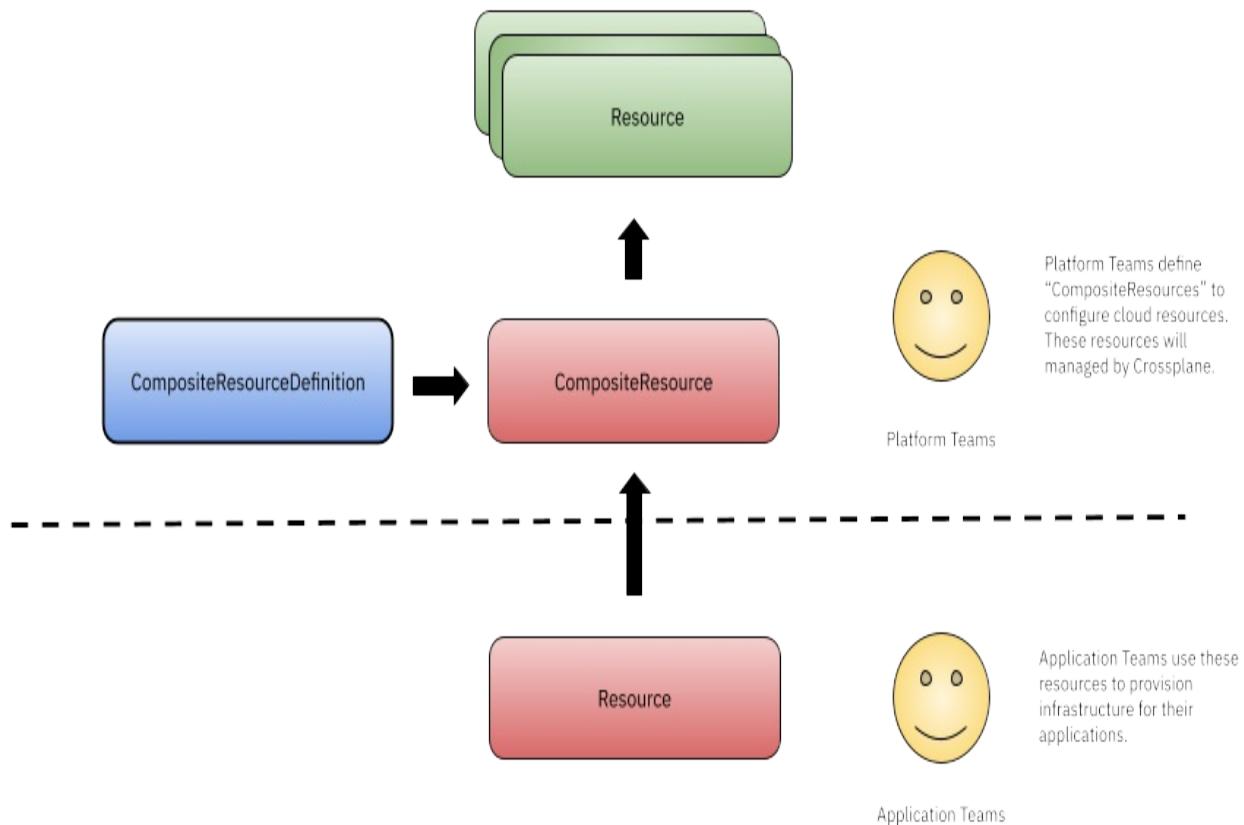
5.2.2 Crossplane Compositions

Crossplane aims to serve two different Personas: “*Platform teams*” and “*Application teams*”. While *Platform* teams are Cloud Providers experts that understand how to provision cloud provider-specific components, *Application* teams know the application requirements and understand what is required from the Application Infrastructure perspective. The interesting thing about this approach is that when using Crossplane, Platform teams can define these complex configurations for a specific Cloud Provider and expose simplified interfaces for Application teams.

In real-life scenarios, it is rare to just create a single component, for example, if we want to provision a Database instance application teams will also

require the correct network and security configurations to be able to access the newly created instance. Being able to compose and wire up together several components is a very convenient feature and for achieving these abstractions and simplified interfaces Crossplane introduced two concepts “*Composite Resource Definitions (XRDs)*” and “*Composite Resources(XR)*”.

Figure 5.x Resource Composition abstraction by Crossplane Composite Resources



The previous diagram shows how you can use Crossplane *Composite Resources (XRD)* to define abstractions for different Cloud Providers. The *Platform team* might be very knowledgeable in Google Cloud or Azure, so they will be in charge of defining which Resources they want to wire up together for a specific application. The *Application team* has a simple Resource interface to request the Resource they are interested in. But as usual, abstractions are complicated and good to show who is responsible for what, but let's look at a concrete example to understand the power of Crossplane Compositions.

Figure 5.x Provisioning a PostgreSQL instance in Google Cloud with Crossplane Compositions

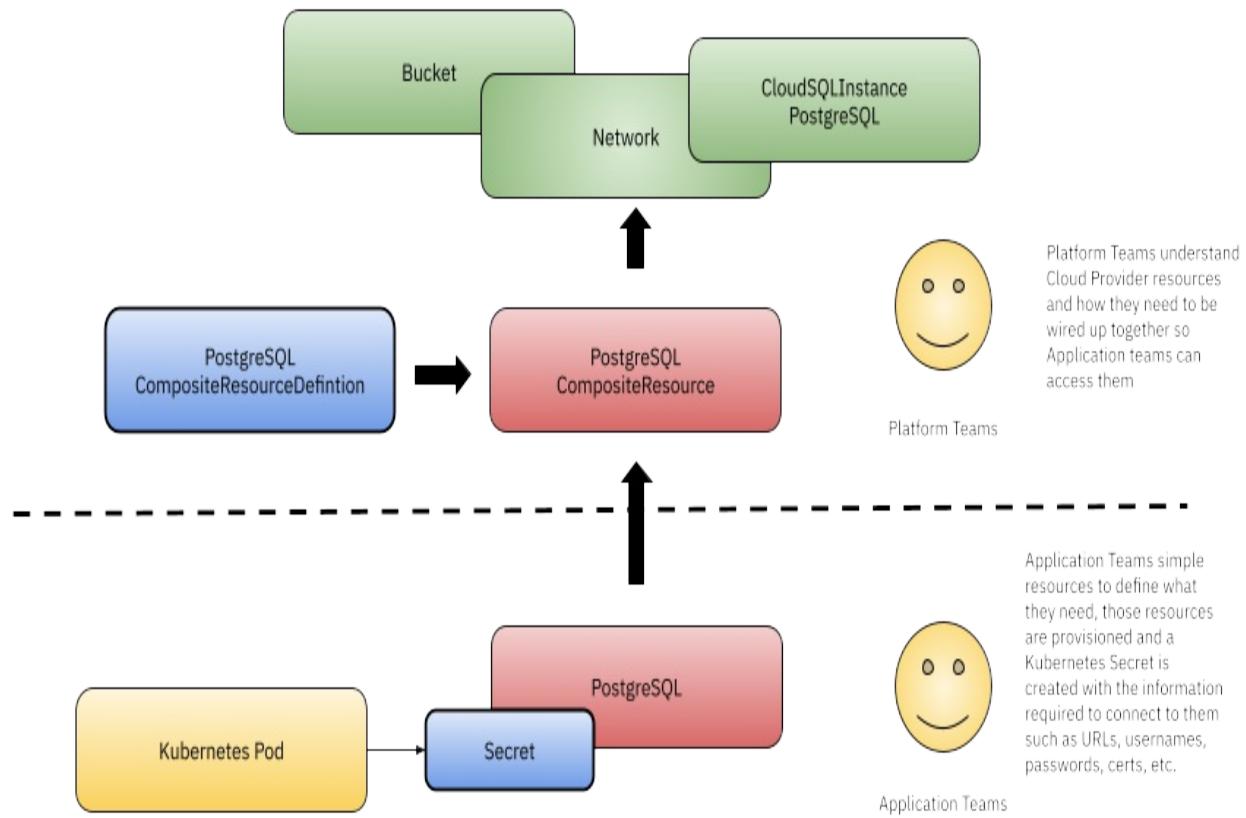


Figure 5.x shows how the application team can create a simple PostgreSQL resource to provision in Google Cloud a CloudSQLInstance plus a Network configuration and a Bucket. The application team is interested in something other than what resources are created or even in which Cloud Provider they were created. They are only interested in having a PostgreSQL instance to connect their applications to.

This takes us to the “Secret” box in the figure, representing a Kubernetes Secret that Crossplane will create for our application/services Pods to connect to the provisioned resources. Crossplane creates this Kubernetes Secret with all the details our applications require to connect to the newly created resources (or just with the one relevant to the application). This Secret typically contains URLs, usernames, passwords, certificates, or anything required for your applications to connect. Platform teams define what will be included in the secret when defining the CompositeResources. In the following sections, when we add real infrastructure to our Conference

application, we will explore how these `CompositeResourceDefinitions` look and how they can be applied to create all the components that our applications need.

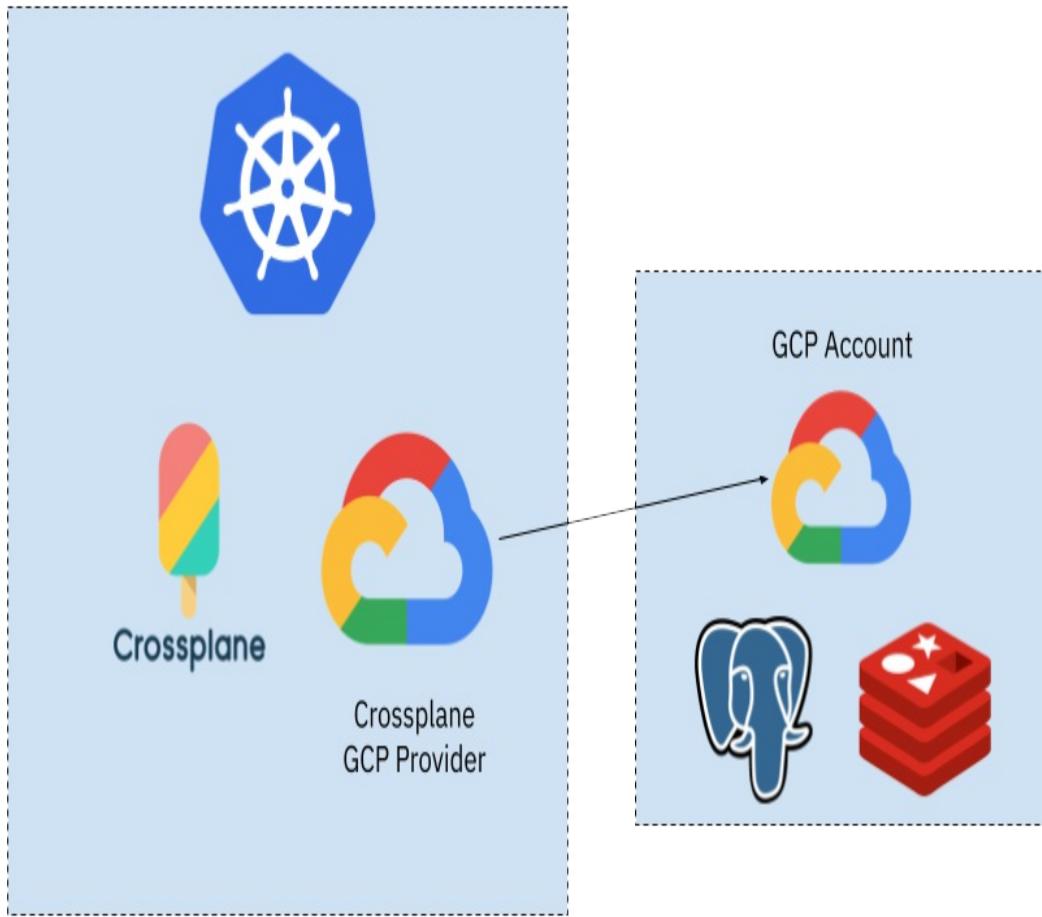
5.2.3 Crossplane Components and Requirements

To work with Crossplane Providers and `CompositeResourceDefinitions` we need to understand how Crossplane components will work together to provision and manage these components inside different Cloud Providers.

This section covers what Crossplane needs to work and how Crossplane components will manage our `CompositeResources`.

First, it is important to understand that you must install Crossplane in a Kubernetes cluster. This can be the Cluster where your applications run or a separate cluster where Crossplane will run. This cluster will have some Crossplane components that will understand our `CompositeResourceDefinitions` and have enough permissions on the cloud platform to provision resources on our behalf.

Figure 5.x Crossplane in Google Cloud Platform



The previous figure shows Crossplane installed inside a Kubernetes Cluster, with the Crossplane GCP provider installed and configured to use a Google Cloud Platform account with enough rights to provision PostgreSQL and Redis instances. This means having, in some cases, admin access to create resources on the cloud provider.

For Figure 5.x to work in GCP, you need the following configurations on the Cloud Provider:

- For creating a Redis instance in GCP
 - Your GCP project need to have the “redis.googleapis.com” APIs enabled
 - You also need to have admin rights on the redis resources `roles/redis.admin`
- For creating a PostgreSQL instance in GCP:
 - Your GCP project need to have the “sqladmin.googleapis.com” APIs enabled
 - You also need to have admin rights on the SQL resources `roles/cloudsql.admin`

Each Crossplane Provider available requires a specific security configuration to work and an account inside the Cloud Provider where we want to create resources.

Once a Crossplane Provider is installed and configured, in this case, the GCP provider, we can start creating resources managed by this provider. You can find the resources offered by each provider on the following documentation site: <https://doc.crds.dev/github.com/crossplane/provider-gcp>.

Figure 5.x Crossplane GCP supported resources



crossplane/provider-gcp@v0.22.0

github.com/crossplane/provider-gcp/tree/v0.22.0

v0.22.0 ▾

CRDs discovered: 29

e.g. CacheCluster, acm.aws

Kind	Group	Version
CloudMemorystoreInstance	cache.gcp.crossplane.io	v1beta1
Firewall	compute.gcp.crossplane.io	v1alpha1
Router	compute.gcp.crossplane.io	v1alpha1
Address	compute.gcp.crossplane.io	v1beta1
GlobalAddress	compute.gcp.crossplane.io	v1beta1
Network	compute.gcp.crossplane.io	v1beta1
Subnetwork	compute.gcp.crossplane.io	v1beta1
NodePool	container.gcp.crossplane.io	v1beta1

As you can see in the previous figure, the GCP Provider version 0.22.0 supports 29 different CRDs (Custom Resource Definitions) for creating resources in the Google Cloud Platform. Crossplane defines each of these resources as Managed Resources. Each of these managed resources will need to be enabled for the Crossplane Provider to have the right access to the list, create and modify these resources.

In section 5.3, we will be looking at how to provision Cloud or Local resources for our applications by using different Crossplane Providers and Crossplane Compositions. Before jumping into the technical aspects, let's take a look at Crossplane core behaviors that you should look for when working with tools in the Kubernetes space.

5.2.4 Crossplane Behaviors

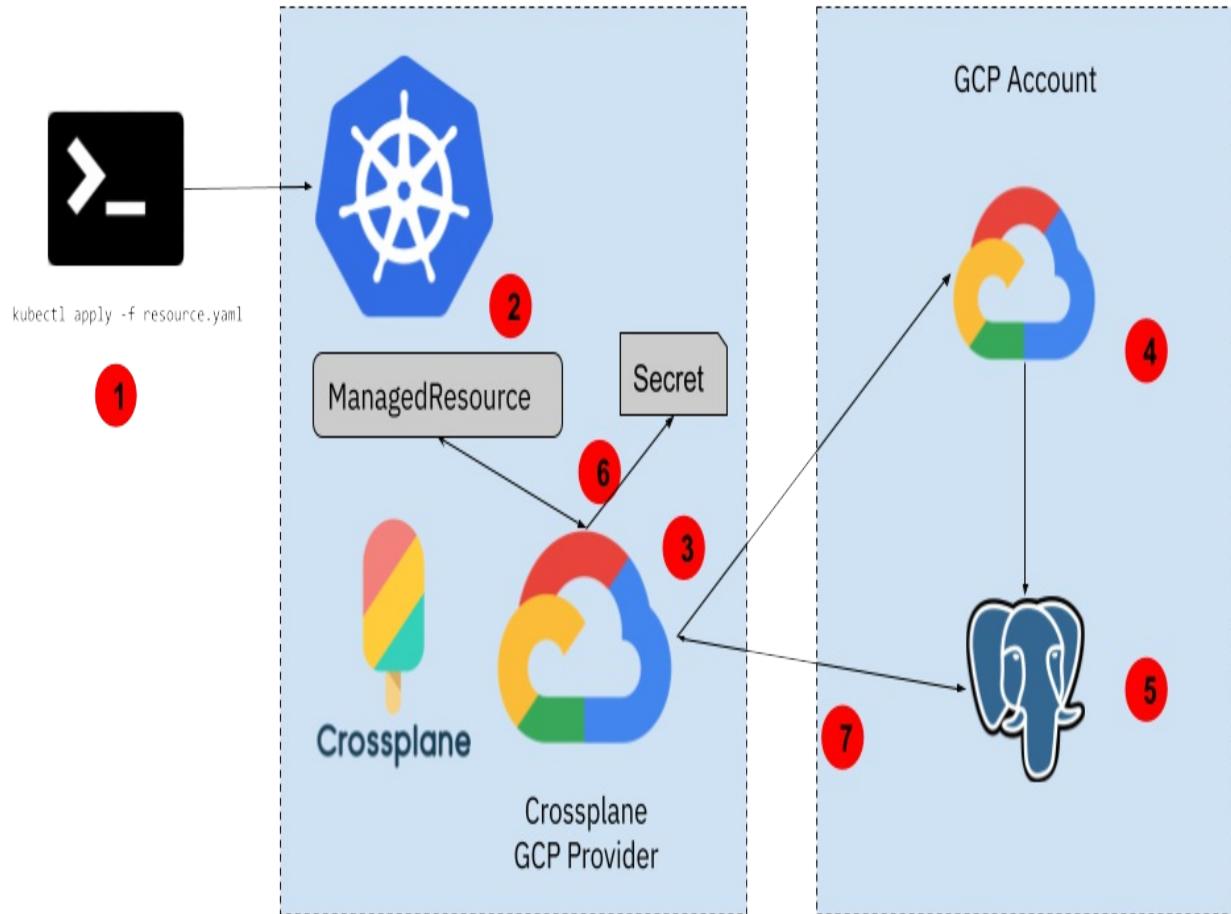
In contrast to installing Helm components in our Kubernetes Clusters, we use Crossplane to interact with the cloud provider-specific APIs to provision resources inside the Cloud infrastructure. This should simplify the maintenance tasks and costs related to these resources. Another important difference is that the Crossplane provider (GCP provider in this case) will monitor the created Managed Resources for us. These Managed Resources offer some advantages compared with just installed resources using Helm. Managed Resources have very well-defined behaviors. Here is a summary of what to expect from a Crossplane Managed Resource:

- **Visible as any other Kubernetes Resource:** Crossplane Managed Resources are just Kubernetes resources, this means that we can use any Kubernetes tool to monitor and query the state of these resources.
- **Continuous Reconciliation:** when a managed resource is created, the Provider will continuously monitor the resource to make sure that it exists and is working and report back the status to the Kubernetes resource. The parameters defined inside the managed resource are considered the desired state (source of truth) and providers will work to apply these configurations to the Cloud Provider resources. Once again, we can use standard Kubernetes tools to monitor changes in state and trigger remediation flows.

- **Immutable Properties:** providers are in charge of reporting back if a user manually changes properties in the cloud provider. The idea here is to avoid configuration drifts from what was defined to what is running in the cloud provider. If so, the state is reported back to the managed resource. Crossplane will not delete the cloud provider resource but will notify back so actions can be taken. Other tools like Terraform (<https://www.terraform.io>) will automatically delete the remote resources to recreate them.
- **Late initialization:** some properties in the Managed Resources can be optional, meaning each provider will select the default values for these properties. When this happens, Crossplane creates the resource with the default values and then sets the selected values into the Managed Resource. This simplifies the configuration needed to create resources and reuse the sensible defaults defined by cloud providers, usually in their user interfaces.
- **Deletion:** when deleting a Managed Resource, the cloud provider immediately triggers the action. However, the Managed Resource is kept until the resource is fully removed from the Cloud Provider. Errors that might happen during deletion on the cloud provider will be added to the Managed resource status field.
- **Importing existing resources:** Crossplane doesn't necessarily need to create the resources to be able to manage them. You can create Managed Resources that start monitoring components created before Crossplane was installed. You can achieve this using a specific Crossplane annotation on the Managed Resource: `crossplane.io/external-name`.

To summarize the interactions between Crossplane, the Crossplane GCP Provider, and our Managed Resources, let's look at the following diagram:

Figure 5.x Lifecycle of Managed Resources with Crossplane



The following points indicate the sequence observed in Figure 5.x:

1. First, we need to create a resource. We can use any tool to create Kubernetes resources, `kubectl` here is just an example.
2. If the resource we created is a Crossplane Managed Resource, let's imagine a CloudSQLInstance resource. The specific Crossplane Provider will pick it up and manage it.
3. The first step to execute when managing a resource will be checking if it exists in the infrastructure (that is, in the configured GCP account). If it doesn't exist, the Provider will request that the resource be created in the infrastructure. The appropriate SQL database will be provisioned depending on the properties set on the resource, such as which kind of SQL database is required. Imagine that we have chosen a PostgreSQL database for the sake of the example.
4. The Cloud Provider, after receiving the request, if the resources are enabled, will create a new PostgreSQL instance with the configured

parameters in the Managed Resource.

5. The status of the PostgreSQL will be reported back to the Managed Resource, which means that we can use `kubectl` or any other tool to monitor the status of the provisioned resources. Crossplane providers will keep these in sync.
6. When the database is up and running, the Crossplane Provider will create a secret to store the credentials and properties that our applications will need to connect to the newly created instance
7. Crossplane will regularly check the status of the PostgreSQL instance and update the managed resource.

By following Kubernetes design patterns, Crossplane leverages the reconciliation cycle implemented by controllers to keep track of external resources. Let's see this in action! The following section will examine how we can use Crossplane with our walking skeleton application.

5.3 Infrastructure for our walking skeleton

In this section, we will use Crossplane to abstract away how we provision infrastructure for our Conference application. Because you might not have access to a Cloud Provider like GCP, AWS or Azure we will be working with a special provider called the Crossplane Helm provider. This Crossplane Helm provider allows us to manage Helm Charts as Cloud Resources. The idea here is to show how with Crossplane, and more specifically using Crossplane Compositions, we can enable users to request resources using a simplified Kubernetes resource to provision local or different cloud resources (hosted in different cloud providers).

For our Conference application, we need a Redis and a PostgreSQL database. From the application perspective, as soon as these two components are available and we can connect to them, all the rest are infrastructural details.

The Agenda Service Helm Chart includes the Redis chart dependency:

```
apiVersion: v2
description: FMT0K8s Agenda Service Helm chart for Kubernetes
name: fmtok8s-agenda-service
version: 0.1.0
```

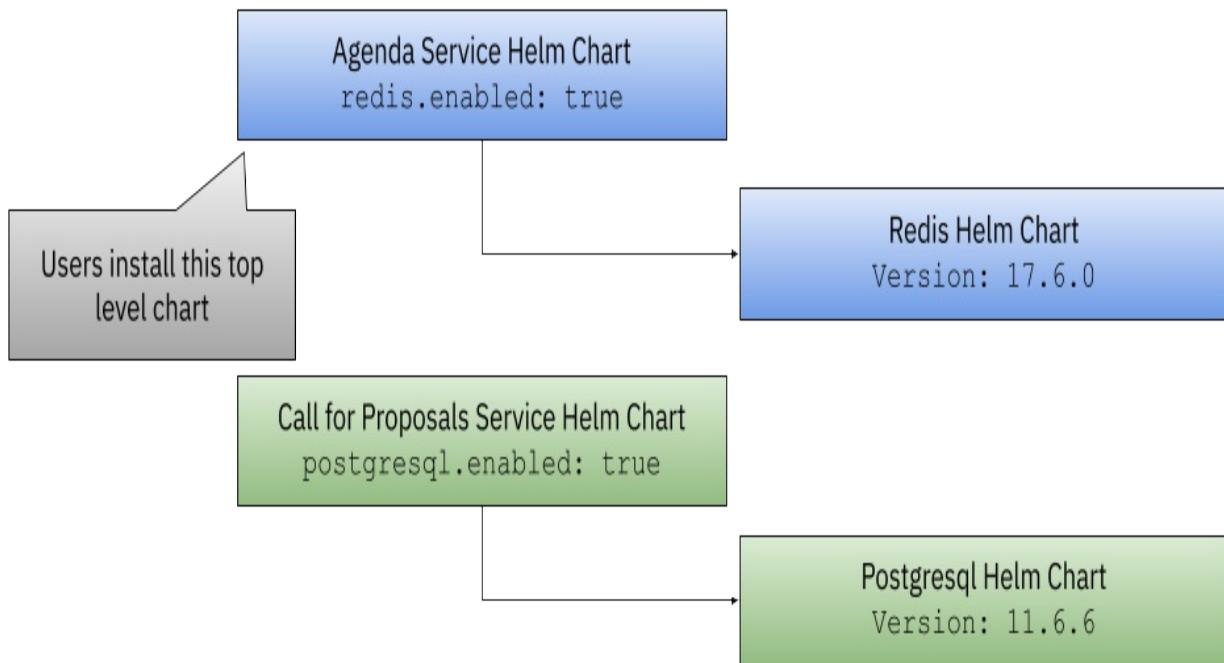
```

dependencies:
- name: redis #A
  version: 17.6.0
  repository: https://charts.bitnami.com/bitnami #B
  condition: redis.enabled #C

```

In “Call for Proposals” service Helm chart, the exact mechanism is used to add the dependency on the PostgreSQL Helm chart. This kind of Chart dependency works for development teams that want to install the service and the database with a single command. Still, we want to decouple all the application infrastructural concerns from application services for larger scenarios. Luckily, the services Helm charts allow us to turn off these component dependencies, allowing us to plug us Redis and PostgreSQL instances hosted and managed by different teams.

Figure 5.x Using Helm Chart dependencies for Application Infrastructure



By separating who installs and who provisions these application’s infrastructure components we enable different teams to control and manage when these components are updated, backed up, or how they need to be restored in case of failure.

By using Crossplane, we can enable teams to request these databases on demand, that then can be connected to our application's services. One important aspect of the mechanisms we will be using in the next sections is that the components we will be requesting can be provisioned locally (using the Crossplane Helm Provider) or remotely using Crossplane Cloud Providers. Let's take a look at what this would look like.

You can follow a step-by-step tutorial to install, configure and create your Crossplane compositions: <https://github.com/salaboy/from-monolith-to-k8s/tree/main/crossplane>

In this example, we will be creating a KinD cluster and configuring Crossplane to allow teams to request databases on demand. More specifically, Redis and PostgreSQL instances. We want to do this in a way that teams can select where these instances are going to be provisioned, but we want to offer a unified experience for all the available options.

After having Crossplane and the Crossplane Helm Provider installed, we need to define two main things:

- **Crossplane Composite Resource Definition:** defines the resources that we want to expose to our teams. For this example, it is called Database. This Composite Resource Definition defines an interface that multiple Compositions can implement.
- **Crossplane Composition:** the Crossplane composition allows us to group a set of resources. We can link a composition to a Composite Resource Definition. By doing so, when the user requests new resources from the Composite defined resource, all the composed resources will be created. I know it sounds confusing, so let's see it in action.

Let's look at the Database Crossplane Composite Resource Definition:

```
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
  name: databases.salaboy.com #A
spec:
  group: salaboy.com #B
  names:
    kind: Database #C
```

```

plural: databases
shortNames:
  - "db"
  - " dbs"
versions:
- additionalPrinterColumns:
  - jsonPath: .spec.parameters.size
    name: SIZE
    type: string
  - jsonPath: .spec.parameters.mockData
    name: MOCKDATA
    type: boolean
  - jsonPath: .spec.compositionSelector.matchLabels.kind
    name: KIND
    type: string
name: v1alpha1
served: true
referenceable: true
schema:
  openAPIV3Schema:
    type: object
    properties:
      spec:
        type: object
        properties:
          parameters: #D
            type: object
            properties:
              size:
                type: string
              mockData:
                type: boolean
            required:
              - size
        required: #E
          - parameters

```

We have defined a new type of resource called a Database which contains two parameters that we can set, `size` and `mockData`. By setting up the `size` parameter, users can define how many resources are allocated for that instance. Instead of worrying about the specifics of how much storage they will need or how many replicas they need for the database instances, they can simply specify a size from a list of possible values (small, medium or large). Using the `mockData` parameters, you can implement a mechanism to inject data into the instance when needed. This is just an example of what can be

done, but it is up to you to define these interfaces and what parameters make sense to your teams.

Let's take a look at what the Crossplane Composition looks like:

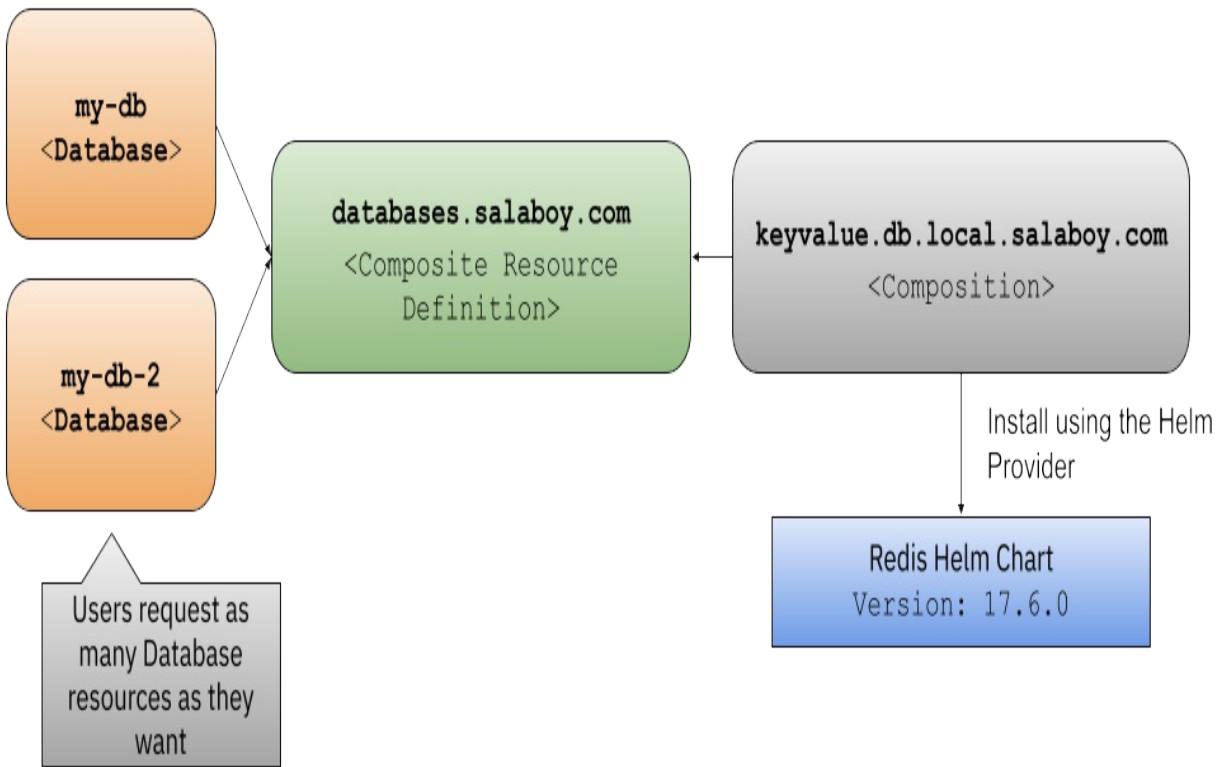
```
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: keyvalue.db.local.salaboy.com #A
  labels: #B
    type: dev
    provider: local
    kind: keyvalue
spec:
  writeConnectionSecretsToNamespace: crossplane-system
  compositeTypeRef: #C
    apiVersion: salaboy.com/v1alpha1
    kind: Database
  resources:
    - name: redis-helm-release #D
      base:
        apiVersion: helm.crossplane.io/v1beta1
        kind: Release
        metadata:
          annotations:
            crossplane.io/external-name: # patched
      spec:
        rollbackLimit: 3
        forProvider:
          namespace: default
          chart: #E
            name: redis
            repository: https://charts.bitnami.com/bitnami
            version: "17.8.0"
          values:
            architecture: standalone
        providerConfigRef: #F
          name: default
      patches: #G
        - fromFieldPath: metadata.name
          toFieldPath: metadata.annotations[crossplane.io/externa
          policy:
            fromFieldPath: Required
        - fromFieldPath: metadata.name
          toFieldPath: metadata.name
          transforms:
```

```

    - type: string
      string:
        fmt: "%s-redis"
      readinessChecks: #H
    - type: MatchString
      fieldPath: status.atProvider.state
      matchString: deployed
  
```

With this composition, we link our Database resources with a set of resources, in this case, installing the Redis Helm Chart using the Default Helm Provider we installed with Crossplane in our Kubernetes cluster.

Figure 5.x Crossplane Composition and Composite Resource Definition working together



It is important to notice that this Helm Chart will be installed in the same Kubernetes Cluster where Crossplane is installed. Still, nothing stops us from configuring the Helm Provider to have the right credentials to install charts in a completely different cluster—more on this in Chapter 6.

In the step-by-step tutorial (<https://github.com/salaboy/from-monolith-to-k8s/tree/main/crossplane>), you will install the Composite Resource Definition

and the Composition. Once these resources are installed, as shown in Figure 5.x above, you will be able to request new Database resources, and for every resource all the resources defined in the Composition will be provisioned. For the sake of simplicity, this Composition just installs Redis, but there are no limits on how many resources you can create (except for the available hardware or quotas that you have).

A Database resource is just another Kubernetes resource that now our cluster understands, and it looks like this:

```
apiVersion: salaboy.com/v1alpha1
kind: Database
metadata:
  name: my-db-keyvalue #A
spec:
  compositionSelector:
    matchLabels: #B
      provider: local
      type: dev
      kind: keyvalue
  parameters: #C
    size: small
  mockData: false
```

Notice that the `spec.compositionSelector.matchLabels` matches with the labels used for the Composition. We can use this mechanism to select a different Composition for the same Database definition.

If you are following the step-by-step tutorial, try to create multiple resources and look at the Crossplane official documentation to understand how to implement parameters like `small` or `mockData` as these values are not being used and only serve for demonstration purposes.

The real power of these mechanisms comes when you have different Compositions (implementations) for the same interface (Composite Resource Definition). For example, we can now create another composition to provision PostgreSQL instances for the Call for Proposals service. The PostgreSQL composition will look the same as the one for Redis, but it will of course, install the PostgreSQL helm chart instead.

```

apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: sql.db.local.salaboy.com #A
  labels:
    type: dev
    provider: local
    kind: sql #B
spec:
  ...
  compositeTypeRef:
    apiVersion: salaboy.com/v1alpha1
    kind: Database
  resources:
    - name: postgresql-helm-release
      base:
        apiVersion: helm.crossplane.io/v1beta1
        kind: Release
        spec:
          forProvider:
            chart: #C
            name: postgresql
            repository: https://charts.bitnami.com/bitnami
            version: "12.2.7"
          providerConfigRef:
            name: default
            ...

```

Let's look at how to create a PostgreSQL instance using this composition. Creating a PostgreSQL instance will look pretty similar to what we did before for Redis:

```

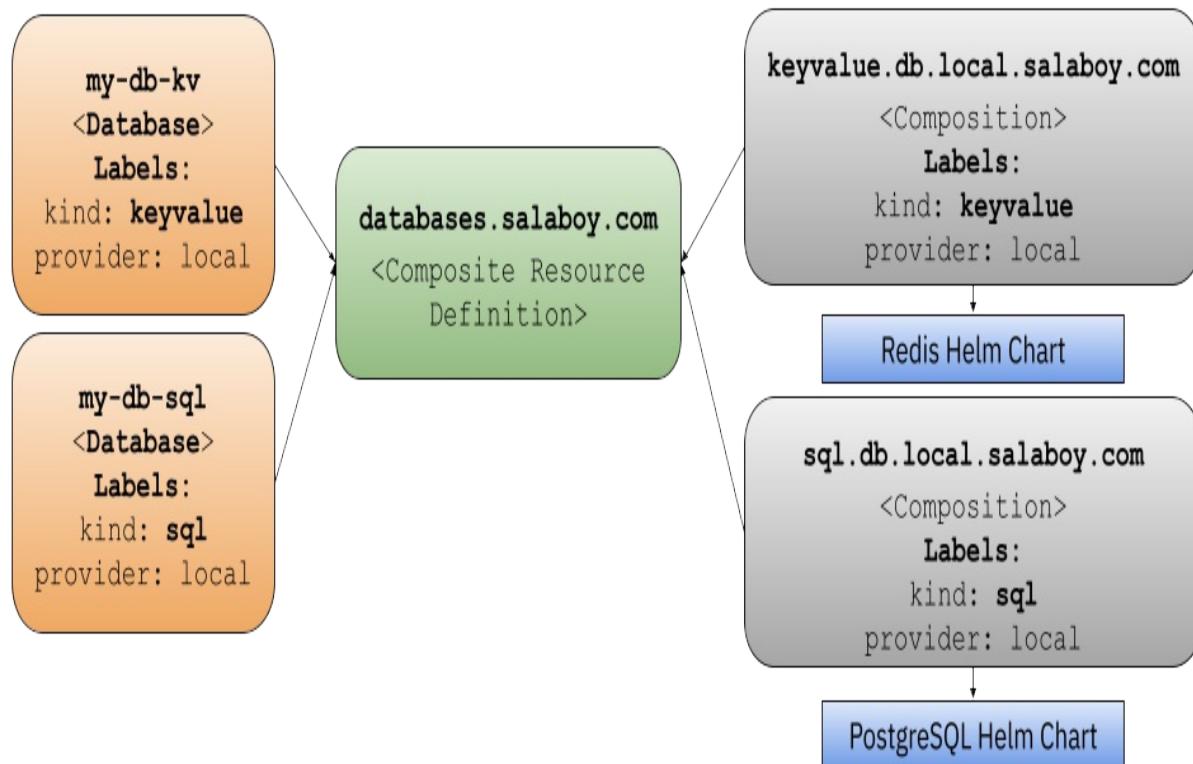
apiVersion: salaboy.com/v1alpha1
kind: Database
metadata:
  name: my-db-sql #A
spec:
  compositionSelector:
    matchLabels:
      provider: local
      type: dev
      kind: sql #B
  parameters:
    size: small
    mockData: false

```

We are just using labels to select which composition will be triggered for our

Database resource. Figure 5.x shows these concepts in action. Notice how labels are used to select the right composition based on the `kind` label value.

Figure 5.x Selecting compositions using labels



Hooray! We can create Databases! But of course, this doesn't stop here. If you have access to a Cloud Provider, you can provide compositions that create Database instances inside the Cloud Provider, and this is where Crossplane shines.

In the step-by-step tutorial directory, you will find under the `databases/` directory two more compositions targeting the Google Cloud Platform services `CloudMemorystoreInstance` for Redis and `CloudSQLInstance` for PostgreSQL. Same as before, we will use different labels to match these compositions.

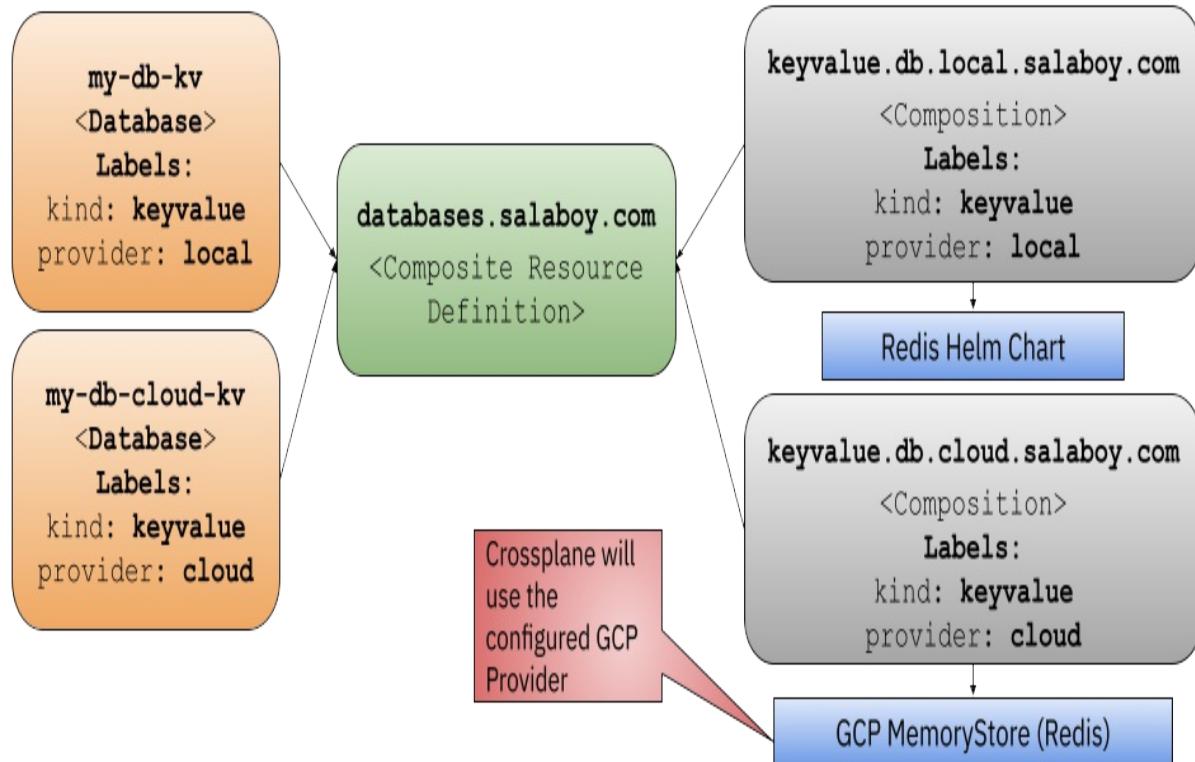
You can find the composition for Redis here (<https://github.com/salaboy/from-monolith-to-k8s/blob/main/crossplane/databases/app-database-redis-gke.yaml>) and the composition for PostgreSQL here (<https://github.com/salaboy/from-monolith-to-k8s/blob/main/crossplane/databases/app-database-postgresql-gke.yaml>).

[to-k8s/blob/main/crossplane/databases/app-database-postgresql-gke.yaml](#)).

You will notice that these compositions are simple and just create one resource each.

To get these compositions working, you will need to install the Crossplane GCP (Google Cloud Platform) Provider and configure it accordingly, as explained here <https://github.com/salaboy/from-monolith-to-k8s/blob/main/crossplane/prerequisites.md#working-with-cloud-providers-example-gcp>

Figure 5.x Selecting compositions using different providers, still using labels



We can still select different providers by matching labels with the composition we want. By changing a label in Figure 5.x, we can use the local Helm Provider or the GCP provider to instantiate a Redis Instance.

Then creating new Database resources that will be provisioned in the Google Cloud Platform will look like this:

```
apiVersion: salaboy.com/v1alpha1
```

```
kind: Database
metadata:
  name: my-db-cloud-sql #A
spec:
  compositionSelector:
    matchLabels:
      provider: cloud #B
      type: dev
      kind: sql #C
  parameters:
    size: small
    mockData: false
```

The beauty of all these mechanisms is that no matter where our Databases are, we can connect our application's services just by pointing at them. Let's do that in the following section.

5.3.1 Connecting our services with the new provisioned infrastructure

When we create new Database resources, Crossplane will monitor the status of these Kubernetes resources against the status of the provisioned components inside the specific Cloud Provider, keeping them in sync and making sure that the desired configurations are applied. This means that Crossplane will make sure that our Databases are up and running, if for some reason that changes, Crossplane will try to reapply the configurations that we requested until what we requested is up and running.

If we don't have the application deployed in our KinD cluster, we can deploy it without installing PostgreSQL and Redis. As we have seen before, this can be disabled by setting two flags.

```
> helm install conference fmtok8s/fmtok8s-conference-chart --set
```

I strongly recommend you to check out the step-by-step tutorial that you can find here: <https://github.com/salaboy/from-monolith-to-k8s/tree/main/crossplane> to get your hands dirty with Crossplane and the Conference application. The best way to learn is by doing!

If we just run this command, no components (Redis and PostgreSQL) will be

provisioned by Helm. Still, the application's services will not know where to connect to the Redis and PostgreSQL instances that we created using our Crossplane Compositions. So we need to add more parameters to the charts so they know where to connect.

First, check which Databases you have available in your cluster:

```
> kubectl get dbs
NAME          SIZE   MOCKDATA   KIND      SYNCED   READY
my-db-keyvalue  small  false      keyvalue  True     True
my-db-sql       small  false      sql       True     True
```

The following Kubernetes Pods back these database instances:

```
> kubectl get pods
NAME                           READY   STATUS    RESTARTS   AGE
my-db-keyvalue-redis-master-0  1/1     Running   0          3m4
my-db-sql-postgresql-0        1/1     Running   0          104
```

Along with the Pods, four Kubernetes Secrets were created. Two to store the Helm Releases used by our Crossplane Compositions and two containing our new databases passwords that our applications will need to use to connect:

```
> kubectl get secret
NAME                      TYPE      DATA
my-db-keyvalue-redis      Opaque    1
my-db-sql-postgresql      Opaque    1
sh.helm.release.v1.my-db-keyvalue.v1  helm.sh/release.v1  1
sh.helm.release.v1.my-db-sql.v1      helm.sh/release.v1  1
```

Take a look at the Services available in the default namespace after we provisioned our databases:

```
> kubectl get services
```

With the Database Service names and secrets we can configure our conference application chart to not only not deploy Redis and PostgreSQL but to connect to the right instances by running the following command:

```
> helm install conference fmtok8s/fmtok8s-conference-chart -f app
```

Instead of setting all the parameters in the command we can use a file for the

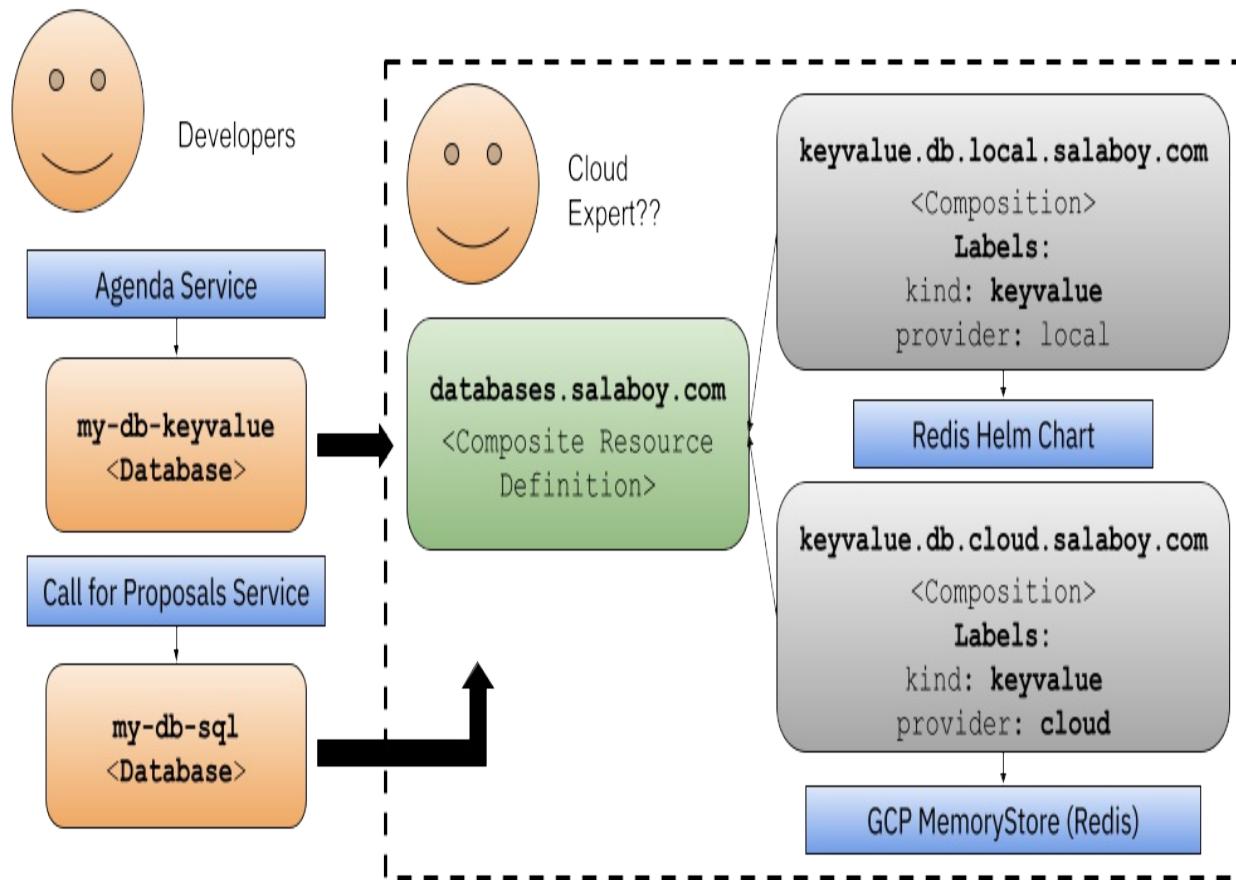
values to be applied to the chart, for this example, the `app-values.yaml` file look like this:

```
fmtok8s-agenda-service:  
  redis:  
    enabled: false #A  
  env:  
    - name: SPRING_REDIS_HOST  
      value: my-db-keyavalue-redis-master #B  
    - name: SPRING_REDIS_PORT  
      value: "6379"  
    - name: SPRING_REDIS_PASSWORD  
      valueFrom:  
        secretKeyRef:  
          name: my-db-keyavalue-redis #C  
          key: redis-password #D  
  
fmtok8s-c4p-service:  
  postgresql:  
    enabled: false  
  env:  
    - name: DB_ENDPOINT  
      value: my-db-sql-postgresql  
    - name: DB_PORT  
      value: "5432"  
    - name: SPRING_R2DBC_PASSWORD  
      valueFrom:  
        secretKeyRef:  
          name: my-db-sql-postgresql  
          key: postgres-password
```

In this `app-values.yaml` file, we are not only turning off the Helm dependencies for PostgreSQL and Redis, but we are also configuring the variables needed for the services to connect to our newly provisioned databases. The SPRING_REDIS_HOST and DB_ENDPOINT variables values are using the Kubernetes Service names created when we requested our Database resources by the installed Helm charts. Notice that if the databases were created in a different namespace, the SPRING_REDIS_HOST and DB_ENDPOINT variables values must include the fully qualified name of the service, which includes the namespace, for example, my-db-sql-postgresql.default.svc.cluster.local (where `default` is the namespace). The SPRING_REDIS_PASSWORD and SPRING_R2DBC_PASSWORD reference the Kubernetes Secrets created specifically for these new instances.

Depending on your applications, and the frameworks that you are using, these environment variables names will be different, and we will look into this topic later on in the developer experience chapters.

Figure 5.x Enabling different teams to work together and focus on their tasks at hand



All this effort enables us to split the responsibility of defining, configuring and running all the application infrastructure to another team that is not responsible for working on the application's services. Services can be released independently without worrying about which databases are being used or when they need to be upgraded. Developers shouldn't be worrying about Cloud Provider accounts or if they have access to create different resources, hence another team with a completely different set of skill sets can take care of creating Crossplane compositions and configuring Crossplane providers.

We have also enabled teams to request application infrastructure components by using Kubernetes Resources, this enables them to create their own setups for experimentation and testing or to quickly set up new instances of the application. This is a major shift in how we (as developers) were used to doing things, as before Cloud Providers and in most companies today, to have access to a database, you need to use a ticketing system to request another team to provision that resource for you, and that can take weeks!

To summarize what we have achieved so far, we can say that:

- We abstracted how to provision Local and Cloud specific components such as PostgreSQL and Redis databases and all the configurations needed to access the new instances.
- We have exposed a simplified interface for the Application teams, which is Cloud Provider independent because it relies on the Kubernetes API.
- Finally, we have connected our application service to the newly provisioned instance by relying on a Kubernetes Secret that is created by Crossplane, containing all the details required to connect to the newly created instance.

You can follow a step-by-step tutorial that covers all the steps described in this section here: <https://github.com/salaboy/from-monolith-to-k8s/tree/main/crossplane>

If you use mechanisms like Crossplane Compositions to create higher-level abstractions, you will be creating your domain-specific concepts that your teams can consume using a self-service approach. We have created our Database concept by creating a Crossplane Composite Resource that uses Crossplane Compositions that knows which resources to provision (and in which Cloud Provider).

But we need to be cautious, we cannot expect every developer to understand or be willing to use tools like the ones we have discussed (Crossplane, ArgoCD, Tekton, etc.). Hence we need a way to reduce all the complexity these tools introduce. In the following chapters, we will be bringing all these tools together using some of the concepts around Platform Engineering that we discussed in Chapter 1.

Let's jump into the next chapter, and let's build Platforms on top of Kubernetes.

5.4 Summary

- Cloud-Native applications depend on application infrastructure to run, as each service might require different persistent storages, a message broker to send messages, and other components to work.
- Creating application infrastructure inside Cloud Providers is easy and can save us a lot of time, but then we rely on their tools and ecosystem.
- Provisioning infrastructure in a cloud-agnostic way can be achieved by relying on the Kubernetes API and tools like Crossplane, which abstracts the underlying Cloud Provider and lets us define which resources need to be provisioned using Crossplane Compositions.
- Crossplane provides support for major Cloud Providers. It can be extended for other service providers, including 3rd Party tools that might not be running on Cloud Providers (for example, legacy systems that we want to manage using the Kubernetes APIs).
- By using Crossplane Composite Resource Definitions, we create an interface that application teams can use to request cloud resources using a self-service approach

6 Let's build a platform on top of Kubernetes

This chapter covers

- The main features that our platform provides on top of Kubernetes
- The challenges with multi-cluster and multi-tenant setups and how to define our Platform architecture
- How does a platform on top of Kubernetes looks like

So far, we have looked at what Platform Engineering is, why we need to think about Platforms in the context of Kubernetes, and how teams must choose the tools they can use from the CNCF landscape (Chapter 1). Then we jumped into figuring out how our applications would run on top of Kubernetes (Chapter 2), and how we were going to build, package, deploy (Chapters 3 and 4), and connect these applications to other services that they need to work (Chapter 5). This chapter will look into putting all the pieces together to create a walking skeleton for our platform. We will use some of the Open Source projects introduced in the previous chapters and new tools to solve some of the challenges we will face when creating the first iteration of our platform.

This chapter is divided into three main sections:

- The Importance of the Platform APIs
- Kubernetes Platform Architecture: multi-tenancy and multi-cluster challenges and how we can architect a scalable platform
- Introducing our Platform Walking Skeleton: we need to start somewhere; let's build a platform on top of Kubernetes

Let's start by considering why defining the Platform APIs is the first step to platform building.

6.1 The Importance of the Platform APIs

In Chapter 1, we looked at existing platforms, such as Google Cloud Platform, to understand what key features they offer to teams that are building and running applications for the cloud. We now need to compare this to the platforms that we are building on top of Kubernetes, as these platforms share some common goals and features with Cloud Providers while at the same time being closer to our organizations' domains.

Platforms are nothing other than software that we will design, create and maintain. As with any good software, our platform will evolve to help teams with new scenarios, make our teams more efficient by providing automation, and give us the tools to make the business more successful. As with any other software, we will start by looking at the Platform APIs, which will not only provide us with a scope that is manageable to start with but also it will define the contracts and behaviors that our platform will provide to its users.

Our Platform APIs are important, as good APIs can simplify the life of development teams wanting to consume services from our platform. If our Platform APIs are well designed, more tailored tools like CLIs and SDKs can be created to assist users in consuming our Platform services.

If we build a custom and more domain-specific API for our platform, we can start by tackling one problem at a time and then expand these APIs/interfaces to cover more and more workflows, even for different teams.

Once we understand which workflows we want to cover and have an initial platform API Dashboards and more tooling can be created to help teams to adopt the platform itself.

Let's use an example to make it more concrete. I hope you can translate the example I am showing here into more concrete examples inside your organization. All the mechanisms should apply in the same way.

Let's enable our development teams to request new Development Environments.

6.1.1 Requesting Development Environments

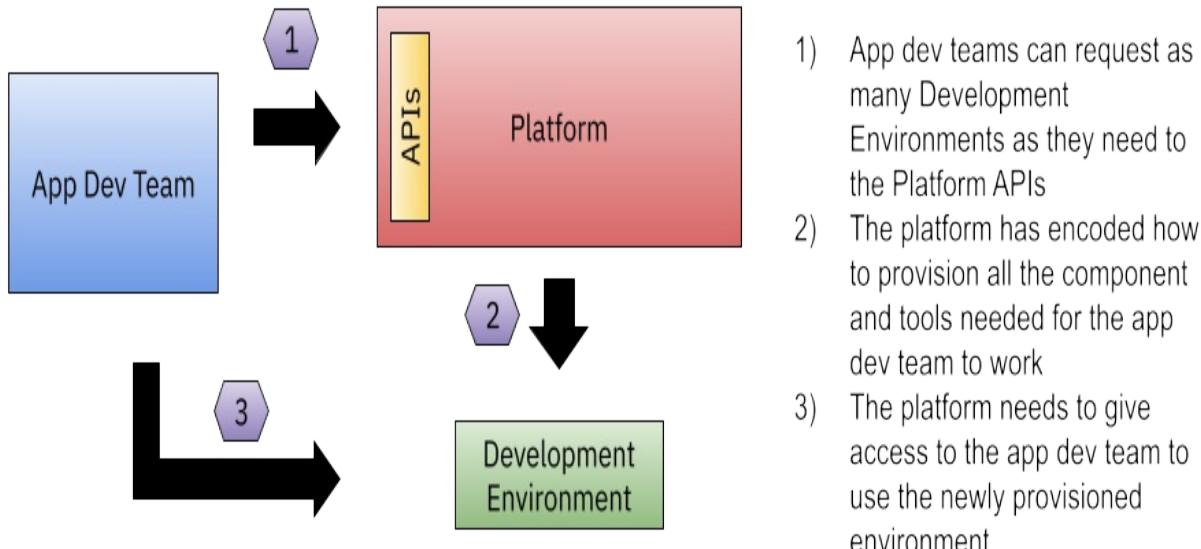
A common scenario where a Platform can help teams get up to speed when they start working on new features is provisioning them all they need to do their work. To achieve this task, the Platform Engineering team must understand what they will work on, what tools they need, and which other services must be available to succeed.

Once the Platform Engineering team has a clear idea of what a team requires, they can define an API enabling teams to create requests to the platform to provision new development environments. Behind this API, the platform will have the mechanisms to create, configure and provide access to teams to all the resources created.

For our Conference Application example, if a development team is extending the application, we will need to ensure they have a running version of the application to work against and test changes. This isolated instance of the application will also need to have its databases and other infrastructural components required for the application to work. More advanced use cases include loading the application with mock data, allowing teams to test their changes with pre-populated data.

The interactions between the application development team and the platform should look like this:

Figure 6.x Application development team interactions with Platform



As mentioned before, Development Environments are just an example. You must ask yourself what tools your teams need to do their work. A development environment might not be the best way to expose tools to a team of data scientists, for example, as they might need other tools to gather and process data or train machine learning models.

Implementing this simple flow in Kubernetes is not an easy task. To implement this scenario, we need to:

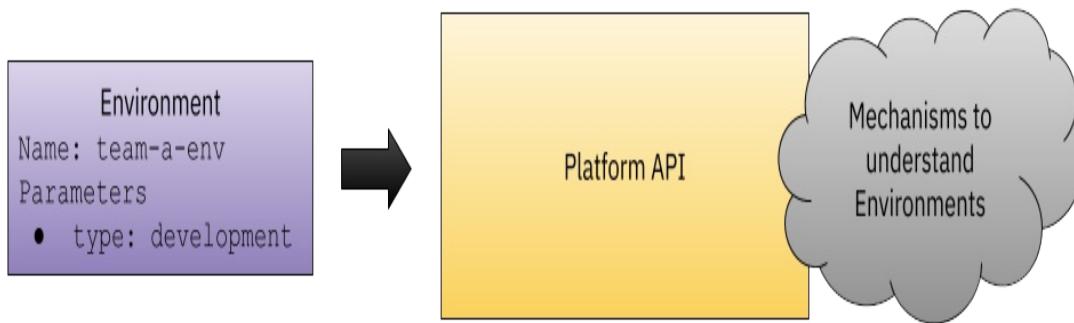
- Create new APIs that understand Development Environment requests
- Have the mechanisms to encode what a Development Environment means for our teams
- Have the mechanisms to provision and configure components and tools
- Have the mechanisms to enable teams to connect to the newly provisioned environments

There are several alternatives to implement this scenario, starting with creating our own custom Kubernetes extensions or using more specialized tools for development environments. But before diving into implementation details, let's define what our Platform API would look like for this scenario.

As with Object Oriented Programming, our APIs are **Interfaces** that can be implemented by different **classes**, which finally provide concrete behavior. For provisioning Development Environments, we can define a very simple

interface called “Environment”. Development teams requesting a new Development Environment can create new requests to the platform by creating new Environment resources. The “Environment” interface represents a contract between the user and the platform. This contract can include parameters to define the type of environment the team is requesting or options and parameters they need to tune for their specific request.

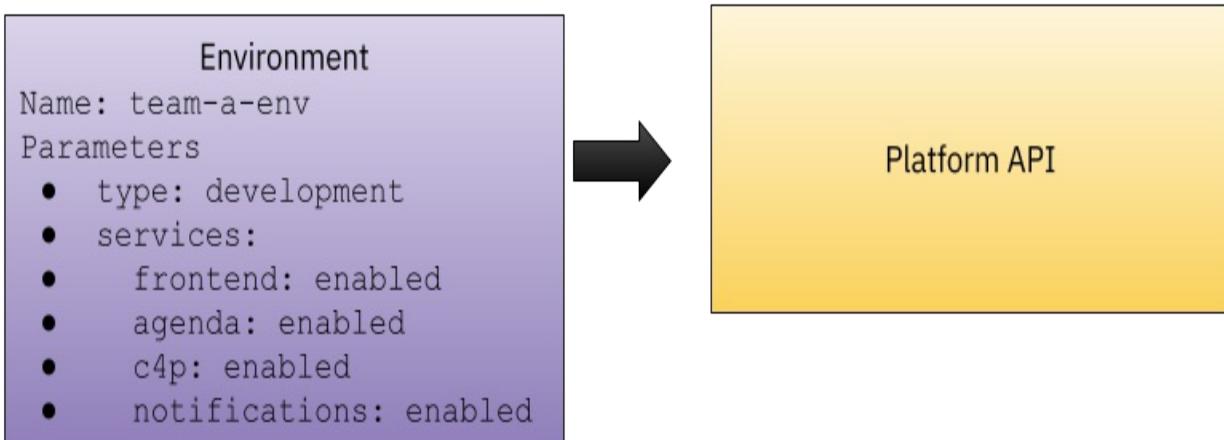
Figure 6.x Environment resource defined by Platform API



It is important to note that the Environment interface shouldn't include (or leak) any implementation detail about our environments. These resources serve as our abstraction layer to hide complexity from our platform users about these environments will be provisioned. The simpler these resources are, the better for the platform users. In this example, the platform can use the Environment `type` parameter to decide which environment to create, and we can plug new types as we evolve our platform mechanisms.

Once we recognize which interfaces we need, we can slowly add parameters that teams can configure. For our example, it might make sense to parameterize which services we want to deploy in our Environment if we also want the application infrastructure to be created or to connect our services to existing components. The possibilities here are endless, depending on what makes sense for your teams to parameterize. The platform team here can control what is possible and what is not. Expanding our Environment interface to cover more use cases can look as follows:

Figure 6.x Extended Environment resource to enable/disable application's services



Encoding this Environment resource into a format like JSON or YAML to implement the platform API is straightforward:

```
"environment": {
  "name": "team-a-env",
  "type": "development",
  "services": {
    "frontend": "enabled",
    "agenda": "enabled",
    "c4p": "enabled",
    "notifications": "disabled"
  }
}
```

Once the interface is defined, the next logical step is to provide one implementation to provision these Environments for our platform users. Before jumping into implementation details, we need to cover two of the main challenges you will face when deciding where the mechanisms for implementing these environments will reside.

We need to talk about something important: how will we architect our Platform?

6.2 Platform Architecture

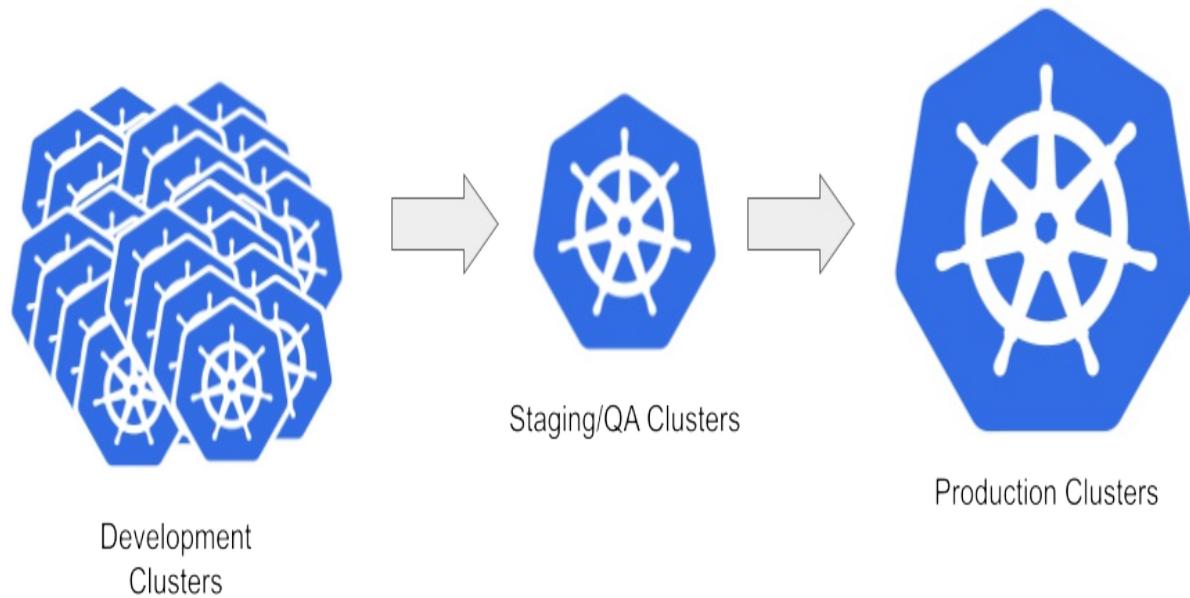
On the technical side of building platforms, we will encounter challenges requiring the platform team to make some hard choices. In this section, we will talk about how we can architect a Platform that allows us to encapsulate

a set of tools behind our Platform APIs and enable development teams to perform their tasks without worrying about which tools are being used by the platform to provision and wire up complex resources.

Because we are already using Kubernetes to deploy our workloads (Conference Application), it makes sense also to run the Platform services on top of Kubernetes, right? But would you run the Platform services and components right beside your workloads? Probably not. Let's step back a bit.

If your organization adopts Kubernetes, you will likely already deal with multiple Kubernetes clusters. As we discussed in Chapter 4 for Environment Pipelines, your organization probably has a Production Environment, Staging, or QA environment already in place. If you want your application development teams to work on environments that feel like the production environment, you will need to enable them with Kubernetes Clusters.

Figure 6.x Environment Clusters, do you want to enable developers to have their own environments?

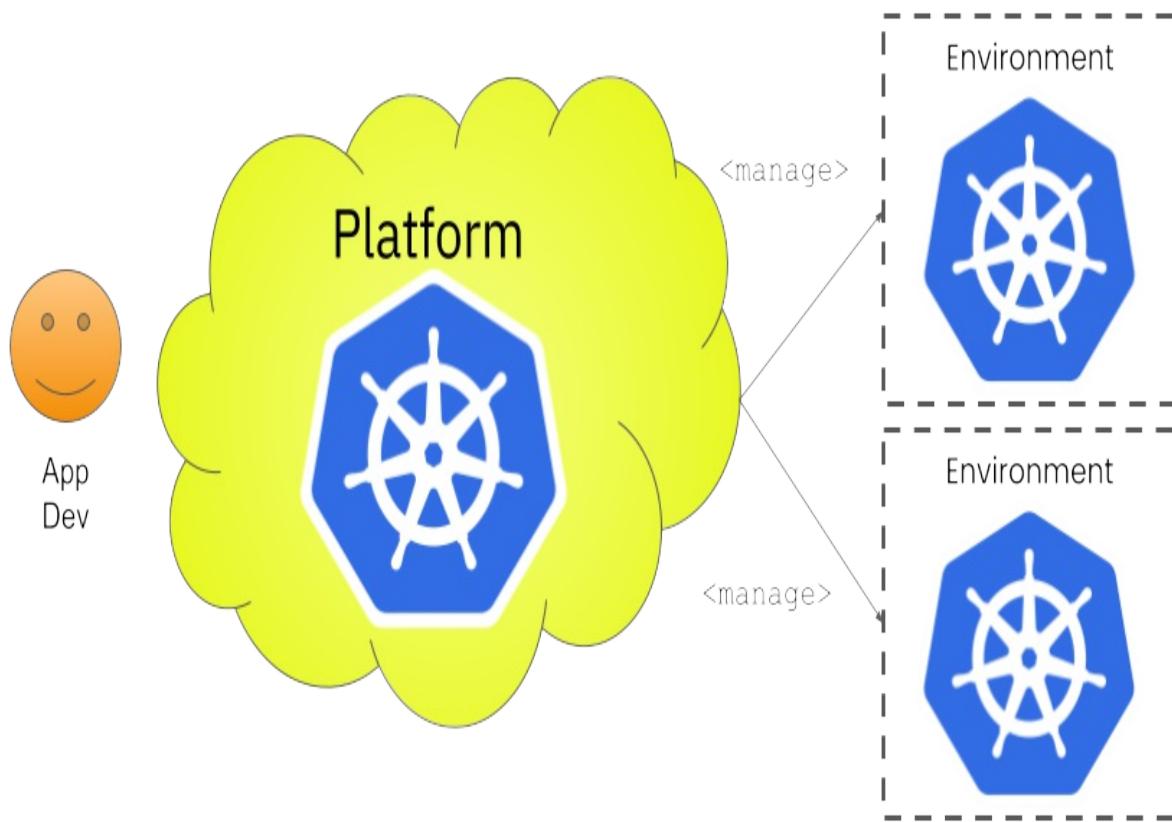


While the Production Cluster(s) and Staging/QA Cluster(s) should be handled carefully and hardened to serve real-life traffic, development environments tend to be more ephemeral and sometimes even run on the development team

laptops. One thing is certain, you don't want to be running any platform-related tools in any of these environments, the reason is simple, tools like Crossplane, ArgoCD, or Tekton shouldn't be competing for resources with our application's workloads.

When looking at building platforms on top of Kubernetes, teams tend to create one or more special clusters to run platform-specific tools. The terms are not standardized yet, but creating a Platform or Management cluster to install platform-wide tools is becoming increasingly popular.

Figure 6.x Platform Cluster with Platform tools managing environments

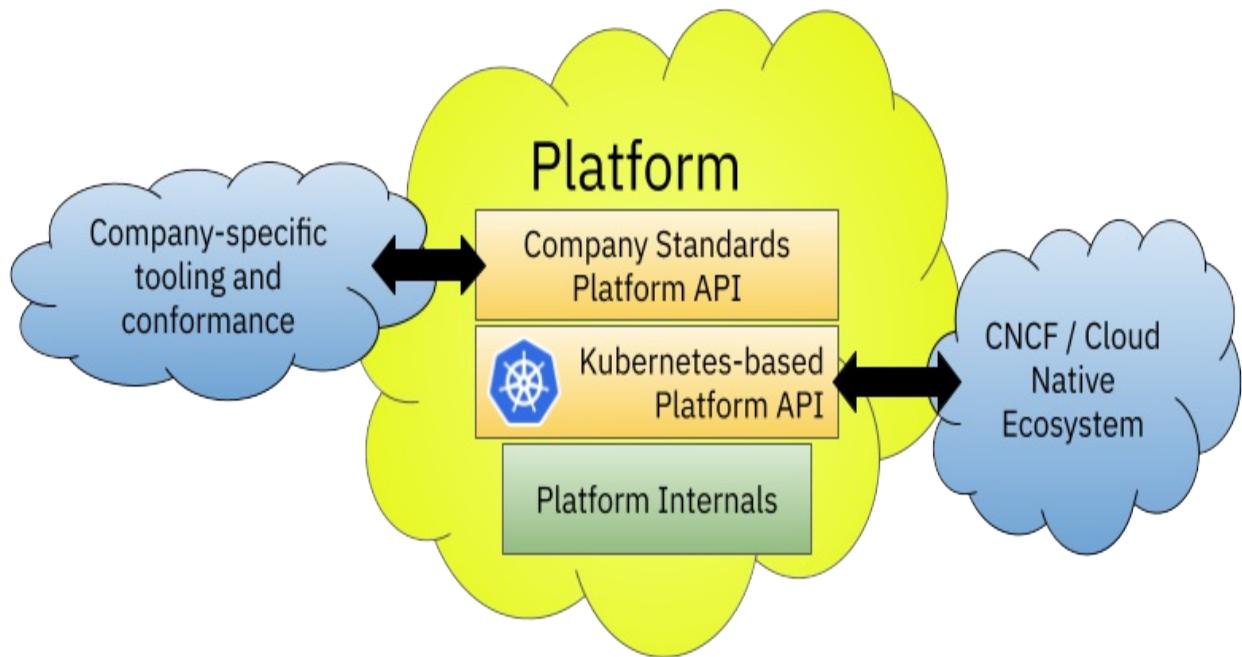


By having separate Platform Cluster(s) you can install the tools that you need to implement your platform capabilities while at the same time building a set of management tools to control environments where your workloads run.

Now that you have a separate place to install these tools, you can also host

the Platform API on this cluster, once again, to not overload your workload clusters with platform components. Wouldn't it be great to reuse or extend the Kubernetes API to serve also as our Platform APIs? There are pros and cons to this approach. For example, suppose we want our Platform API to follow Kubernetes conventions and behaviors. In that case, our platform will leverage the declarative nature of Kubernetes and promote all the best practices followed by the Kubernetes APIs, such as versioning, the resource model, etc. For non-Kubernetes users, this API might be too complex, or the organization might follow other standards when creating new APIs that do not match the Kubernetes style. **If we reuse the Kubernetes APIs for our Platform APIs, all the CNCF tools designed to work with these APIs will automatically work with our Platform.** Our platform automatically becomes part of the ecosystem. In the last couple of years, I've seen a trend around teams adopting the Kubernetes APIs as their Platform APIs.

Figure 6.x Kubernetes-based Platform APIs complemented by Company-specific APIs

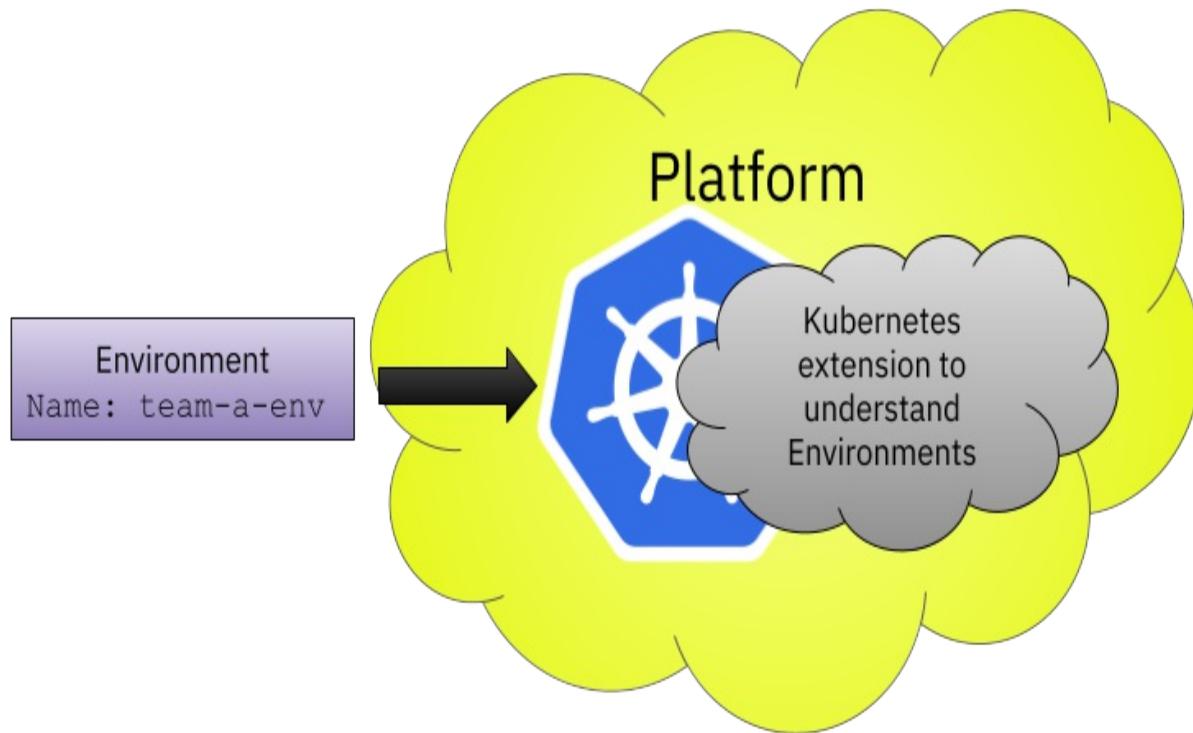


Adopting the Kubernetes APIs for your Platform API doesn't stop you from building a layer on top for other tools to consume or to follow the company's standards. By having a Kubernetes-based API layer you can access all the amazing tools that are being created in the CNCF and Cloud-Native space. On top of the Kubernetes-based APIs, another layer can follow company

standards and conformance checks, enabling easier integrations with other existing systems.

Following our previous example, we can extend Kubernetes to understand our Environment requests and provide the mechanisms to define how these environments will be provisioned.

Figure 6.x Extending Kubernetes to understand Environments and Serve as our Platform APIs



In principle, this looks good and doable. Still, before implementing these Kubernetes extensions to serve our Platform API and central hub of Platform tooling, we need to understand the questions that our platform implementation will try to answer. Let's take a look at the main platform challenges that teams in these scenarios will face.

6.2.1 Platform Challenges

Sooner or later, if you are dealing with multiple Kubernetes Clusters, you

will need to manage them and all the resources related to these clusters. What does it take to manage all these resources? The first step to start understanding the underlying problems is to understand who the users of our platforms are. Are we building a platform for external customers or internal teams? What are their needs and the level of isolation that they need to operate autonomously without bothering their neighbors? What guardrails do they need to be successful?

While I cannot answer these questions for all use cases, one thing is clear, platform tools and workloads need to be separated. We need to encode in our platform our tenant boundaries based on each tenant's expectations. No matter if these tenants are customers or internal teams. We must set clear expectations for our platform users about our tenancy model and guarantees. Hence, they understand the limitations of the resources the platform gives them to do their work.

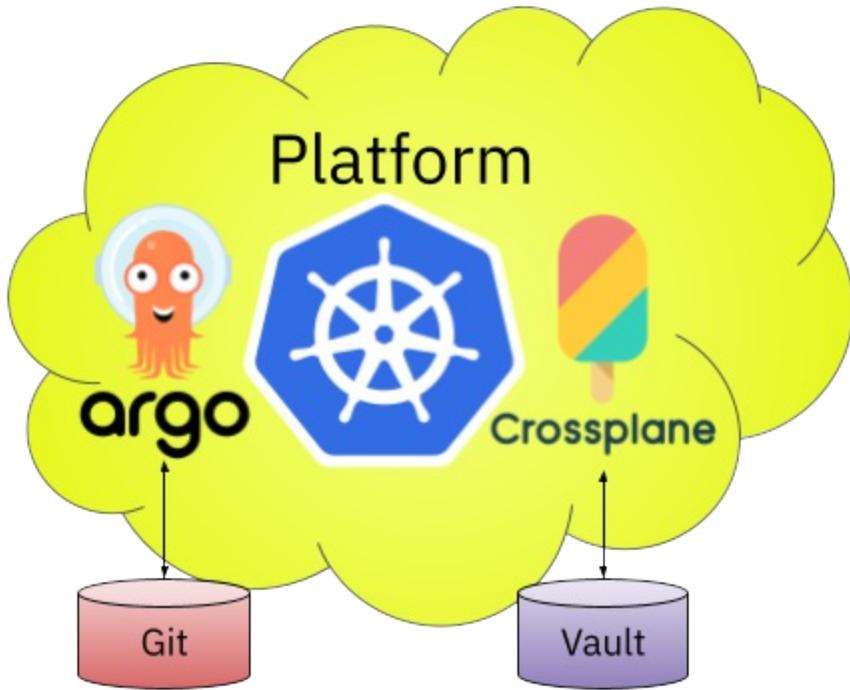
One thing is clear, the platform that we are going to build needs to take care of encoding all these decisions. Let's start by looking at one of the first challenges you will encounter when trying to manage multiple clusters.

6.2.2 Managing more than one cluster

The platform we will build needs to manage and understand which environments are available for teams to use. More importantly, it should enable the team to request their own Environments whenever needed.

Using the Kubernetes APIs as our Platform API to request environments, we can leverage tools like ArgoCD (covered in Chapter 4) to persist and sync our Environment configurations to live Kubernetes Clusters. Managing our Clusters and Environments becomes just managing Kubernetes Resources that must be synced to our Platform Cluster(s).

Figure 6.x Combining GitOps and Crossplane for managing Environments and Clusters



By combining tools like ArgoCD and Crossplane inside our Platform Clusters, we promote using the same techniques that we already discussed in Chapter 4, for Environment Pipelines, which sync application-level components, but now for managing high-level platform concerns.

As you can see in the previous figure, our platform configuration itself will become more complex, as it will need to have its source of truth (Git Repository) to store the environment and resources that the platform is managing as well as it will need to have access to a Secret Store such as Hashicorp Vault to enable Crossplane to connect and create resources in different cloud providers. In other words, you now have two extra concerns. You will need to define, configure, and give access to one or more Git repositories to contain the configurations for the resources created in the platform. You must manage a set of Cloud Provider accounts and their credentials so the platform cluster(s) can access and use these accounts.

While the example in section 6.3 doesn't go in-depth on configuring all these concerns, it provides a nice playground to build on top and experiment with more advanced setups depending on your teams' requirements.

My recommendation to prioritize which configurations make sense first is to

understand what your teams or tenants will do with the resources and their expectations and requirements. Let's dig a bit more into that space.

6.2.3 Isolation and multi-tenancy

Depending on your tenants' (teams, internal or external customers) requirements, you might need to create different isolation levels so they don't disturb each other when working under the same Platform roof.

Multi-tenancy is a complicated topic in the Kubernetes community. Using Kubernetes RBAC (Role-based access control), Kubernetes Namespaces, and multiple Kubernetes Controllers that might have been designed with different tenancy models in mind makes it hard to define isolation levels between different tenants inside the same cluster.

Companies embarking on adopting Kubernetes tend to take one of the following approaches for isolation:

- Kubernetes Namespaces:

Pros:

- Creating namespaces is very easy, and it has almost zero overhead
- Creating namespaces is cheap, as it is just a logical boundary that Kubernetes uses to separate resources inside the cluster

Cons:

- Isolation between namespaces is very basic, and it will require RBAC roles to limit users' visibility outside the namespaces they have been assigned. Resource quotas must also be defined to ensure that a single namespace is not consuming all the Cluster resources.
- Providing access to a single namespace requires sharing access to the same Kubernetes APIs endpoints that admins and all the other tenants are using. This limits the operations clients can execute on the cluster, such as installing cluster-wide resources.

- Kubernetes Clusters:

Pros:

- Users interacting with different clusters can have full admin capabilities enabling them to install as many tools as they need.
- You have full isolation between Clusters, tenants connecting to

different clusters will not share the same Kubernetes API Server endpoints. Hence each cluster can have different configurations for how scalable and resilient they are. This allows you to define different tenants' categories based on their requirements.

Cons:

- This approach is expensive, as you will be paying for computing resources to run Kubernetes. The more clusters you create, the more you will be spending money on running Kubernetes.
- Managing multiple Kubernetes Clusters becomes complex if you enable teams to create (or request) their own. Zombie Clusters (clusters nobody uses and abandoned) start to pop up, wasting valuable resources.
- Sharing resources, installing and maintaining tools across a fleet of different Kubernetes Clusters is challenging and a full-time job

Based on my experience, teams will create isolated Kubernetes Clusters for sensitive environments such as Production Environments and performance testing. These sensitive environments tend not to change and are only managed by operation teams. When you shift towards development teams and more ephemeral environments for testing or day-to-day development tasks, using a big cluster with namespaces is a common practice.

Choosing between these two options is hard, but what is important is not to over-commit to just a single option. Different teams might have different requirements, so in the next section, we will look at how the platform can abstract these decisions, enabling teams to access different setups depending on their needs.

6.3 Our Platform walking skeleton

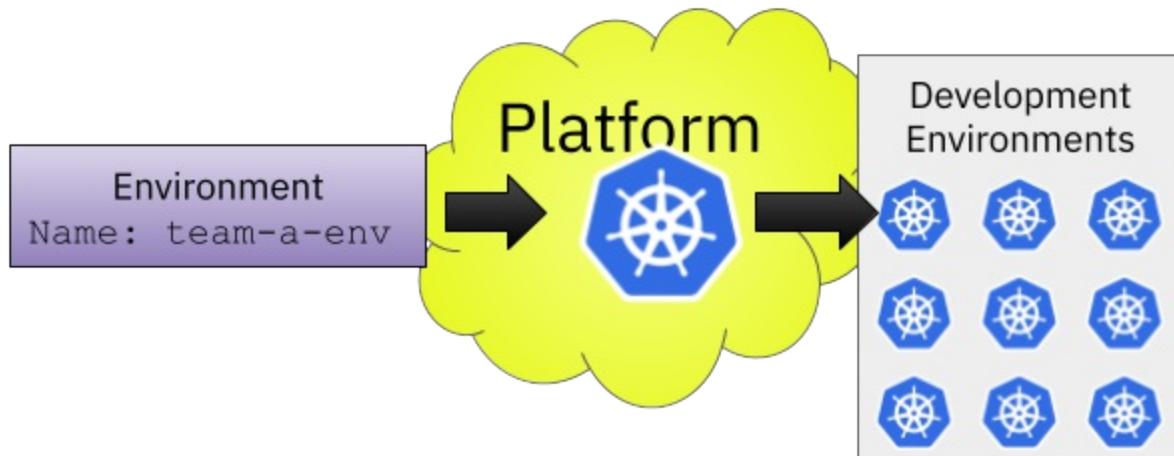
This section will look into creating a very simple platform that allows internal teams to create Development Environments. Because our teams are deploying the conference application to Kubernetes Clusters, we want to offer them the same developer experience.

You can follow a step-by-step tutorial, where you will install and interact with the Platform walking skeleton: <https://github.com/salaboy/from->

monolith-to-k8s/tree/main/platform/prototype

To achieve this, we will use some tools that we used before, like Crossplane, to extend Kubernetes to understand Development Environments. Then we will use a project called vcluster (<https://vcluster.com>) to provision small Kubernetes clusters for our teams. These clusters are isolated, allowing teams to install extra tools without worrying about what other teams are doing. Because teams will have access to the Kubernetes APIs, they can do whatever they need with the cluster without requesting complicated permissions to debug their workloads.

Figure 6.x Building a platform prototype to provision development environments



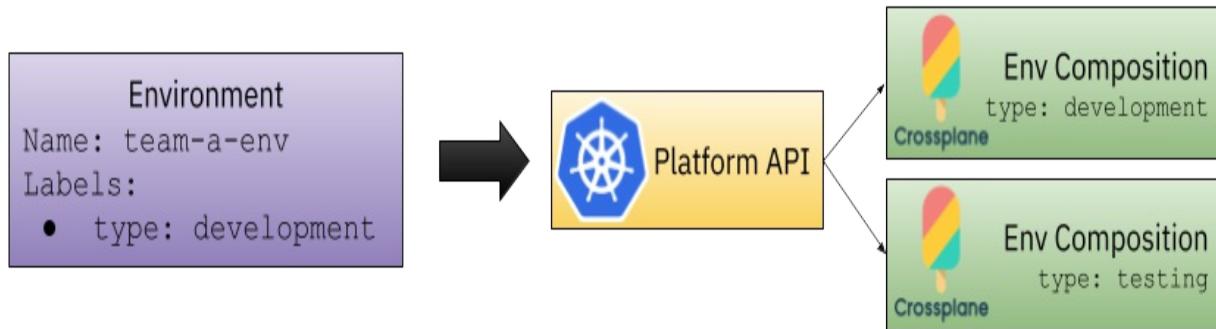
We will keep things simple for the walking skeleton, but the platforms are complicated.

I can't stress enough the importance that this example, on purpose, is using existing tools instead of creating our custom Kubernetes extensions. If you create custom controllers to manage Environments, you create a complex component that will require maintenance and probably overlaps 95% with the mechanisms shown in this example.

In the same way that we started this chapter talking about our Platform APIs, let's take a look at how we can build these APIs without creating our custom Kubernetes extensions. We will use Crossplane Compositions in the same

way we did for our Databases in Chapter 5, but now we will implement our Environment Custom Resource. We can keep the Environment resource simple and use Kubernetes label matchers and selectors to match a resource with one of the possible compositions we can create to provision our environments.

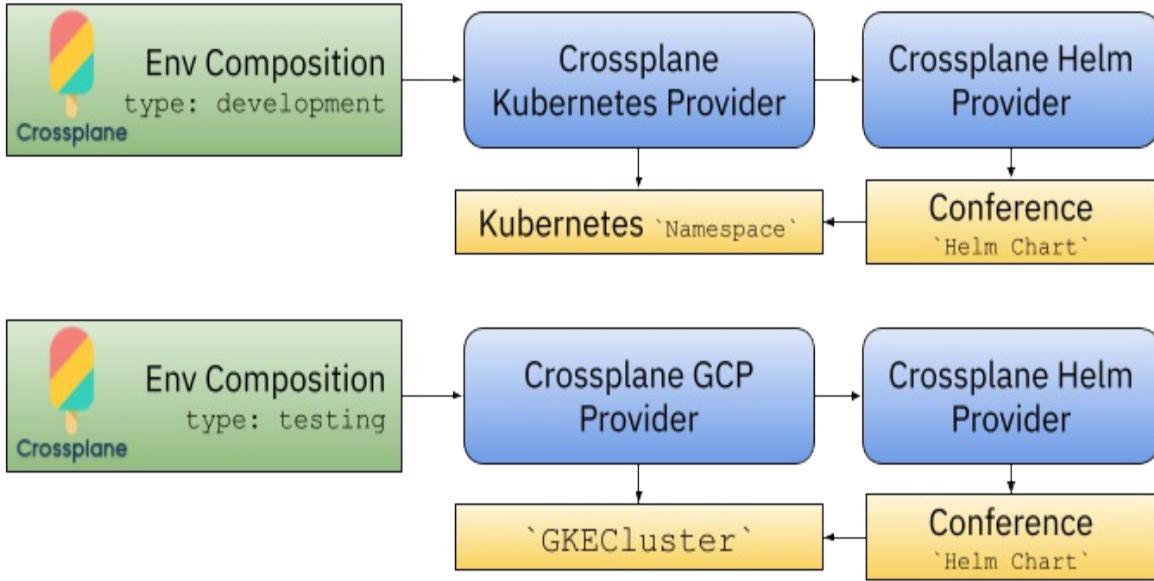
Figure 6.x Mapping Environment Resources to Crossplane Compositions



Crossplane Compositions offer the flexibility to use different providers to provision and configure resources together, as we saw in Chapter 5, multiple compositions (implementations) can be provided for different kinds of environments.

For this example, we want each Environment to be isolated from the other to avoid teams unintentionally deleting others' team resources. The two most intuitive ways of creating isolated environments would be to create a new namespace per Environment or a full-blown Kubernetes Cluster for each environment.

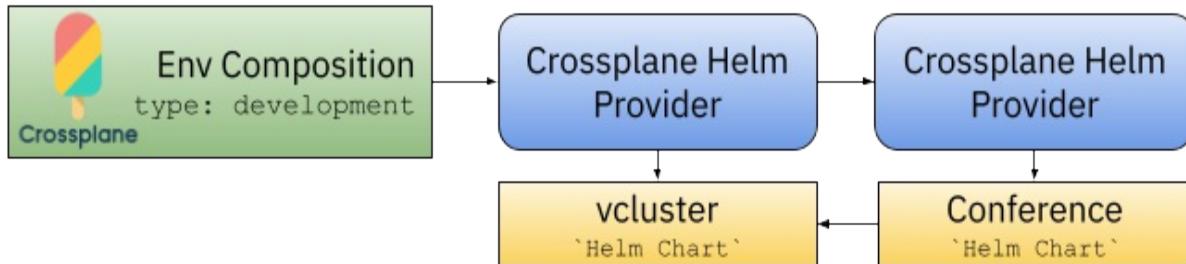
Figure 6.x Different Environment Compositions, namespace, and GKECluster



While creating a fully-fledged Kubernetes Cluster might be overkill for every development team, a Kubernetes Namespace might not provide enough isolation for your use case, as all teams will interact with the same Kubernetes API server.

For this reason, we will be using `vcluster` in conjunction with the Crossplane Helm Provider, which gives us the best of both worlds without the costs of creating new clusters.

Figure 6.x Using `vcluster` to create isolated environments



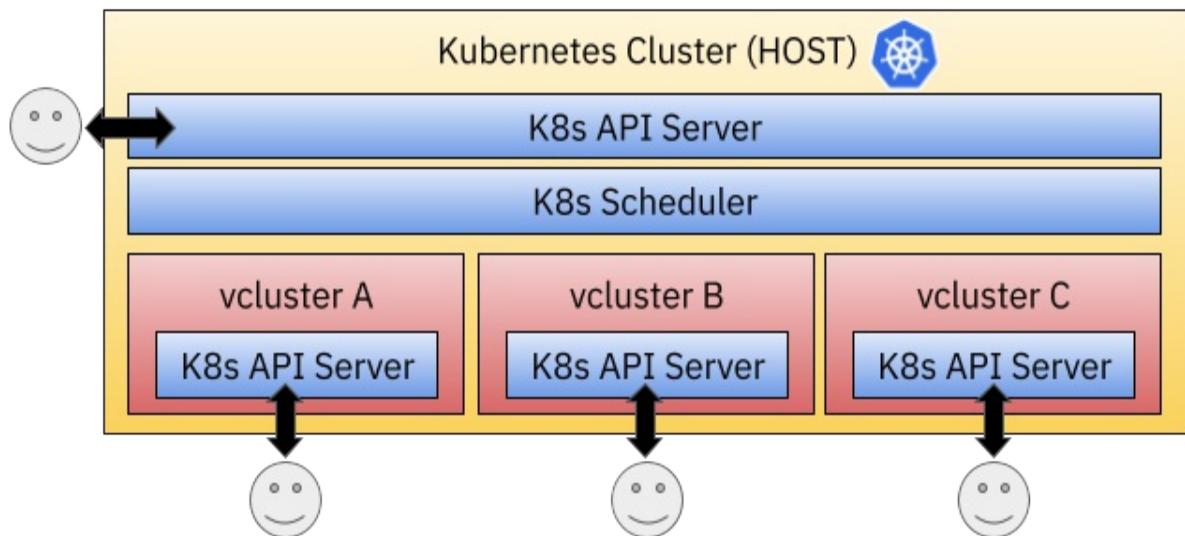
You might be wondering: what is a `vcluster`? And why are we using the Crossplane Helm Provider to create one? While `vcluster` is just another option that you can use to build your platforms, I consider it a key tool for every Platform Engineer toolbox.

6.3.1 `vcluster` for Virtual Kubernetes Clusters

I am a big fan of the `vcluster` project. If you are discussing multi-tenancy on top of Kubernetes `vcluster` tends to pop up in the conversation as it offers a really nice alternative to the Kubernetes Namespaces vs. Kubernetes Clusters discussions.

`vcluster` focuses on providing Kubernetes API Server isolation between different tenants by creating Virtual Clusters inside your existing Kubernetes Cluster (host cluster).

Figure 6.x `vcluster` provides isolation at the Kubernetes (K8s) API Server



By creating new virtual clusters, we can share an isolated API server with tenants where they can do whatever they need without worrying about what other tenants are doing or installing. For scenarios where you want each tenant to have cluster-wide access and full control of the Kubernetes API Server, `vcluster` provides a simple alternative to implement this.

Creating a vcluster is easy, you can create a new vcluster installing the vcluster Helm Chart.

Alternatively, you can use the `vcluster` CLI to create and connect to it.

<DIAGRAM>

Finally, a great table comparing `vcluster`, Kuebernetes Namespaces and Kubernetes Clusters can be found in their documentation, as if you are already having these conversations with your teams this table makes it crystal clear to explain the advantages and trade offs.

Figure 6.x Kubernetes `Namespaces` vs `vcluster` vs Kubernetes Cluster tenants pros and cons

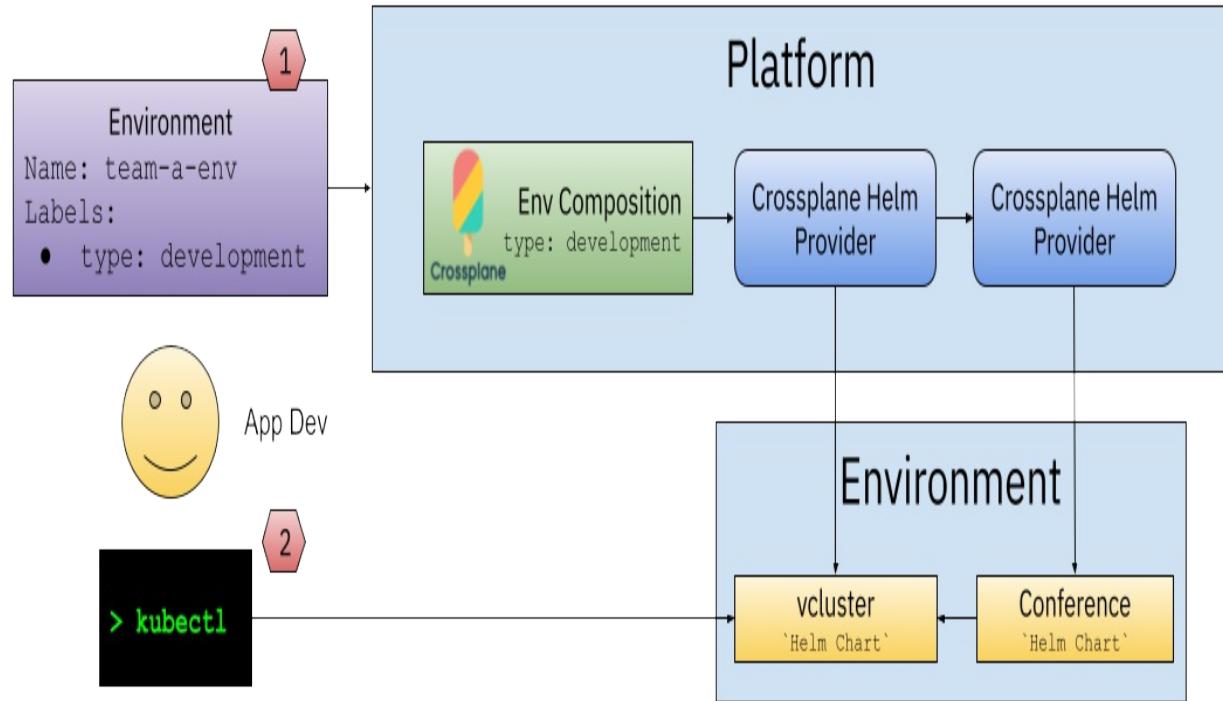
Separate Namespace For Each Tenant		vcluster	
		Separate Cluster For Each Tenant	
Isolation	very weak	strong	very strong
Access For Tenants	very restricted	vcluster admin	cluster admin
Cost	very cheap	cheap	expensive
Resource Sharing	easy	easy	very hard
Overhead	very low	very low	very high

Let's see what our platform walking skeleton looks like for teams that want to create, connect and work against new Environments that use `vcluster`.

6.3.2 Platform experience and next steps

The platform walking skeleton implemented in the GitHub repository:
<https://github.com/salaboy/from-monolith-to-k8s/tree/main/platform/prototype> allows teams connected to the Platform API to create new Environment resources and submit a request for the platform to provision it for them.

Figure 6.x Using Crossplane and `vcluster` to create isolated environments for application development teams



The Platform cluster uses Crossplane and Crossplane Compositions to define the provision of the environment. To run the composition in a local Kubernetes Cluster (and not require access to a specific Cloud Provider), the walking skeleton uses `vcluster` to provision each Environment on its own (virtual) Kubernetes Cluster. Having separate Kubernetes Clusters enables teams to connect to these environments and do the work they need to do with our Conference application, which is by default installed when the environment is created.

A natural extension to the walking skeleton would be to create the compositions to create environments spawning Kubernetes Clusters on a Cloud Provider or managing the Environment resources inside a Git repository, in contrast, to require application development teams to connect directly with the Platform APIs.

Now that we have a simple walking skeleton of our platform let's look at

what we can do to make application development teams more efficient when working with these environments.

In the next couple of chapters, we will explore the capabilities the platform should provide when creating environments for application development teams. Different tools can implement these capabilities, but it is important to recognize what kind of functionalities should be offered to teams so they can be more efficient. Chapter 7 covers Release Strategies and why they are important to enable teams to experiment and release more software. Chapter 8 covers Shared Concerns that you will need to provide to all services of your applications and different approaches to facilitate these mechanisms to your developers.

6.4 Summary

- Building platforms on top of Kubernetes is a complex task that involves combining different tools to serve teams with different requirements.
- Platforms are software projects as your business applications, starting by understanding who the main users will be and defining clear APIs is the key to prioritizing tasks on how to build your own platform.
- Managing multiple Kubernetes Clusters and dealing with tenant isolations are two main challenges that Platform teams face early on in their platform-building journey
- Having a platform walking skeleton can help you to demonstrate to internal teams what can be built to speed up their Cloud-Native journey
- Using tools like Crossplane, ArgoCD, `vcluster`, and others can help you to promote Cloud-Native best practices at the platform level but most importantly, avoid the urge to create your own custom tools and ways to provision and maintain complex configurations of Cloud-Native resources

7 Platform capabilities I: Enabling developers to experiment

This chapter covers

- Why providing release strategies as a platform capability enables teams to experiment and release more software when working with Kubernetes
- Kubernetes releases strategies using Deployments, Services and Ingress
- How to implement rolling updates, canary releases, blue/green deployments, and A/B testing
- The limitations and challenges of using Kubernetes built-in resources for releasing software efficiently
- Knative Serving advanced traffic routing capabilities for reducing releases risk and speeding up release cadence

If you and your teams are worried about deploying a new version of your service, you are doing something wrong. You are slowing down your release cadence due to the risk of making or deploying changes to your running applications.

This chapter focuses on one of the capabilities that good platforms should offer their users. If your platform doesn't enable teams with the right mechanisms to experiment by releasing software using different strategies, you are not fully leveraging the power of Kubernetes and its building blocks.

Reducing risk and having the proper mechanisms to deploy new versions drastically improves confidence in the system. It also reduces the time from a requested change until it is live in front of your users. New releases with fixes and new features directly correlate to business value, as software is not valuable unless it serves our company's users.

While Kubernetes built-in resources such as Deployments, Services, and Ingresses provide us with the basic building blocks to deploy and expose our services to our users, a lot of manual and error-prone work must happen to

implement well-known release strategies.

This chapter is divided into two main sections:

- Release strategies using Kubernetes built-in mechanisms
 - Rolling Updates
 - Canary Releases
 - Blue/Green Deployments
 - A/B Testing
 - Limitations and complexities of using Kubernetes built-in mechanisms
- Knative Serving: advanced traffic management and release strategies
 - Introduction to Knative Serving
 - Advanced traffic-splitting features
 - Knative Serving applied to our Conference application
- Argo Rollouts: release strategies automated with GitOps

In this chapter, you will learn about Kubernetes built-in mechanisms and Knative Serving to implement release strategies such as rolling updates, canary releases, blue-green deployments, and A/B testing. This chapter covers these patterns, understanding the implementation implications and limitations.

The second half of the chapter introduces Knative Serving, which allows us to implement more fine-grain control on how traffic is routed to our application's services. We explore how Knative Serving traffic-splitting mechanisms can be used to improve how we release new versions of our Cloud-Native application.

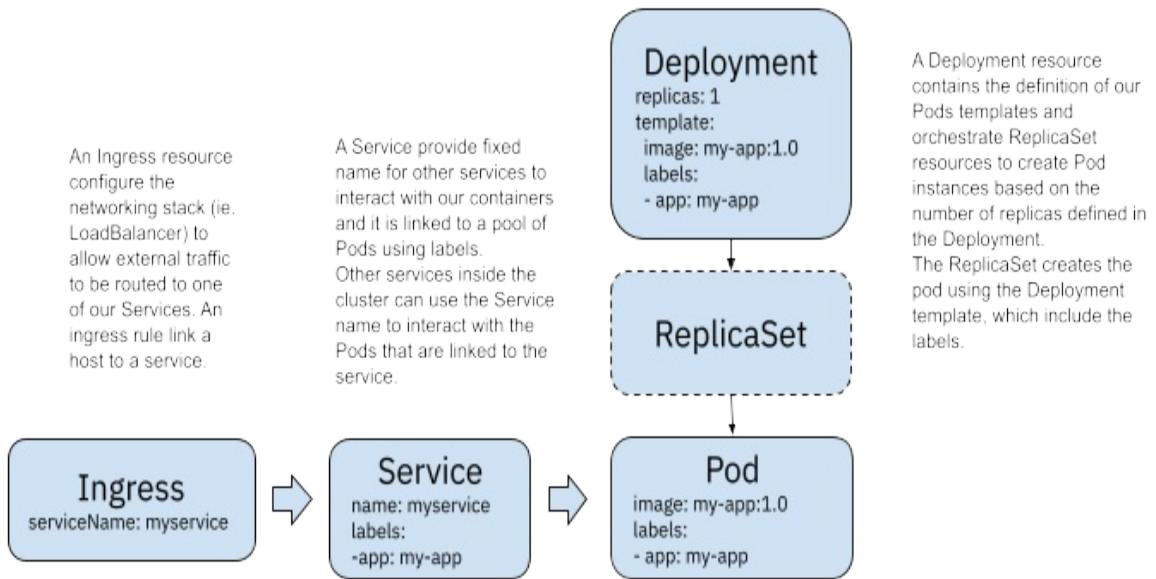
7.1 Kubernetes built-in mechanisms for releasing new versions

Kubernetes comes with built-in mechanisms ready to deploy and upgrade your services. Deployment and StatefulSets resources orchestrate ReplicaSets to run a built-in rolling update mechanism when the resource's configurations change. Deployments and StatefulSets keep track of the changes between one version and the next, allowing rollbacks to previously recorded versions.

Deployments and StatefulSets are like cookie cutters; they define how the container(s) for our service(s) needs to be configured and, based on the replicas specified, will create that amount of containers. We will need to create a Service to route traffic to these containers, as explained in Chapter 2.

Using a Service for each Deployment is standard practice, and it is enough to enable different services to talk to each other by using a well-known Service name. But if we want to allow external users (outside our Kubernetes Cluster) to interact with our Services, we will need an Ingress resource (plus an ingress controller). The Ingress resource will be in charge of configuring the networking infrastructure to enable external traffic into our Kubernetes cluster; this is usually done for a handful of services. In general, not every service is exposed to external users.

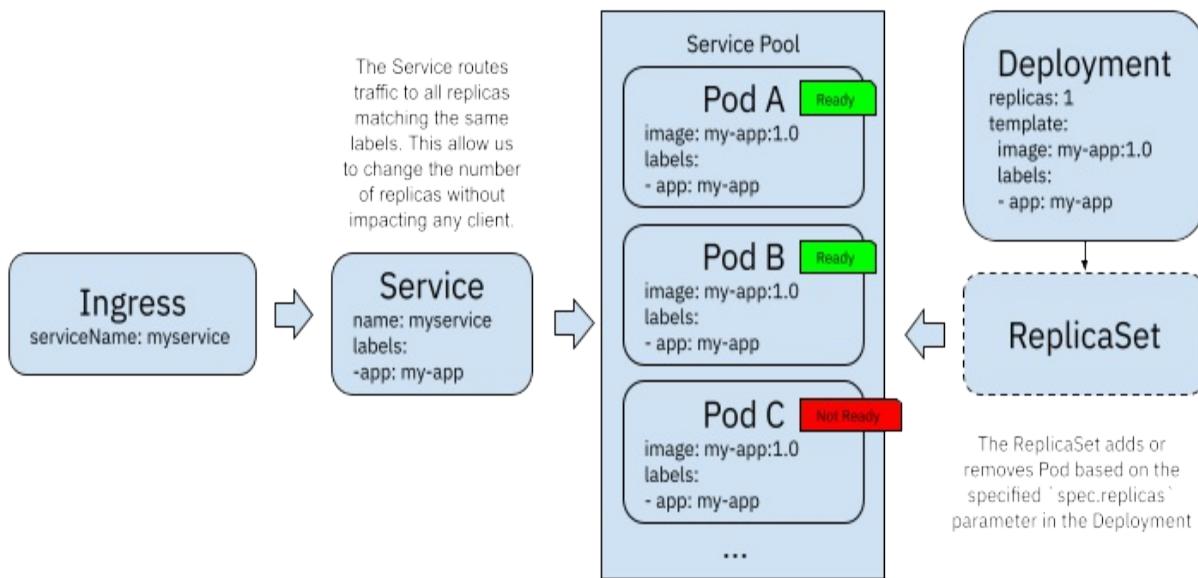
Figure 7.1 Kubernetes built-in resources for routing traffic



Now, imagine what happens when you want to update the version of one of these external-facing services. Commonly, these services are user interfaces. And it will be pretty good to upgrade the service without downtime. The good news is that Kubernetes was designed with zero-downtime upgrades in mind, so both Deployments and StatefulSets come equipped with a rolling update mechanism.

If you have multiple replicas of your application Pods, the Kubernetes Service resource acts as a load balancer. This allows other services inside your cluster to use the Service name without caring which replica they interact with (or which IP address each replica has).

Figure 7.2 Kubernetes Service acting as a load balancer



To attach each of these replicas to the load balancer (Kubernetes Service) pool, Kubernetes uses a probe called “Readiness Probe” (<https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>) to make sure that the container running inside the Pod is ready to accept incoming requests, in other words, it has finished bootstrapping. In Figure 7.2, Pod C is not ready yet. Hence it is not attached to the Service pool, and no request has been forwarded to this instance yet.

Now, if we want to upgrade to using the `my-app:1.1` container image, we need to perform some very detailed orchestration of these containers.

Suppose we want to ensure we don't lose any incoming traffic while doing the update. We need to start a new replica using the `my-app:1.1` image and ensure that this new instance is up and running and ready to receive traffic before removing the old version. If we have multiple replicas, we probably don't want to start all the new replicas simultaneously, as this will cause doubling up all the resources required to run this service.

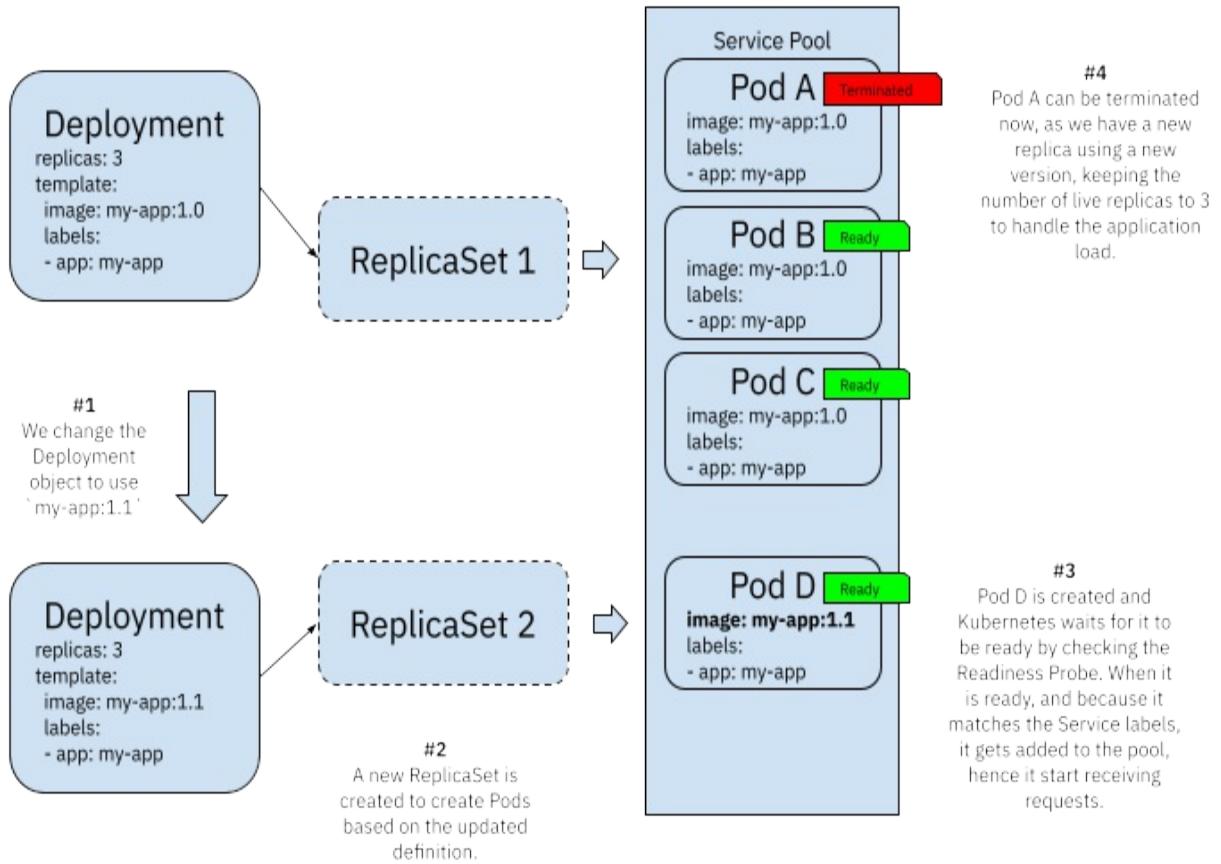
We also don't want to stop the old `my-app:1.0` replicas in one go. We need to guarantee that the new version is working and handling the load correctly before we shut down the previous version that was working fine. Luckily for us, Kubernetes automates all the starting and stopping of containers using a rolling update strategy (<https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/>).

7.1.1 Rolling updates

Deployments and StatefulSets come with these mechanisms built-in, and we need to understand how these mechanisms work to know when to rely on them and their limitations and challenges.

A rolling update consists of well-defined checks and actions to upgrade any number of replicas managed by a Deployment. Deployment resources orchestrate ReplicaSets to achieve rolling updates, and the following figure shows how this mechanism works:

Figure 7.3 Kubernetes Deployments rolling updates



The rolling update mechanism kicks off by default whenever you update a Deployment resource. A new ReplicaSet is created to handle the creation of Pods using the newly updated configuration defined in `spec.template`. This new ReplicaSet will not start all three replicas immediately, as this will cause a surge in resource consumption. Hence it will create a single replica, validate that it is ready, attach it to the service pool, and terminate a replica with the old configuration.

By doing this, the Deployment object guarantees three replicas active at all times handling clients' requests. Once Pod D is up and running, and Pod A is terminated, ReplicaSet 2 can create Pod E, wait for it to be ready, and then terminate Pod B. This process repeats until all Pods in ReplicaSet 1 are drained and replaced with new versions managed by ReplicaSet 2. If you change the Deployment resource again, a new ReplicaSet (ReplicaSet 3) will be created, and the process will repeat again.

A benefit of rolling updates is that ReplicaSets contain all the information

needed to create Pods for a specific Deployment configuration. If something goes wrong with the new container image (in this example, `my-app:1.1`) we can easily revert (rollback) to a previous version. You can configure Kubernetes to keep a certain number of revisions (changes in the Deployment configuration), so changes can be rolled back or forward.

Changing a Deployment object will trigger the rolling update mechanism, and you can check some of the parameters that you can configure to the default behavior here

(<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>).

StatefulSets have a different behavior as the responsibility for each replica is related to the state that is handled. The default rolling update mechanism works a bit differently. You can find the differences and a detailed explanation of how this works here

(<https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/>).

Check out the following commands to review the Deployment revisions history and do rollbacks to previous versions

(<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#check-rollout-history-of-a-deployment>):

```
kubectl rollout history deployment/frontend  
kubectl rollout undo deployment/frontend --to-revision=2
```

If you haven't played around with Kubernetes, I strongly recommend you create a simple example and try these mechanisms out. There are many online and interactive examples and tutorials where you can see this in action.

7.1.2 Canary Releases

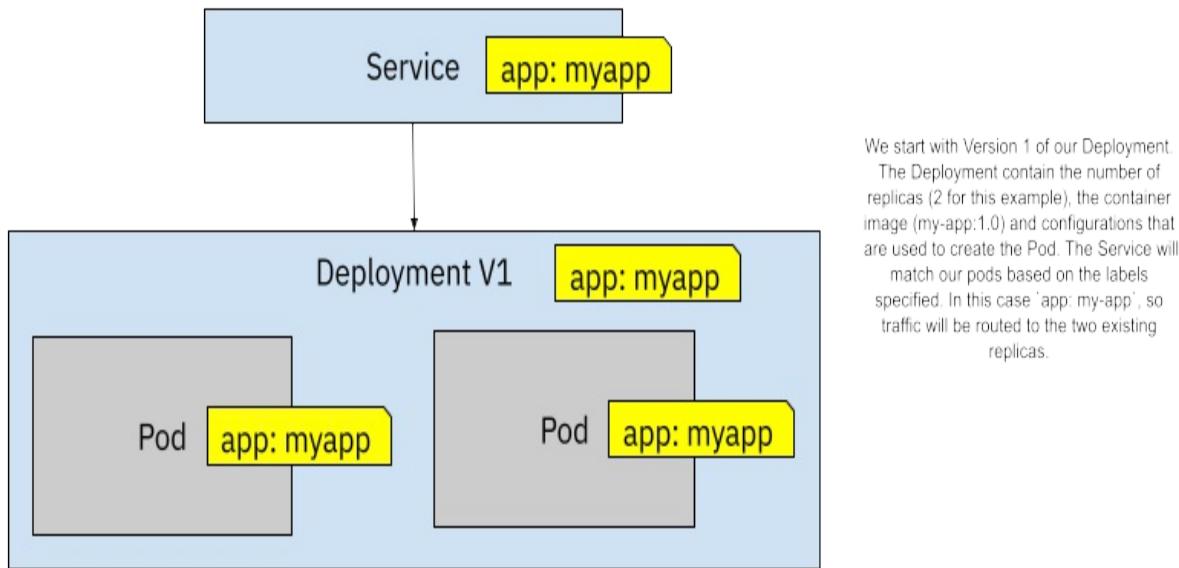
Rolling updates kick in automatically and are performed as soon as possible if we are using Deployments; what happens when we want more control over when and how we roll out new versions of our services?

Canary Releases (<https://martinfowler.com/bliki/CanaryRelease.html>) is a technique used to test if a new version is behaving as expected before pushing it live in front of all our live traffic.

While rolling updates will check that the new replicas of the service are ready to receive traffic, Kubernetes will not check that the new version is not failing to do what it is supposed to. Kubernetes will not check that the latest versions perform the same or better than the previous. Hence we can introduce issues to our applications. More control over how these updates are done is needed.

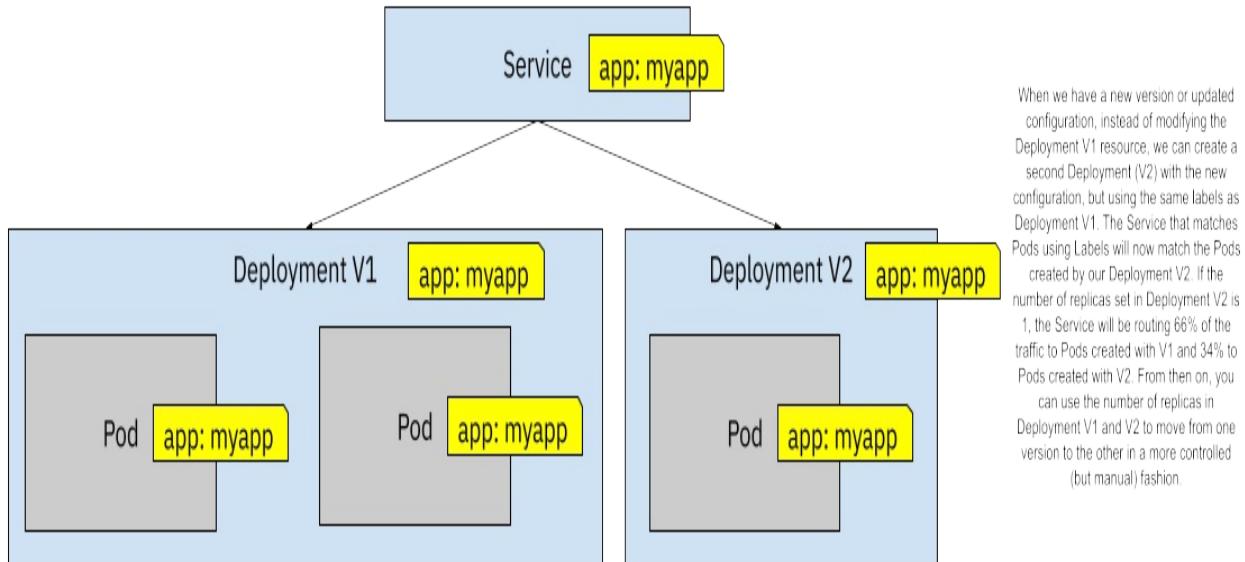
If we start with a Deployment configured to use a Docker image called `my-app:1.0`, have two replicas, and label it with `app: myapp`, a Service will route traffic as soon as we use the selector `app:myapp`. Kubernetes will be in charge of matching the Service selectors to our Pods. In this scenario, 100% of the traffic will be routed to both replicas using the same Docker image (`my-app:1.0`).

Figure 7.4 Kubernetes Service routes traffic to two replicas by matching labels



Imagine changing the configuration or having a new Docker image version (maybe `my-app:1.1`). We don't want to automatically migrate our Deployment V1 to the new version of our Docker image. Alternatively, we can create a second Deployment (V2) resource and leverage the service `selector` to route traffic to the new version.

Figure 7.5 One Service and two Deployments sharing the same labels



By creating a second Deployment (using the same labels), we are routing traffic to both versions simultaneously, and how much traffic goes to each version is defined by the number of replicas configured for each Deployment. By starting more replicas on *Deployment V1* than *Deployment V2* you can control what percentage of the traffic will be routed to each version. Figure 7.5 shows a 66%/34% (2 to 1 pods) traffic split between V1 and V2. Then you can decrease the number of replicas for Deployment V1 and increase the replicas for V2 to move towards V2 slowly. Notice that you don't have a fine grain control about which requests go to which version—the Service forwards traffic to all the matched pods in a round-robin fashion.

Because we have replicas ready to receive traffic at all times, there shouldn't be any downtime of our services when we do Canary Releases.

A significant point about rolling updates and Canary Releases is that they depend on our services to support traffic being forwarded to different versions simultaneously without breaking. This usually means that we cannot have breaking changes from version 1.0 to version 1.1 that will cause the application (or the service consumers) to crash when switching from one version to another. Teams making the changes must be aware of this restriction when using Rolling Updates and Canary Releases, as traffic will be forwarded to both versions simultaneously. For cases when two different versions can not be used simultaneously, and we need to have a hard switch

between version 1.0 and version 1.1, we can look at Blue/Green Deployments.

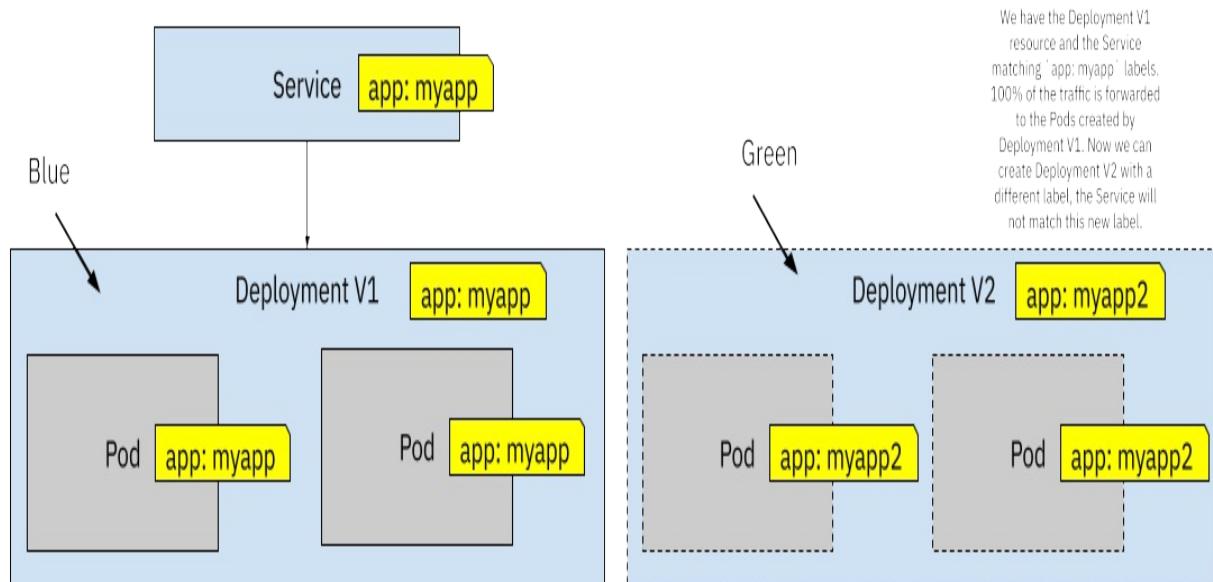
7.1.3 Blue/Green Deployments

Whenever you face a situation where you can not upgrade from one version to the next and have users/clients consuming both versions simultaneously, you need a different approach. Canary Deployments or Rolling Updates, as explained in the previous section, will just not work. If you have breaking changes, you might want to try Blue/Green deployments (<https://martinfowler.com/bliki/BlueGreenDeployment.html>).

Blue/Green deployments help us move from version 1.0 to version 1.1 at a fixed point in time, changing how the traffic is routed to the new version without allowing the two versions to receive requests simultaneously and without downtime.

Blue/Green Deployments can be implemented using built-in Kubernetes Resources by creating two different Deployments, as shown in Figure 7.6.

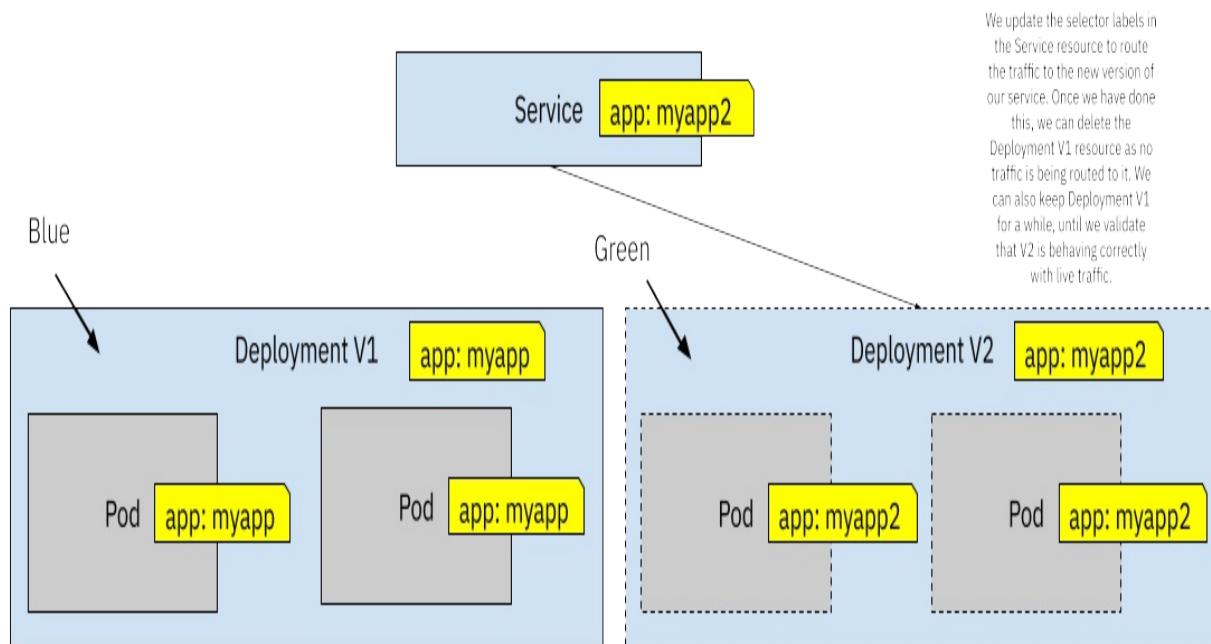
Figure 7.6 Two Deployments using different Labels



In the same way, as we did with Canary Releases, we start with a Service and

a Deployment. The first version of these resources is what we call “Blue”. For Blue/Green Deployments, we can create a separate Deployment (V2) resource to start and prepare for our service’s new version when we have a new version. This new Deployment needs different labels for the Pods it will create, so the Service doesn’t match these Pods yet. We can connect to Deployment V2 pods by using `kubectl port-forward` or running other in-cluster tests until we are satisfied that this new version is working. When we are happy with our testing, we can switch from Blue to Green by updating the `selector` labels defined in the Service resource.

Figure 7.7 When the new version is ready we switch the label from the Service selector to match version V2 labels



Blue/Green Deployments make a lot of sense when we cannot send traffic to both versions simultaneously, but it has the drawback of requiring both versions to be up simultaneously to switch traffic from one to the other. When we do Blue/Green deployments, the same amount of replicas is recommended for the new version. We need to ensure that when traffic is redirected to the new version, this version is ready to handle the same load as the old version.

When we switch the label selector in the Service resource, 100% of the traffic

is routed to the new version, and Deployment V1 pods stop receiving traffic. This is quite an important detail as if some state was being kept in V1 Pods, you will need to drain the state from the Pods, migrate and make this state available for V2 Pods. In other words, if your application holds a state in memory, you should write that state to persistent storage so V2 Pods can access it and resume whatever work was done with that data. Remember that most of these mechanisms were designed for stateless workloads, so you need to make sure that you follow these principles to make sure that things work as smoothly as possible.

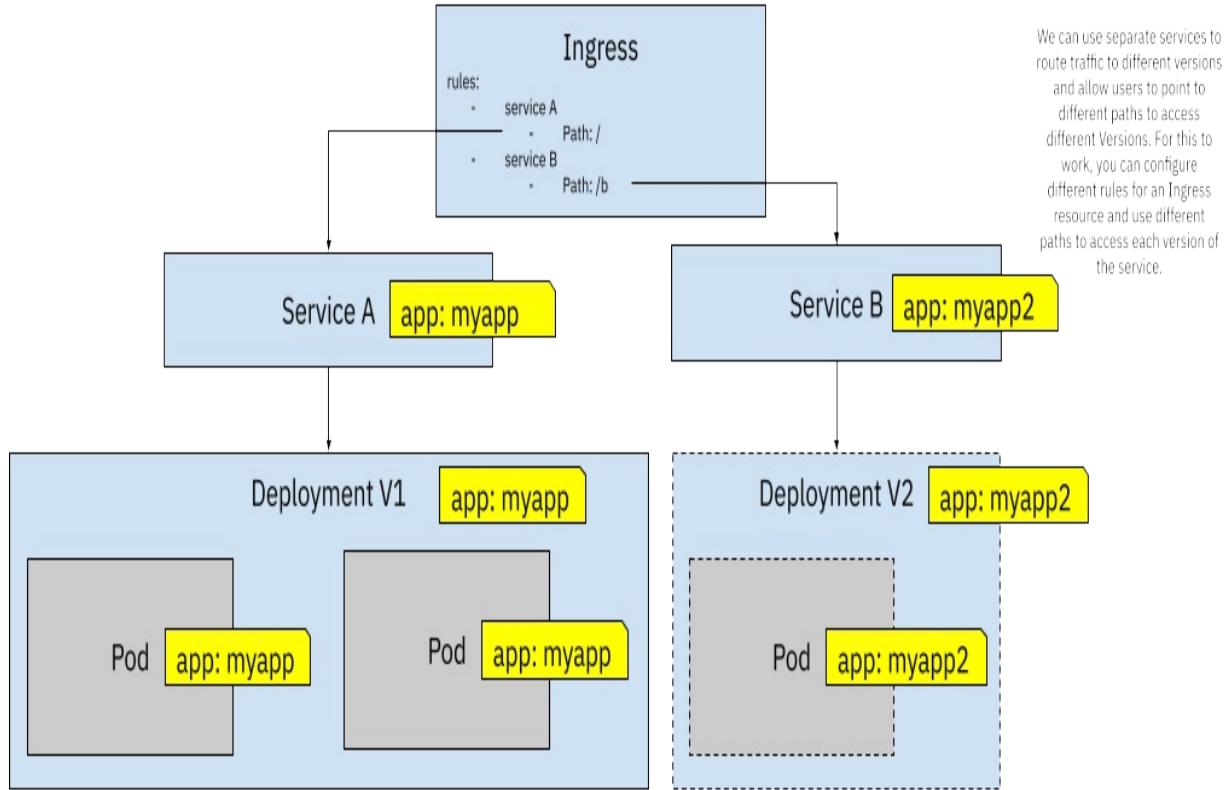
For Blue/Green deployments, we are interested in moving from one version to the next at a given time, but what about scenarios when we want to actively test two or more versions with our users simultaneously to decide which one performs better? Let's take a look at A/B testing next.

7.1.4 A/B testing

It is quite a common requirement to have two versions of your services running simultaneously, and we want to test these two versions to see which one performs better. (<https://blog.christianposta.com/deploy/blue-green-deployments-a-b-testing-and-canary-releases/>). Sometimes you want to try a new user interface theme, place some UI elements in different positions or new features added to your app, and gather user feedback to decide which works best.

To implement this kind of scenario in Kubernetes, you need two different Services pointing to two different Deployments. Each Deployment handles just one version of the application/service. If you want to expose these two versions outside the cluster, you can define an Ingress resource with two rules. Each rule will be in charge of defining the external path or subdomain to access each service.

Figure 7.8 Using two Ingress rules for pointing to A and B versions



If you have your application hosted under the www.example.com domain, the Ingress resource defined in Figure 7.8 will direct traffic to Service A, allowing users to point their browsers to www.example.com/b to access Service B. Alternatively, and depending on how you have configured your Ingress Controller, you can also use subdomains instead of path-based routing meaning that to access the default version you can keep using www.example.com but to access Service B you can use a subdomain such as test.example.com

I can hear you saying, what a pain. Look at all the Kubernetes resources I need to define and maintain to achieve something that feels basic and needed for everyday operations. And if things are complicated, teams will not use them. Let's quickly summarize the limitations and challenges we have found so far, so we can look at Knative Serving and how it can help us to streamline these strategies.

7.1.5 Limitations and complexities of using Kubernetes built-in building blocks

Canary Releases, Blue/Green deployments, and A/B testing can be implemented using built-in Kubernetes resources. But as you have seen in the previous sections, creating different deployments, changing labels, and calculating the number of replicas needed to achieve percentage-based distribution of the requests is quite a major and error-prone task. Even if you use a GitOps approach, as shown with ArgoCD or with other tools in Chapter 4, creating the required resources with the right configurations is quite hard, and it takes a lot of effort.

We can summarize the drawbacks of implementing these patterns using Kubernetes building blocks as follows:

- Manual creation of more Kubernetes resources, such as Deployments, Services, and Ingress Rules to implement these different strategies can be error-prone and cumbersome. The team implementing the release strategies must understand how Kubernetes behaves to achieve the desired output.
- No automated mechanisms are provided out-of-the-box to coordinate and implement the resources required by each release strategy.
- Error-prone, as multiple changes need to be applied at the same time in different resources for everything to work as expected
- Suppose we notice a demand increase or decrease in our services. In that case, we need to manually change the number of replicas for our Deployments or install and configure a custom auto scaler (more on this at the end of this chapter). Unfortunately, if you set the number of replicas to 0, there will not be any instance to answer requests, requiring you to have at least one replica running all the time.

Out of the box, Kubernetes doesn't include any mechanism to automate or facilitate these release strategies, which becomes a problem quite quickly if you are dealing with many services that depend on each other.

One thing is clear, your teams need to be aware of the implicit contracts imposed by Kubernetes regarding 12-factor apps and how their services APIs evolve to avoid downtime. Your developers need to know how Kubernetes' built-in mechanisms work to have more control over how your applications are upgraded.

If we want to reduce the risk of releasing new versions, we want to empower our developers to have these release strategies available for their daily experimentation.

In the next sections, we will look at Knative Serving and Argo Rollouts, a set of tools and mechanisms built on top of Kubernetes to simplify all the manual work and limitations that we will find when trying to set up Kubernetes building blocks to enable teams with different release mechanisms.

Let's start first with Knative Serving that provides us with a flexible set of features to easily implement the release strategies described before.

7.2 Knative Serving: advanced traffic management and release strategies

Knative is one of these technologies that are hard to go back to and not use when you learn about what it can do for you. After working with the project for almost 3 years and observing the evolution of some of its components, I can confidently say that every Kubernetes Cluster should have Knative Serving installed; your developers will appreciate it. Knative provides higher-level abstractions on top of Kubernetes built-in resources to implement good practices and common patterns that enable your teams to go faster, have more control over their services, and facilitate the implementation of Event-Driven applications.

As the title of this section specifies, the following sections focus on a subset of the functionality provided by Knative, called Knative Serving. Knative Serving allows you to define "*Knative Services*", which dramatically simplifies implementing the release strategies exemplified in the previous sections. Knative Services will create Kubernetes built-in resources for you and keep track of their changes and versions, enabling scenarios that require multiple versions to be present simultaneously. Knative Services also provides advanced traffic handling and autoscaling to scale down to zero replicas for a serverless approach.

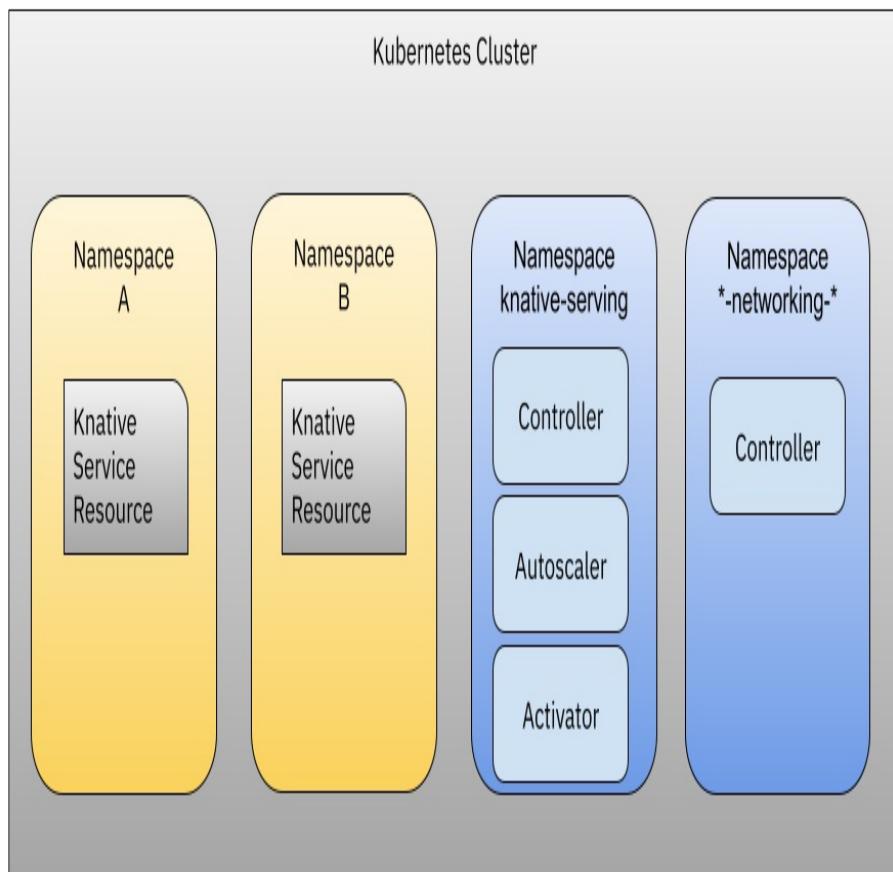
To install Knative Serving on your cluster, I highly recommend you to check the official Knative documentation that you can find [here](#):

<https://knative.dev/docs/admin/install/serving/install-serving-with-yaml/>

If you want to install Knative locally, you should the getting started guide for local development (<https://knative.dev/docs/getting-started/#install-the-knative-quickstart-environment>). Optionally, you can install `kn`, a small binary that allows you to interact with Knative more naturally. You can always use `kubectl` to interact with Knative resources.

When installing Knative Serving, you are installing a set of components in the `knative-serving` namespace that understand how to work Knative Serving custom resource definitions.

Figure 7.9 Knative Serving components in your Kubernetes Cluster



Knative Serving consist on a set of components are installed in a new namespace called 'knative-serving' and they rely on a networking stack to be installed to support advanced traffic policies. You can choose from a list of different networking stacks including Istio, Courier, Contour and others. Once you have Knative Serving installed you can start creating Knative Services in your application namespaces and the Knative Serving Controller will coordinate with the Networking Controller what configurations needs to be changes in the networking layer to route traffic to your Services. The Knative Serving Autoscaler and the Activator are described later in this chapter.

It is outside of the scope of this book to explain how Knative Serving components and resources work; my recommendation is that if I manage to

get your attention with the examples in the following sections, you should check out the Knative In Action book by Jacques Chester (<https://www.manning.com/books/knative-in-action>).

7.2.1 Knative Services

Once you have Knative Serving installed, you can create *Knative Services*. I can hear you thinking: “But we already have Kubernetes Services, why do we need Knative Services?” believe me, I had the same feeling when I saw the same name but follow along, it does make sense.

When deploying each Service in the Conference Application before, you needed to create at least two Kubernetes resources, a Deployment, and a Service. We have covered in a previous section that using ReplicaSets internally, a Deployment can perform rolling updates by keeping track of the configuration changes in the Deployment resources and creating new ReplicaSets for each change we make. We also discussed in Chapter 2 the need for creating an Ingress resource if we want to route traffic from outside the cluster. Usually, you will only create an Ingress resource to map the publicly available services, such as the User Interface of the Conference Platform.

Knative Services build on top of these resources and simplify how we define these resources that you need to create for every application service. While it simplifies the task and reduces the amount of YAML that we need to maintain, it also adds some exciting features, but before jumping into the features, let’s take a look at how a Knative Service looks in action. Let’s start simple and use the Email Service from the Conference Platform to demonstrate how Knative Services work:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: email-service #A
spec:
  template:
    spec:
      containers:
        - image: salaboy/fmtok8s-email-rest:0.1.0 #B
```

In the same way, as a Deployment will pick the `spec.template.spec` field to cookie-cut Pods, a Knative Service defines the configuration for creating other resources using the same field.

Nothing too strange so far, but how is this different from a Kubernetes Service?

If you create this resource using `kubectl apply -f` you can start exploring the differences.

You can also list all Knative Services using `kubectl get ksvc` (ksvc stands for Knative Service), and you should see your newly created Knative Service there:

```
NAME          URL
email-service  http://email-service.default.X.X.X.X.sslip.io  ema
```

There are a couple of details to notice right here; first of all, there is a URL that you can copy into your browser and access the service. If you were running in a Cloud Provider and configured DNS while installing Knative, this URL should be accessible immediately. The LASTCREATED column shows the name of the latest Knative Revision of the Service. Knative Revisions are pointers to the specific configuration of our Service, meaning that we can route traffic to them.

You can go ahead and test the Knative Service URL by using `curl` or by pointing your browser to <http://email-service.default.X.X.X.X.sslip.io/info>.

You should see the following output:

```
{"name": "Email Service (REST)", "version": "v0.1.0", "source": "https
```

As with any other Kubernetes resource, you can also use `kubectl describe ksvc email-service` to get a more detailed description of the resource.

If you list other well-known resources such as Deployment, Services, and Pods you will find out that Knative Serving is creating them for you and managing them. Because these are managed resources now, it is usually not recommended to change them manually. If you want to change your

application configurations, you should edit the Knative Service resource.

A Knative Service, as we applied it before to our cluster, by default behaves differently from creating a Service, a Deployment, and an Ingress manually.

A Knative Service by default:

- **It is accessible:** Expose itself under a public URL so you can access it from outside the cluster. It doesn't create an Ingress resource, as it uses the available Knative Networking stack that you installed previously. Because Knative has more control over the network stack and manages Deployments and Services, it knows when the service is ready to serve requests, reducing configuration errors between Services and Deployments.
- **Manages Kubernetes resources:** It creates two Services and a Deployment: Knative Serving allows us to run multiple versions of the same service simultaneously. Hence it will create a new Kubernetes Service for each version (which in Knative Serving are called Revisions)
- **Collects Service usage:** It creates a Pod with the specified `user-container` and a sidecar container called `queue-proxy`.
- **Scale-up and down based on demand:** Automatically downscale itself to zero if no requests are hitting the service (by default after 90 seconds)
 - It achieves this by downscaling the Deployment replicas to 0 using the data collected by the `queue-proxy`
 - If a request arrives and there is no replica available, it scales up while queuing the request, so it doesn't get lost.
- **Configuration changes history is managed by Knative Serving:** If you change the Knative Service configuration, a new *Revision* will be created. By default, all traffic will be routed to the latest revision.

Of course, these are the defaults, but you can fine-tune each of your Knative Services to serve your purpose and, for example, implement the previously described release strategies.

In the next section, we will look at how Knative Serving advanced traffic-handling features can be used to implement Canary Releases, Blue/Green Deployments, A/B testing, and Header-based routing.

7.2.2 Advanced traffic-splitting features

Let's start by looking at how you can implement a Canary Release for one of our application's services with a Knative Service.

This section starts by looking into doing Canary Releases using percentage-based traffic splitting. Then it goes into A/B testing by using tag-based and header-based traffic splitting.

Canary releases using percentage-based traffic splitting

If you get the Knative Service resource (with `kubectl get ksvc email-service -oyaml`), you will notice that the `spec` section now also contains a `spec.traffic` section that was created by default, as we didn't specify anything.

```
traffic:  
  - latestRevision: true  
    percent: 100
```

By default, 100% of the traffic is being routed to the latest Knative Revision of the service.

Now imagine that you made a change in your service to improve how emails are sent, but your team is not sure how well it will be received by people and we want to avoid having any backlash from people not wanting to sign in to our conference because of the website. Hence we can run both versions side by side and control how much of the traffic is being routed to each version (Revision in Knative terms).

Let's edit (`kubectl edit ksvc email-service`) the Knative Service and apply the changes:

```
apiVersion: serving.knative.dev/v1  
kind: Service  
metadata:  
  name: email-service  
spec:  
  template:  
    spec:
```

```

  containers:
    - image: salaboy/fmtok8s-email-rest:0.1.0-improved #A
  traffic: #B
  - percent: 80
    revisionName: email-service-00001
  - latestRevision: true
    percent: 20

```

If you try now with `curl` you should be able to see the traffic split in action:

```

<pre class="codeacxspfirst">salaboy>&nbsp;curl&nbsp;http://ema
</pre> <pre class="codeacxspmiddle">{"name":"Email
</pre> <pre class="codeacxspmiddle">"source":<a href="http://email-service-00001">Email</a>
</pre> <pre class="codeacxspmiddle">salaboy>&nbsp;curl&nbsp;http://email-service-00001
</pre> <pre class="codeacxspmiddle">{"name":"Email
</pre> <pre class="codeacxspmiddle">"source":<a href="http://email-service-00001">Email</a>
</pre> <pre class="codeacxspmiddle">salaboy>&nbsp;curl&nbsp;http://email-service-00001
</pre> <pre class="codeacxspmiddle">{"name":"Email
</pre> <pre class="codeacxspmiddle">"source":<a href="http://email-service-00001">Email</a>
</pre> <pre class="codeacxspmiddle">salaboy>&nbsp;curl&nbsp;http://email-service-00001
</pre> <pre class="codeacxspmiddle">{"name":"Email
</pre> <pre class="codeacxspmiddle">"source":<a href="http://email-service-00001">Email</a>
</pre> <pre class="codeacxspmiddle">salaboy>&nbsp;curl&nbsp;http://email-service-00001
</pre> <pre class="codeacxspmiddle">{"name":"Email
</pre> <pre class="codeacxspmiddle">"source":<a href="http://email-service-00001">Email</a>
</pre> <pre class="codeacxsplast">&nbsp;"source":<a href="http://email-service-00001">Email</a>
</pre>

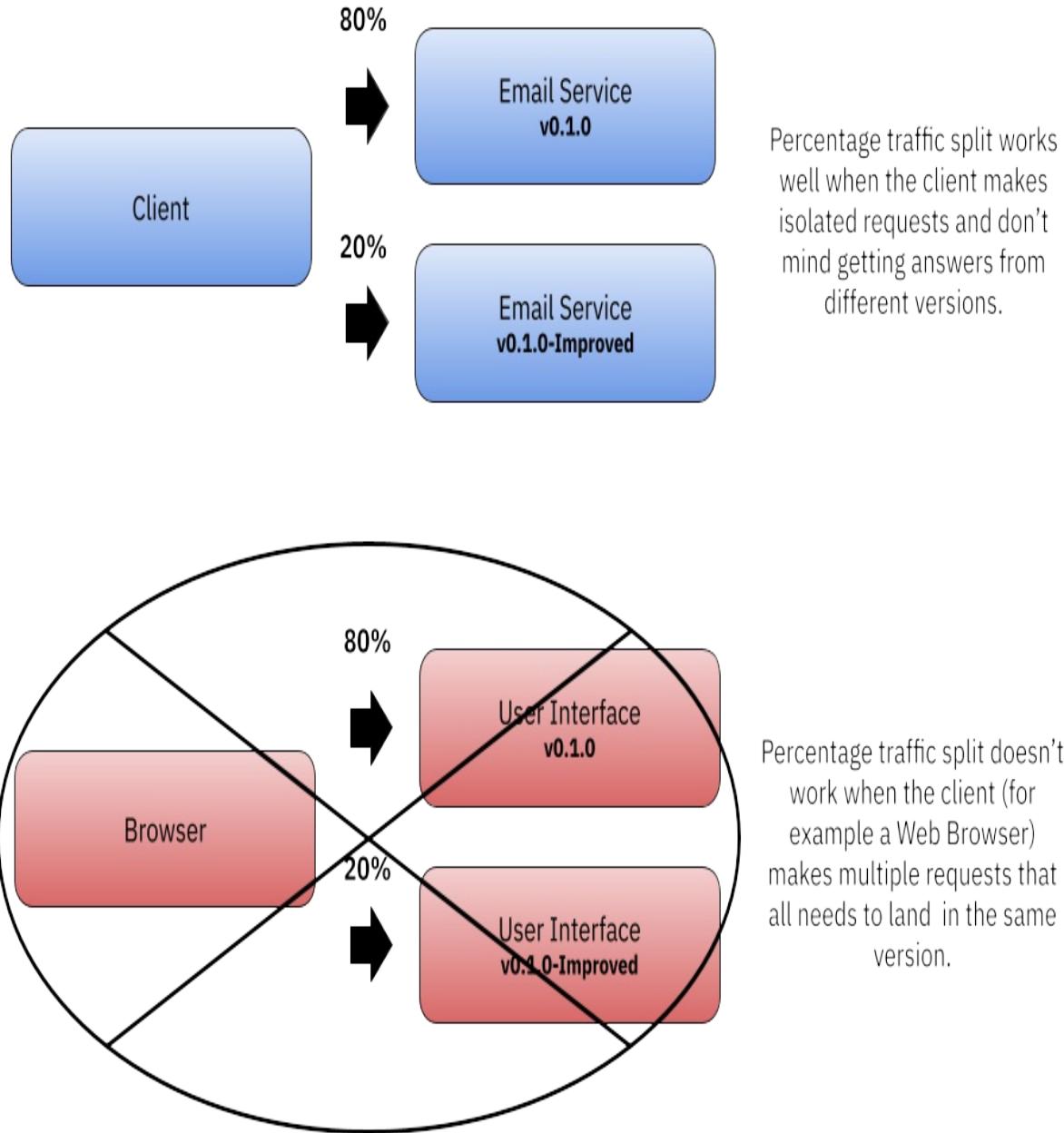
```

Once you have validated that the new version of your service is working correctly, you can start sending more traffic until you feel confident to move 100% of the traffic to it. If things go wrong, revert the traffic split to the stable version.

Notice that you are not limited to just two service revisions; you can create as many as you want as long as the traffic percentage is 100%. Knative will follow these rules and scale up the required revisions of your services to serve requests. You don't need to create any new Kubernetes resources, as Knative will create those for you, reducing the likelihood of errors that come with modifying multiple resources simultaneously.

By using percentages, you don't have control over where subsequent requests will land. Knative will just make sure to maintain a fair distribution based on the percentages that you have specified. This can become a problem if, for example, you have a User Interface instead of a simple REST endpoint.

Figure 7.10 Percentage based traffic split scenarios and challenges



User Interfaces are complex because a browser will perform several GET requests to render the page HTML, CSS and images, and so forth. You can quickly end up in a situation where each request hits a different version of your application. Let's look at a different approach that might be better suited for testing User Interfaces or scenarios when we need to ensure that several

requests end up in the correct version of our application.

A/B testing with tag-based routing

If you want to perform A/B testing for a User Interface, you will need to give Knative some way to differentiate where to send the requests. You have two options. First, you can point to a special URL for the service you want to try out, and the second is to use a request header to differentiate where to send the request. Let's look at these two alternatives in action.

But let's now use the User Interface for the Conference Platform:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: ui-service #A
spec:
  template:
    spec:
      containers:
        - image: salaboy/fmtok8s-api-gateway:0.1.0 #B
```

Once again, we have just created a Knative Service, but we cannot specify percentage-based routing rules because this container image contains a Web Application. Knative will not stop you from doing so. Still, you will notice requests going to different versions and errors popping up because a given image is not in one of the versions, or you end up with the wrong stylesheet (CSS) coming from the wrong version of the application.

Let's start by defining a Tag that you can use to test a new stylesheet. You can do that by modifying the Knative Service resource as we did before, first change the image to `salaboy/fmtok8s-api-gateway:0.1.0-color` and then let's create some new traffic rules using tags:

```
traffic:
  - percent: 100 #A
    revisionName: ui-service-00001
  - latestRevision: true
    tag: color #B
```

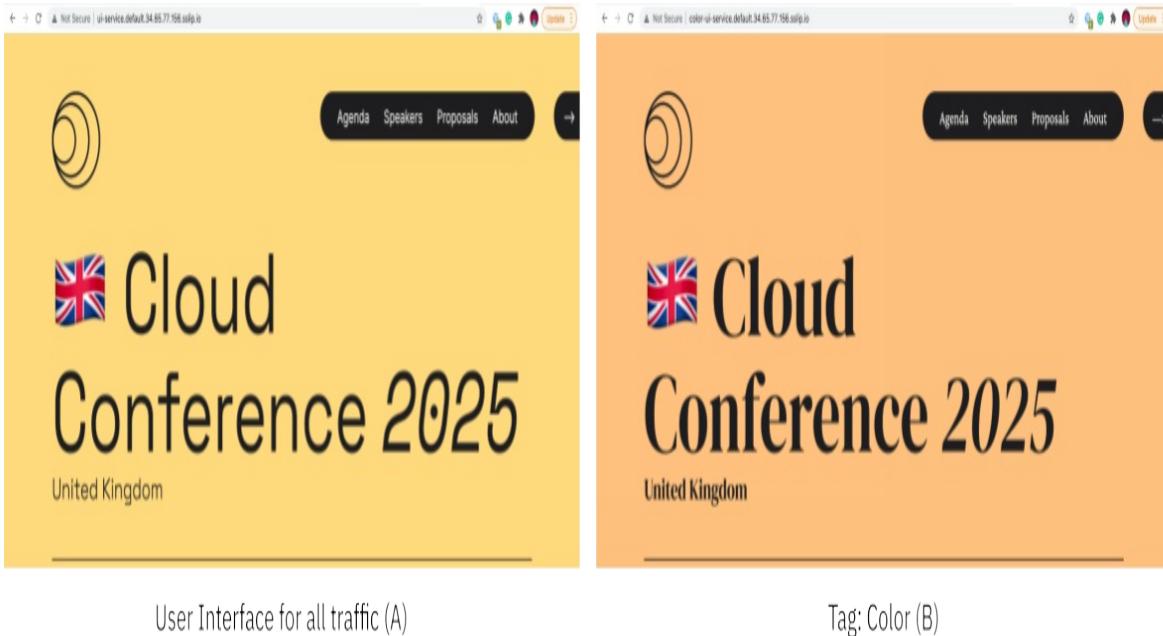
If you describe the Knative Service (`kubectl describe ksvc ui-service`)

you will find the URL for the Tag that we just created:

```
<pre class="codeacxspfirst">&nbsp;Traffic:  
</pre> <pre class="codeacxspmiddle">&nbsp;&nbsp;&nbsp;Lates  
</pre> <pre class="codeacxspmiddle">&nbsp;&nbsp;&nbsp;Perce  
</pre> <pre class="codeacxspmiddle">&nbsp;&nbsp;&nbsp;Revis  
</pre> <pre class="codeacxspmiddle">&nbsp;&nbsp;&nbsp;Lates  
</pre> <pre class="codeacxspmiddle">&nbsp;&nbsp;&nbsp;Perce  
</pre> <pre class="codeacxspmiddle">&nbsp;&nbsp;&nbsp;Revis  
</pre> <pre class="codeacxspmiddle">&nbsp;&nbsp;&nbsp;&nbsp;Tag:&  
</pre> <pre class="codeacxsplast">&nbsp;&nbsp;&nbsp;URL:&nb  
</pre>
```

Knative Serving had created a new URL (#A) by prepending the tag name to the service name. Check using the browser that can consistently access the modified version and the original one as well.

Figure 7.11 A/B testing with Tag-based routing



User Interface for all traffic (A)

Tag: Color (B)

Using tags guarantees that all requests are hitting the URL to the correct version of your service. One more option avoids you pointing to a different URL for doing A/B testing, and it might be useful for debugging purposes. The next section looks at tag-based routing using HTTP Headers instead of different URLs.

A/B testing with header-based routing

Finally, let's look at an experimental feature (at the time of writing) that allows you to use HTTP Headers to route requests. This experimental (at the time of writing) feature also uses tags to know where to route traffic, but instead of using a different URL to access a specific revision, you can add an HTTP Header that will do the trick.

Imagine that you want to enable developers to access a debugging version of the application. Application developers can set a special header in their browsers and then access a specific revision.

To enable this experimental feature, you or the administrator that installs Knative needs to patch a ConfigMap inside the `knative-serving` namespace:

```
kubectl patch cm config-features -n knative-serving -p '{"data":{}
```

Once the feature is enabled you can test this by changing again the Knative Service that we created in the previous section as follows:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: ui-service
spec:
  template:
    spec:
      containers:
        - image: salaboy/fmtok8s-api-gateway:0.1.0-debug #A
traffic:
  - percent: 100
    revisionName: ui-service-00001
  - revisionName: ui-service-00002 #B
    tag: color
  - latestRevision: true #C
    tag: debug
```

If you point your browser to the Knative Service URL (`kubectl get ksvc`) you will see the same application as always as shown in figure 7.12, but if you use a tool like ModHeader extension

(<https://chrome.google.com/webstore/detail/modheader/idgpnmonknjnojddfkplhl=en>) for Chrome you can set your custom HTTP Headers that will be included in every request that the browser produce. For this example, and because the tag that you created is called `debug` you need to set the following Http Header: `Knative-Serving-Tag: debug`. As soon as the HTTP Header is present Knative Serving will route the incoming request to the `debug` tag.

Figure 7.12 A/B testing with Http Header-based routing to Debug version

← → ⚡ Not Secure | ui-service.default:34.65.77.156.sslip.io

DEBUG

C4P Service N/A
Namespace: N/A
Id: N/A
Node: N/A

Agenda Service N/A
Namespace: N/A
Id: N/A
Node: N/A

Email Service N/A
Namespace: N/A
Id: N/A
Node: N/A

Agenda Speakers Proposals About

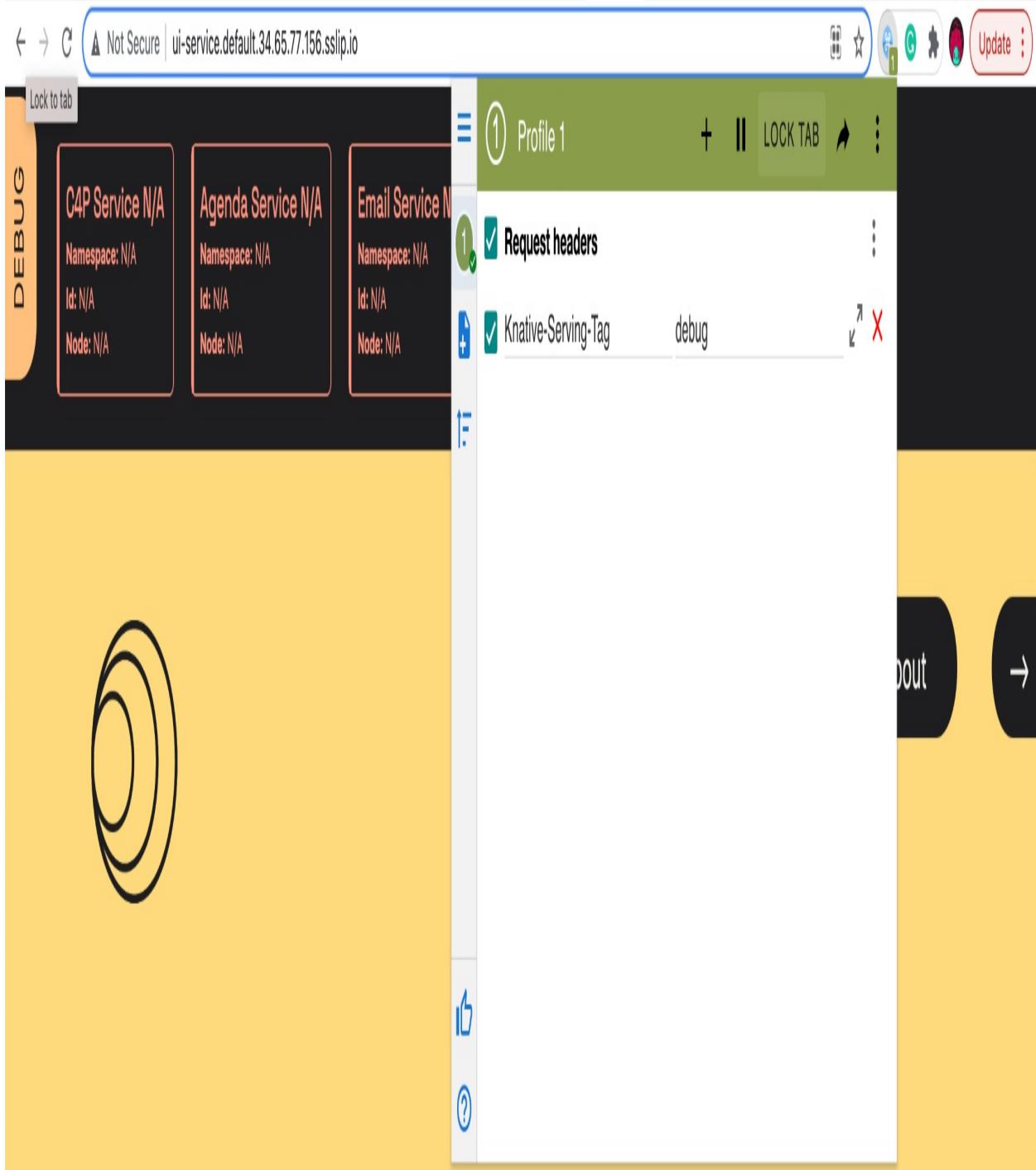
→



Cloud Conference 2025

United Kingdom

Figure 7.13 Using ModHeader Chrome extension to set custom HTTP Headers



Because the Header-based routing experimental feature also uses tags to route traffic. If you still have the `color` tag created, you can test traffic is being

routed by changing the `Knative-Serving-Tag` header value to `color` and see if the correct page comes back.

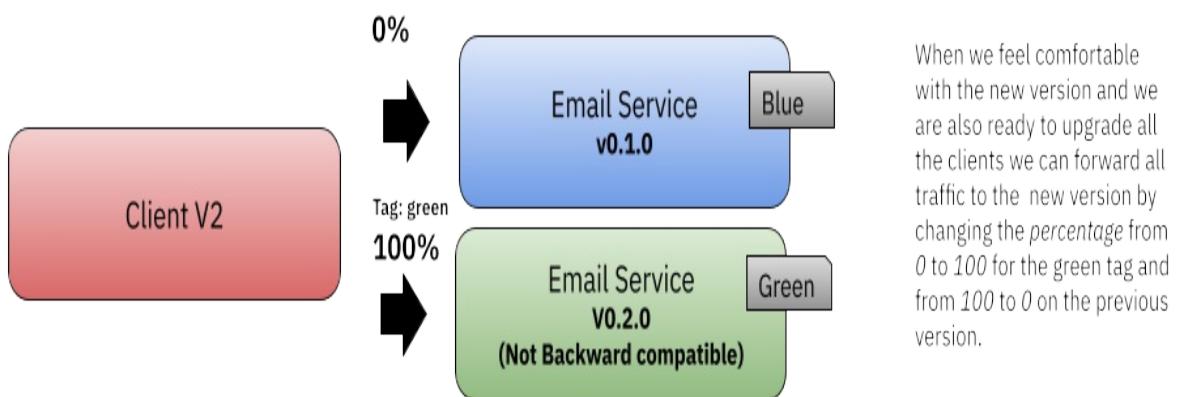
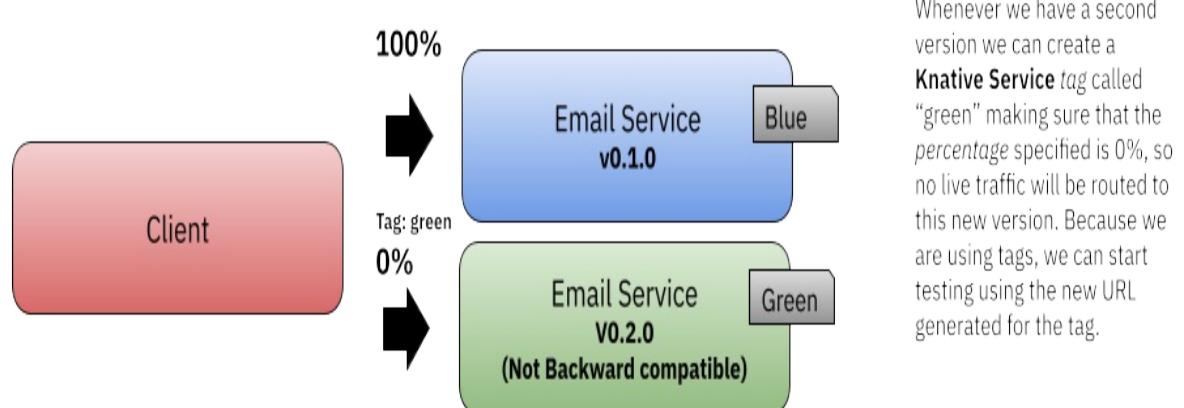
Both tag and header-based routing are designed to ensure that all requests will be routed to the same revision if you hit a specific URL (created for the tag) or if one particular Header is present. Finally, let's take a look at how to do Blue/Green Deployments with Knative Serving.

Blue/Green Deployments

For situations where we need to change from one version to the next at a very specific point in time because there is no backward compatibility, we can still use tag-based routing with percentages. Instead of going gradually from one version to the next, we use percentages as a switch from 0 to 100 on the new version and from 100 to 0 on the old version.

Most Blue/Green deployment scenarios require coordination between different teams and services to make sure that both the service and the clients are updated at the same time. Knative Serving allows you to declaratively define when to switch from one version to the next in a controller way.

Figure 7.14 Blue/Green Deployments using Knative Serving tag-based routing



To achieve this scenario described in figure 7.14, we can create the “green” tag for the new version inside our Knative Service as follows:

```
...
traffic:
  - revisionName: <first-revision-name>
    percent: 100 # All traffic is still being routed to the first
  - revisionName: <second-revision-name>
    percent: 0 # 0% of traffic routed to the second revision
    tag: green # A named route
```

By creating a new tag (called “green”) we will now have a new URL to access the new version for testing. This is particularly useful to test new versions of the clients, as if the Service API is changing with a non-backward compatible change, clients might need to be updated as well.

Once all tests are performed we can safely switch all traffic to the “green” version of our service.

```
...
traffic:
  - revisionName: <first-revision-name>
    percent: 0 # All traffic is still being routed to the first
  - revisionName: <second-revision-name>
    percent: 100 # 100% of traffic routed to the second revision
    tag: green # A named route
```

Generally, we cannot progressively move traffic from one version to the next, as the client consuming the service will need to understand that requests might land into different (and non-compatible) versions of the service.

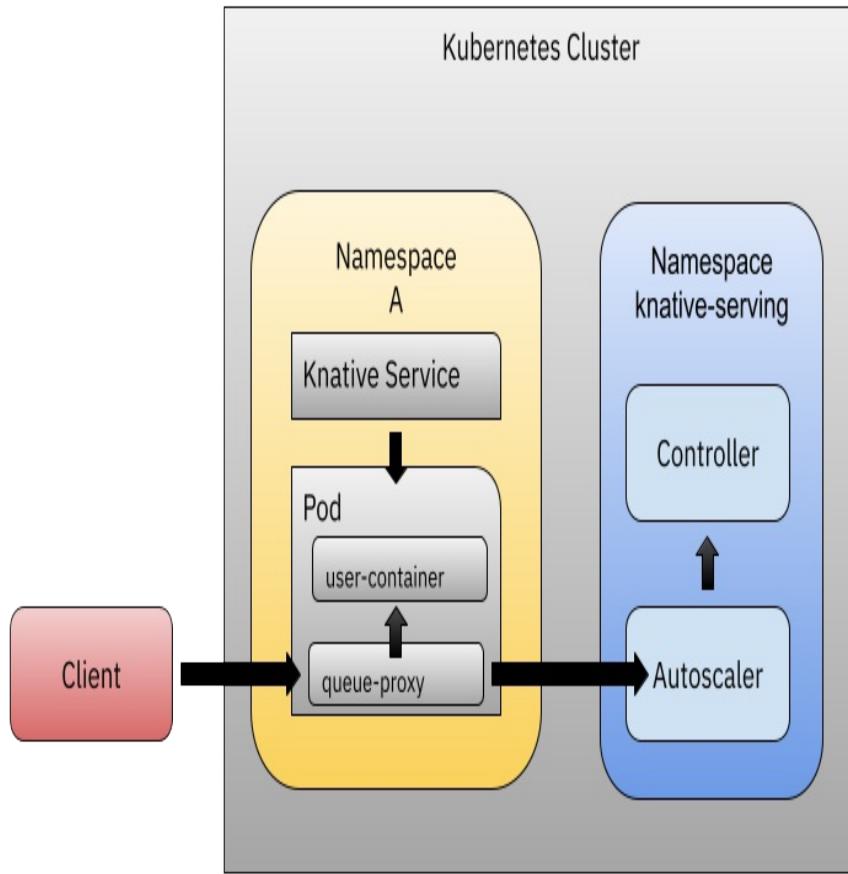
In the previous sections, we have been looking into how Knative Serving simplifies the implementation of different Release Strategies for your teams to continuously deliver features and new versions of your services. Knative Serving reduces the need to create several Kubernetes built-in resources to manually implement the release strategies described in this chapter. It does so by providing high-level abstractions such as Knative Services which create and manage Kubernetes built-in resources and a network stack for advanced traffic management.

In the final section of this chapter, we will take a look at the Knative Autoscaler, which adds to the benefits of using Knative.

7.2.3 The Knative Serving Autoscaler

An important part of Knative Serving is the Autoscaler, which allows your Knative Services by default to scale down to zero and handle the logic to queue requests if no replica is available to answer incoming requests until at least one replica is up.

Figure 7.15 Knative Serving Autoscaler and `queue-proxy`



When you create a Knative Service it will automatically add a sidcar container to your services. This sidcar container ('queue-proxy') is in charge of receiving all the incoming traffic and forward it to your application container. The 'queue-proxy' also gathers metrics and report back to the Autoscaler which can make decisions about upscaling or downscaling the number of replicas of your Knative Service. If the Autoscaler notice that no traffic is being received, it can notify the controller to trigger a scale down to zero operation.

The Knative Autoscaler is installed by default when you install Knative Serving and it continuously monitors your Knative Service requests. As figure 7.15 shows, in order to monitor all inbound traffic a sidcar container is attached to your application container and all traffic is routed via the “queue-proxy” container. The “queue-proxy” container collects metrics and informs the Autoscaler about how much traffic the application container (“user-container”) is receiving.

Based on this information, the Autoscaler can notify the Knative Service controller to scale up or down the replicas of your service. If no requests are being handled by the “user-container” for a while, the Autoscaler will notify the Knative Service controller to scale down to zero the number of replicas.

Scaling up and down works out of the box, but you can always fine-tune the

behavior based on your application capabilities. In general, to configure the Autoscaler you can annotate your Knative Services with the amount of concurrent requests that your application can handle per replica. This information will be used by the Autoscaler to decide when new replicas are needed.

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/target: "100" # 100 concurrent re
```

You can also change this for all Knative Services by changing a global parameter inside the `config-autoscaler` ConfigMap:

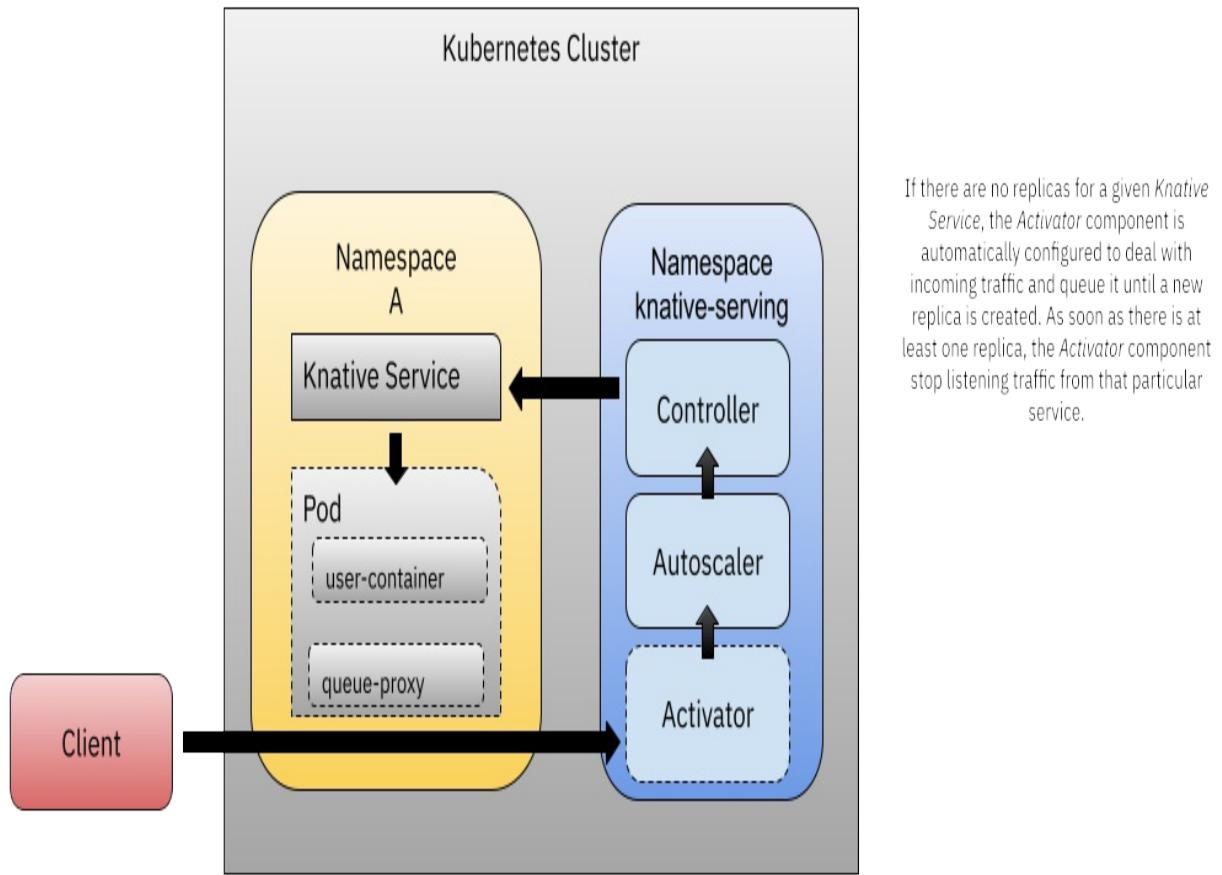
```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-autoscaler
  namespace: knative-serving
data:
  container-concurrency-target-default: "100" # 100 concurrent req
```

Knative Serving also allows you to configure how to deal with sudden burst of traffic doing buffering and making sure that no request is lost. I recommended you to check the official documentation for more details about how these mechanisms can be configured:

<https://knative.dev/docs/serving/autoscaling/concurrency/>

Finally, you might be wondering how Knative Serving deals with traffic when there is no replica available, as both, the `user-container` and the `queue-proxy` are downscaled to zero if no requests are going to the service for some time. Here is where the Activator component jumps into the queue and buffers the requests until a replica is spawned up by the Knative Service Controller, and the requests can be forwarded to that replica.

Figure 7.16 Knative Serving Activator queues traffic for services with no replicas



On top of helping us to implement different release strategies without the burden of creating tons of different Kubernetes resources, Knative Serving also allows us to scale up and down based on the concurrent requests that our services are handling at a given time. These mechanisms work out of the box and are based on live traffic, so there is no need to change the number of replicas of a given service manually, Knative Serving will adapt to the demand, making sure that the number of replicas alive can handle the current traffic and saving resources when they are not needed.

Let's now switch to another alternative to manage Release Strategies in Kubernetes with Argo Rollouts.

7.3 Argo Rollouts: release strategies automated with GitOps

In most cases you will see Argo Rollouts working hand in hand with ArgoCD. This makes a lot of sense because we want to enable a delivery pipeline that removes the need to manually interact with our environments to apply configuration changes. For the examples in the following sections, we will focus only on Argo Rollouts, but in real-life scenarios, you shouldn't be applying resources to the environments using `kubectl`, ArgoCD will do it for you.

Argo Rollouts, as defined on the project website, is: “a Kubernetes controller and set of CRDs which provide advanced deployment capabilities such as blue-green, canary, canary analysis, experimentation, and progressive delivery features to Kubernetes”. As we have seen with other projects Argo Rollouts extend Kubernetes with the concepts of Rollouts, Analysis, and Experimentations to enable progressive delivery features. The main idea with Argo Rollouts is to leverage the Kubernetes built-in blocks without the need to manually modifying and keep track of Deployment and Services resources.

Argo Rollouts is composed of two big parts, the Kubernetes Controller that implements the logic to deal with our Rollouts definitions (also Analysis and Experimentations) and then a `kubectl` plugin which allows you to control how these rollouts progress, enabling manual promotions and rollbacks. Using the `kubectl` Argo Rollouts plugin you can also install the Argo Rollouts Dashboard and run it locally.

You can follow a tutorial on how to install Argo Rollouts on a local Kubernetes KinD cluster here:

<https://github.com/salaboy/from-monolith-to-k8s/blob/main/argorollouts/README.md>

Let's start by looking at how we can implement canary releases with Argo Rollouts to see how it compares with using plain Kubernetes resources or using Knative Services.

7.3.1 Argo Rollouts Canary Rollouts

First we'll begin by creating our first Rollout resource. With Argo Rollouts we will be not defining Deployments as we will be delegating this

responsibility to the Argo Rollouts controller. Instead, we define an Argo Rollout resource that also provides our Pod specification (PodSpec in the same way that a Deployment defines how pods need to be created).

For these examples, we will be using only the Email Service from the Conference Platform application and we will not be using Helm as when using Argo Rollouts we need to deal with a different Resource type which is currently not included in the application Helm charts. Argo Rollouts can work perfectly fine with Helm, but for these examples, we will be creating files to test how Argo Rollouts behave. You can take a look at an Argo Rollout example using Helm here: <https://argoproj.github.io/argo-rollouts/features/helm/>

Let start by creating an Argo Rollout resource for the Email Service:

```
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: email-service-canary
spec:
  replicas: 3
  strategy:
    canary:
      steps:
        - setWeight: 25
        - pause: {}
        - setWeight: 75
        - pause: {duration: 10}
  revisionHistoryLimit: 2
  selector:
    matchLabels:
      app: email-service
  template:
    metadata:
      labels:
        app: email-service
  spec:
    containers:
      - name: email-service
        image: ghcr.io/salaboy/fmtok8s-email-service:v0.1.0-native
        env:
          - name: VERSION
            value: v0.1.0
...
...
```

You can find the full file here: <https://github.com/salaboy/from-monolith-to-k8s/blob/main/argorollouts/canary-release/rollout.yaml>

This Rollout resource manages the creation of Pods using what we define inside the `spec.template` and `spec.replicas` fields. But it adds the `spec.strategy` section which for this case is set to `canary` and defines the steps (amount traffic (weight) that will be sent to the canary) in which the rollout will happen. As you can see you can also define a pause between each step. The `duration` is expressed in seconds and allows us to have a fine grain control on how the traffic is shifted to the canary version. If you don't specify the `duration` parameter, the rollout will wait there until manual intervention happens. Let's see how this rollout works in action.

Let's apply the Rollout resource to our Kubernetes Cluster:

```
> kubectl apply -f rollout.yaml
```

Remember that if you are using Argo CD, instead of manually applying the resource you will be pushing this resource to your Git repository that Argo CD is monitoring. Once the resource is applied, we can see that a new Rollout resource is available by using `kubectl`:

```
> kubectl get rollouts.argoctl.io
NAME                      DESIRED   CURRENT   UP-TO-DATE
email-service-canary      3          3          3
```

This looks pretty much like a normal Kubernetes Deployment, but it is not. If you use `kubectl get deployments` you shouldn't see any Deployment resource available for our `email-service`. Argo Rollouts replace the use of Kubernetes Deployments by using Rollouts resources, which are in charge of creating and manipulating Replica Sets, we can check using `kubectl get rs` that our Rollout has created a new ReplicaSet:

```
> kubectl get rs
NAME                      DESIRED   C
email-service-canary-7f45f4d5c6  3          3
```

Argo Rollouts will create and manage these replica sets that we used to manage with Deployment resources, but in a way that enable us to smoothly perform canary releases.

If you have installed the Argo Rollouts Dashboard you should see our Rollout on the main page:

Figure 7.x Argo Rollouts Dashboard



Argo Rollouts



NS: default

v1.2.2+22aff27

Search...

email-service-canary



Strategy



Weight



email-service-canary-7f454d5c6

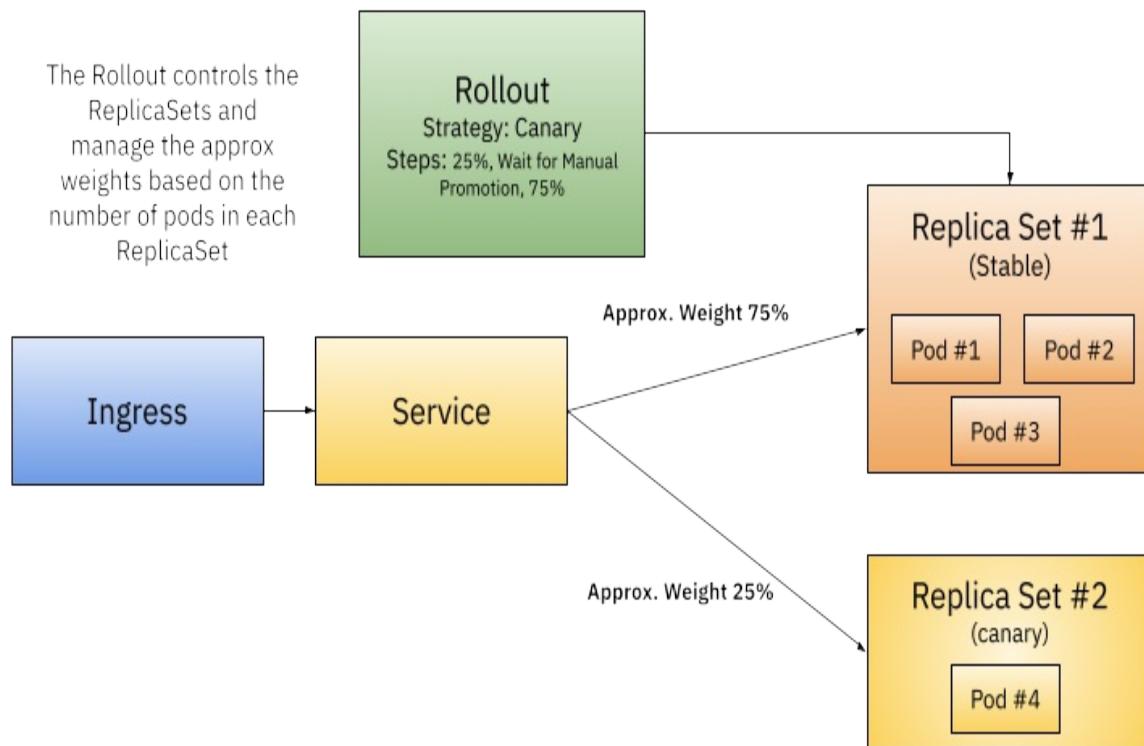


Revision 1



As with Deployments, we still need a Service and an Ingress to route traffic to our service from outside the Cluster. If you create the following resources, you can start interacting with the stable service and with the canary:

Figure 7.x Argo Rollouts Canary Release Kubernetes resources



If you create a service and an ingress you should be able to query the Email Service `info` endpoint by using the following `http` command:

```
> http localhost/info
```

The output should look like this:

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 230
Content-Type: application/json
Date: Thu, 11 Aug 2022 09:38:18 GMT
```

```
{
  "name": "Email Service",
  "podId": "email-service-canary-7f45f4d5c6-fhzzt",
```

```

    "podNamespace": "default",
    "podNodeName": "dev-control-plane",
    "source": "https://github.com/salaboy/fmtok8s-email-service/r
    "version": "v0.1.0"
}

```

The request shows the output of the `info` endpoint of our email service application. Because we have just created this Rollout resource, the Rollout canary strategy mechanism didn't kick in just yet. Now if we want to update the Rollout `spec.template` section with a new container image reference or change environment variables a new revision will be created and the canary strategy will kick in.

In a new terminal, we can watch the Rollout status before doing any modification, so we can see the rollout mechanism in action when we change the Rollout specification. If we want to watch how the rollout progress after we make some changes you can run in a separate terminal the following command:

```
> kubectl argo rollouts get rollout email-service-canary --watch
```

You should see something like this:

```

Name:          email-service-canary
Namespace:     default
Status:        ✓ Healthy
Strategy:      Canary
Step:          8/8
SetWeight:    100
ActualWeight: 100
Images:        ghcr.io/salaboy/fmtok8s-email-service:v0.1.0-nat
Replicas:
  Desired:    3
  Current:    3
  Updated:    3
  Ready:      3
  Available:  3

```

NAME	KIND	STA
email-service-canary	Rollout	✓
# revision:1		
email-service-canary-7f45f4d5c6	ReplicaSet	✓
email-service-canary-7f45f4d5c6-52j9b	Pod	✓ F
email-service-canary-7f45f4d5c6-f8f6g	Pod	✓ F

```
└─□ email-service-canary-7f45f4d5c6-fhzzt Pod
```

✓ F

Let's modify the `rollout.yaml` file with the following two changes:

- **Container image:** `spec.template.spec.containers[0].image` from `ghcr.io/salaboy/fmtok8s-email-service:v0.1.0-native` to `ghcr.io/salaboy/fmtok8s-email-service:v0.2.0-native` . We have just increased the minor version of the container image to v0.2.0-native
- **Environment Variable:** let's also update the Environment Variable called `VERSION` from `v0.1.0` to `v0.2.0`

We can now reapply the `rollout.yaml` with the new changes, this will cause our Rollout resource to be updated in the cluster.

```
> kubectl apply -f rollout.yaml
```

As soon as we apply the new version of the resource the rollout strategy will kick in. If you go back to the terminal where you are watching the rollout, you should see that a new `# revision: 2` was created:

```
NAME                                     KIND
email-service-canary                     Rollout      || P
└─# revision:2
   └─□ email-service-canary-7784fb987d    ReplicaSet  ✓
     └─□ email-service-canary-7784fb987d-q7ztt Pod        ✓ F
```

You can see that revision 2 is labeled as the `canary` and the status of the rollout is `|| Paused` and only one Pod is created for the canary. So far the rollout has only executed the first step:

```
strategy:
  canary:
    steps:
      - setWeight: 25
      - pause: {}
```

You can also check the status of the canary Rollout in the dashboard:

Figure 7.x A canary release has been created with approximately 20% of the traffic being routed to it

email-service-canary



Strategy

Canary

Weight

20

email-service-canary-7784fb987d

Revision 2



email-service-canary-7f45f4d5c6

Revision 1



RESTART



PROMOTE

The Rollout is currently paused waiting for manual intervention. We can now test that our canary is receiving traffic to see if we are happy with how the canary is working before continuing the rollout process. To do that, we can query the `info` endpoint again to see that approximately 25% of the time we hit the canary:

```
salaboy> http localhost/info
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 230
Content-Type: application/json
Date: Fri, 12 Aug 2022 07:56:56 GMT

{
  "name": "Email Service", # stable
  "podId": "email-service-canary-7f45f4d5c6-fhzzt",
  "podNamespace": "default",
  "podNodeName": "dev-control-plane",
  "source": "https://github.com/salaboy/fmtok8s-email-service/r
  "version": "v0.1.0"
}

salaboy> http localhost/info
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 243
Content-Type: application/json
Date: Fri, 12 Aug 2022 07:56:57 GMT

{
  "name": "Email Service - IMPROVED!!", #canary
  "podId": "email-service-canary-7784fb987d-q7ztt",
  "podNamespace": "default",
  "podNodeName": "dev-control-plane",
  "source": "https://github.com/salaboy/fmtok8s-email-service/r
  "version": "v0.2.0"
}
```

We can see that one request hit our stable version and one went to the canary.

Argo Rollouts is not dealing with traffic management in this case, the Rollout resource is only dealing with the underlying Replica Set objects and their replicas. You can check the replicaset by running `kubectl get rs`

```
> kubectl get rs
NAME                                     DESIRED
email-service-canary-7784fb987d          1
email-service-canary-7f45f4d5c6          3
```

The traffic management between these different pods (canary and stable pods) is being managed by the Kubernetes Service resource, hence in order to see our request hitting both, the canary and the stable version pods we need to go through the Kubernetes Service. I am only mentioning this, because if you use `kubectl port-forward svc/email-service 8080:80` for example, you might be tempted to think that traffic is being forwarded to the Kubernetes service (as we are using `svc/email-service`) but `kubectl port-forward` resolves to a Pod instance and connect to a single pod, allowing you only hit the canary or a stable pod. For this reason, we have used an Ingress which will use the Service to load balance traffic and hit all the pods that are matching to the Service selector.

If we are happy with the We can continue the rollout process by executing the following command which promotes the canary to be the stable version:

```
> kubectl argo rollouts promote email-service-canary
```

Although we have just manually promoted the rollout, the best practice would be utilizing Argo Rollouts automated analysis steps which we will dig into in section 7.3.2.

If you take a look at the Dashboard you will notice that you can also promote the rollout to move forward using the Button Promote in the Rollout. Promotion in this context only means that the rollout can continue to execute the next steps defined in the `spec.strategy` section:

```
strategy:
  canary:
    steps:
      - setWeight: 25
      - pause: {}
      - setWeight: 75
      - pause: {duration: 10}
```

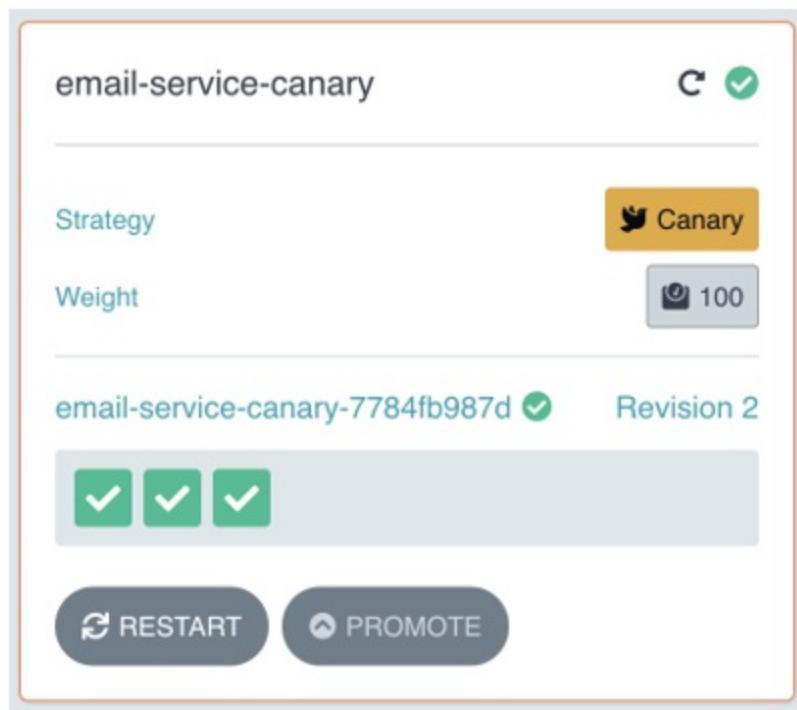
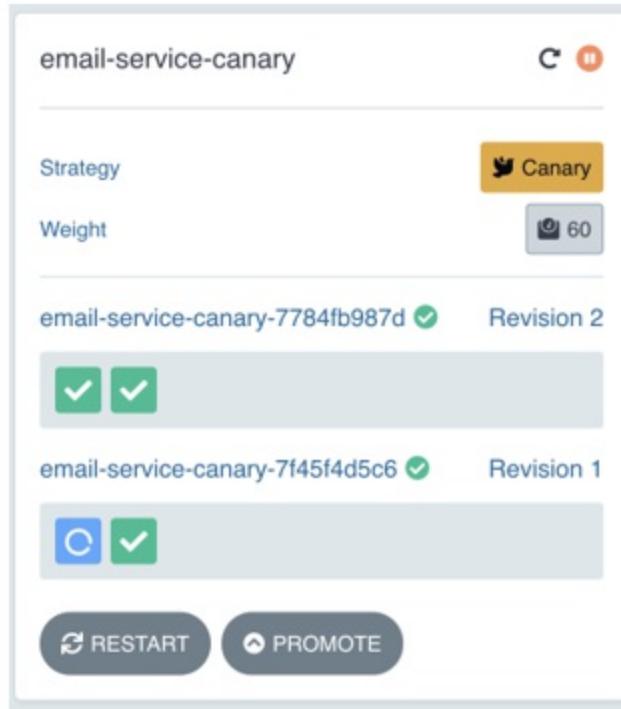
After the manual promotion, the weight is going to be set to 75% followed by a pause of 10 seconds, to finally set the wait to 100%. At that point, you

should see that revision 1 is being downscaled while progressively revision 2 is being upscaled to take all the traffic.

NAME	KIND	STA
email-service-canary	Rollout	✓
# revision:2		
email-service-canary-7784fb987d	ReplicaSet	✓
email-service-canary-7784fb987d-q7ztt	Pod	✓ F
email-service-canary-7784fb987d-zmd7v	Pod	✓ F
email-service-canary-7784fb987d-hwwbk	Pod	✓ F
# revision:1		
email-service-canary-7f45f4d5c6	ReplicaSet	• S

You can see this rollout progression live in the dashboard as well

Figure 7.x The canary revision is promoted to be the stable version



As you can see revision 1 was downscaled to have zero pods and revision 2 is now marked as the stable version. If you check the Replica Sets you will see the same output:

```
> kubectl get rs
```

NAME		DESIRED
email-service-canary-7784fb987d	3	3
email-service-canary-7f45f4d5c6	0	0

We have successfully created, tested and promoted a canary release with Argo Rollouts!

Compared to what we have seen in section 7.1.2 using two Deployment resources to implement the same pattern, with Argo Rollouts you have full control over how your canary release is promoted, how much time do you want to wait before shifting more traffic to the canary and how many manual interventions steps do you want to add.

Let's now jump to see how a Blue-Green Deployment works with Argo Rollouts.

7.3.2 Argo Rollouts Blue-Green Deployments

In section 2.5.3 we covered the advantages and the reasons behind why you would be interested in doing a Blue-Green deployment using Kubernetes basic building blocks. We have also seen how manual the process is and how these manual steps can open the door for silly mistakes that can bring our services down. In this section, we will look at how Argo Rollouts allows us to implement Blue-Green deployments following the same approach that we used previously for Canary Deployments.

Let's take a look at how our Rollout with blueGreen strategy looks like:

```
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: email-service-bluegreen
spec:
  replicas: 2
  revisionHistoryLimit: 2
  selector:
    matchLabels:
      app: email-service
  template:
    metadata:
      labels:
```

```

    app: email-service
spec:
  containers:
    - name: email-service
      image: ghcr.io/salaboy/fmtok8s-email-service:v0.1.0-native
      env:
        - name: VERSION
          value: v0.1.0
      imagePullPolicy: Always
      ports:
        - name: http
          containerPort: 8080
          protocol: TCP
  strategy:
    blueGreen:
      activeService: email-service-active
      previewService: email-service-preview
      autoPromotionEnabled: false

```

You can find the full file here: <https://github.com/salaboy/from-monolith-to-k8s/blob/main/argorollouts/blue-green/rollout.yaml>

Let's apply this Rollout resource using `kubectl` or by pushing this resource to a Git repository if you are using ArgoCD.

```
> kubectl apply -f rollout.yaml
```

We are using the same `spec.template` as before but now we are setting the strategy of the rollout to be `blueGreen` and because of that we need to configure the reference to two Kubernetes services. One service will be the Active Service (Blue) which is serving production traffic and the other one is the Green service that we want to preview but without routing production traffic to it. The `autoPromotionEnabled: false` is required to allow for manual intervention for the promotion to happen. By default, the rollout will be automatically promoted as soon as the new ReplicaSet is ready/available.

You can watch the rollout running the following command or in the Argo Rollouts Dashboard:

```
> kubectl argo rollouts get rollout email-service-bluegreen --watch
```

You should see a similar output to the one we saw for the canary release:

```
Name: email-service-bluegreen
Namespace: default
Status: ✓ Healthy
Strategy: BlueGreen
Images: ghcr.io/salaboy/fmtok8s-email-service:v0.1.0-nat
Replicas:
  Desired: 2
  Current: 2
  Updated: 2
  Ready: 2
  Available: 2
```

```
NAME                                     KIND
email-service-bluegreen                  Rollout
└─# revision:1
   └─✉ email-service-bluegreen-54b5fd4d7c      ReplicaSet
     └─□ email-service-bluegreen-54b5fd4d7c-gvwvt  Pod
     └─□ email-service-bluegreen-54b5fd4d7c-r9dxs  Pod
```

And in the dashboard you should see something like this:

Figure 7.x Blue-Green deployment in Argo Rollout Dashboard

email-service-bluegreen



Strategy

 BlueGreen

email-service-bluegreen-54b5fd4d7c  Revision 1



 RESTART

 PROMOTE

We can interact with revision #1 using an Ingress to the service and then sending a request like the following:

```
> http localhost/info
```

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 233
Content-Type: application/json
Date: Sat, 13 Aug 2022 08:46:44 GMT
```

```
{
  "name": "Email Service",
  "podId": "email-service-bluegreen-54b5fd4d7c-jnszp",
  "podNamespace": "default",
  "podNodeName": "dev-control-plane",
  "source": "https://github.com/salaboy/fmtok8s-email-service/r",
  "version": "v0.1.0"
}
```

If we now make changes to our Rollout `spec.template` the blueGreen strategy will kick in. For this example the expected result that we want to see is that the previewService is now routing traffic to the second revision that is created when we save the changes into the rollout.

Let's modify the `rollout.yaml` file with the following two changes:

- **Container image:** `spec.template.spec.containers[0].image` from `ghcr.io/salaboy/fmtok8s-email-service:v0.1.0-native` to `ghcr.io/salaboy/fmtok8s-email-service:v0.2.0-native` . We have just increased the minor version of the container image to v0.2.0-native
- **Environment Variable:** let's also update the Environment Variable called `VERSION` from `v0.1.0` to `v0.2.0`

We can now reapply the `rollout.yaml` with the new changes, this will cause our Rollout resource to be updated in the cluster.

```
> kubectl apply -f rollout.yaml
```

As soon as we save these changes, the rollout mechanism will kick in and it will automatically create a new Replica Set with revision 2 including our changes. Argo Rollouts for Blue-Green deployments will use selectors to route traffic to our new revision by modifying the previewService that we have referenced in our Rollout definition.

If you describe the `email-service-preview` Kubernetes service you will notice that a new selector was added:

```
> kubectl describe svc email-service-preview
```

Name :	email-service-preview
--------	-----------------------

```
Namespace:           default
Labels:
Annotations:        argo-rollouts.argoproj.io/managed-by-rollouts:
Selector:           app=email-service,rollouts-pod-template-hash=6
Type:               ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                 10.96.27.4
IPs:                10.96.27.4
Port:               http  80/TCP
TargetPort:         http/TCP
Endpoints:          10.244.0.23:8080,10.244.0.24:8080
Session Affinity:  None
Events:
```

This selector is matching with the revision 2 Replica Set that was created when we made the changes.

```
> kubectl describe rs email-service-bluegreen-64d9b549cf
```

```
Name:                 email-service-bluegreen-64d9b549cf
Namespace:            default
Selector:             app=email-service,rollouts-pod-template-hash=64d9
Labels:               app=email-service
                      rollouts-pod-template-hash=64d9b549cf
Annotations:          rollout.argoproj.io/desired-replicas: 2
                      rollout.argoproj.io/revision: 2
Controlled By:       Rollout/email-service-bluegreen
Replicas:             2 current / 2 desired
Pods Status:          2 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:              app=email-service
                      rollouts-pod-template-hash=64d9b549cf
```

By using the selector and labels the Rollout with blueGreen strategy is handling these links automatically for us. This avoids us manually creating these labels and making sure they match.

You can check now that there are two revisions (and ReplicaSets) with 2 pods each.

NAME	KIND
email-service-bluegreen	Rollout
# revision:2	
email-service-bluegreen-64d9b549cf	ReplicaSet

```
    └─── email-service-bluegreen-64d9b549cf-glkjv Pod
        └─── email-service-bluegreen-64d9b549cf-sv6v2 Pod
# revision:1
└─── email-service-bluegreen-54b5fd4d7c ReplicaSet
    ├─── email-service-bluegreen-54b5fd4d7c-gvvwt Pod
    └─── email-service-bluegreen-54b5fd4d7c-r9dxs Pod
```

In the Argo Rollouts Dashboard you should see the same information:

Figure 7.x Argo Rollout Dashboard Blue and Green revisions are up

email-service-bluegreen



Strategy

 BlueGreen

email-service-bluegreen-64d9b549cf  Revision 2



email-service-bluegreen-54b5fd4d7c  Revision 1



 RESTART

 PROMOTE

We can now interact with the Preview Service (revision #2) using a different path in our Ingress:

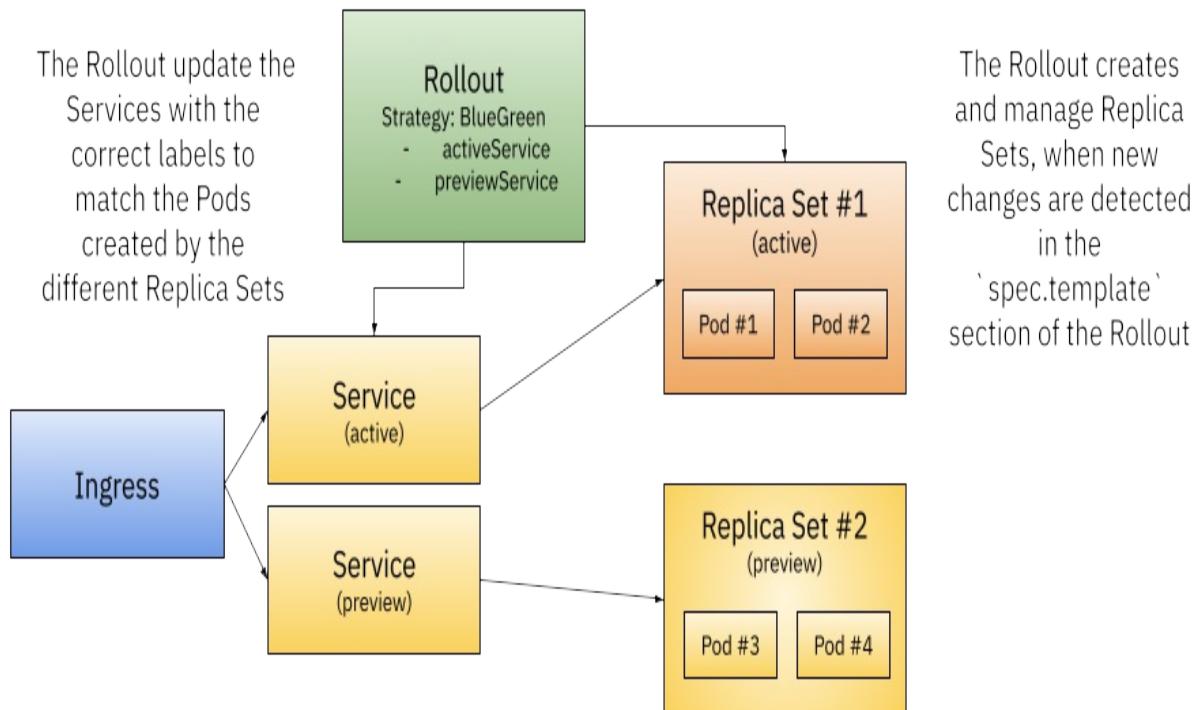
```
> http localhost/preview/info
```

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 246
Content-Type: application/json
Date: Sat, 13 Aug 2022 08:50:28 GMT
```

```
{
  "name": "Email Service - IMPROVED!!",
  "podId": "email-service-bluegreen-64d9b549cf-8fpnr",
  "podNamespace": "default",
  "podNodeName": "dev-control-plane",
  "source": "https://github.com/salaboy/fmtok8s-email-service/r
  "version": "v0.2.0"
}
```

Once we have the preview (Green) service running the Rollout is in a Paused state until we decide to promote it to be the stable service.

Figure 7.x Blue-Green deployment Kubernetes Resources



Because we now have two services, we can access both at the same time and make sure that our Green (preview-service) is working as expected before promoting it to be our main (active) service. While the service is in preview, other services in the cluster can start routing traffic to it for testing purposes, but to route all the traffic and replace our blue service with our green service we can use once again the Argo Rollouts promotion mechanism from the terminal using the CLI or from the Argo Rollouts Dashboard. Try to promote the Rollout using the Dashboard now instead of using `kubectl`.

Notice that a 30 seconds delay is added by default before the scaling down of our revision #1 (this can be controlled using the property called: `scaleDownDelaySeconds`), but the promotion (switching labels to the services) happens the moment we hit the `PROMOTE` button.

Figure 7.x Green service promotion using the Argo Rollouts Dashboard (delay 30 seconds)

email-service-bluegreen



Strategy

 BlueGreen

email-service-bluegreen-64d9b549cf  Revision 2



email-service-bluegreen-
54b5fd4d7c

Revision
1 

 00:19
min



 RESTART

 PROMOTE

This promotion only switches labels to the services resources, which automatically changes the routing tables to now forward all the traffic from the Active Service to our Green Service (preview).

If we make more changes to our Rollout the process will start again and the preview service will point to a new revision which will include these changes.

Now that we have seen the basics of Canary Releases and Blue-Green deployments with Argo Rollouts, let's take a look at more advanced mechanisms provided by Argo Rollouts.

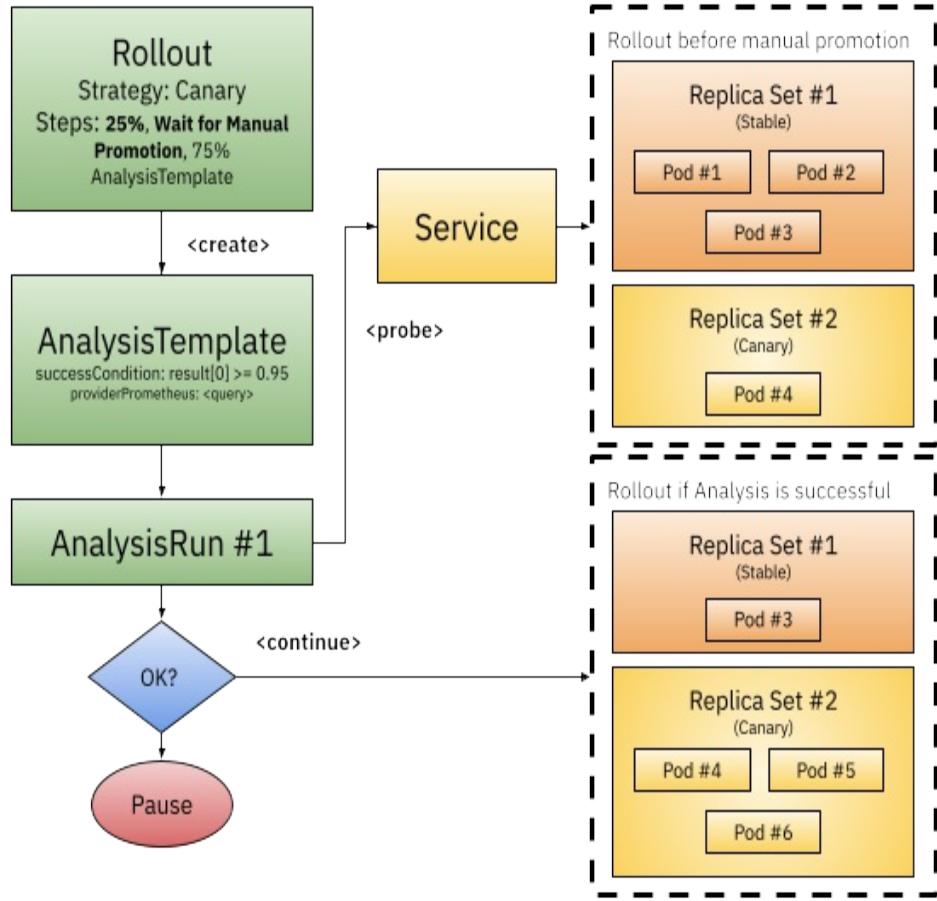
7.3.3 Argo Rollouts Analysis for Progressive Delivery

So far, we have managed to have more control over our different release strategies but where Argo Rollouts really shine is by providing the `AnalysisTemplate` CRD which let us make sure that our Canary and Green services are working as expected when progressing through our rollouts. These Analyses are automated and serve as gates for our Rollouts to not progress unless the analysis probes are successful.

These analyses can use different providers to run the probes, ranging from Prometheus, DataDog, and New Relic among others, providing maximum flexibility to define these automated tests against the new revisions of our services.

Figure 7.x Argo Rollouts and Analysis working together to make sure that our new revisions are sound before shifting more traffic to them

When receiving the signal to move forward to the next step of the Rollout an 'AnalysisRun' is created to probe the service by running a query defined in the 'AnalysisTemplate'. The 'AnalysisRun' result affect if the 'Rollout' 's update will continue, abort, or pause.



For canary releases analysis can be triggered as part of the step definitions, meaning between arbitrary steps, to start at a predefined step or for every step defined in the Rollout.

An AnalysisTemplate using the Prometheus Provider definition look like this:

```

apiVersion: argoproj.io/v1alpha1
kind: AnalysisTemplate
metadata:
  name: success-rate
spec:
  args:
  - name: service-name
  metrics:
  - name: success-rate
    interval: 5m
    # NOTE: prometheus queries return results in the form of a ve
    # So it is common to access the index 0 of the returned array
    successCondition: result[0] >= 0.95
    failureLimit: 3
  
```

```
provider:  
  prometheus:  
    address: http://prometheus.example.com:9090  
    query: <Prometheus Query here>
```

Then in our Rollout we can make reference to this template and define when a new AnalysisRun will be created, for example if we want to run the first analysis after step 2:

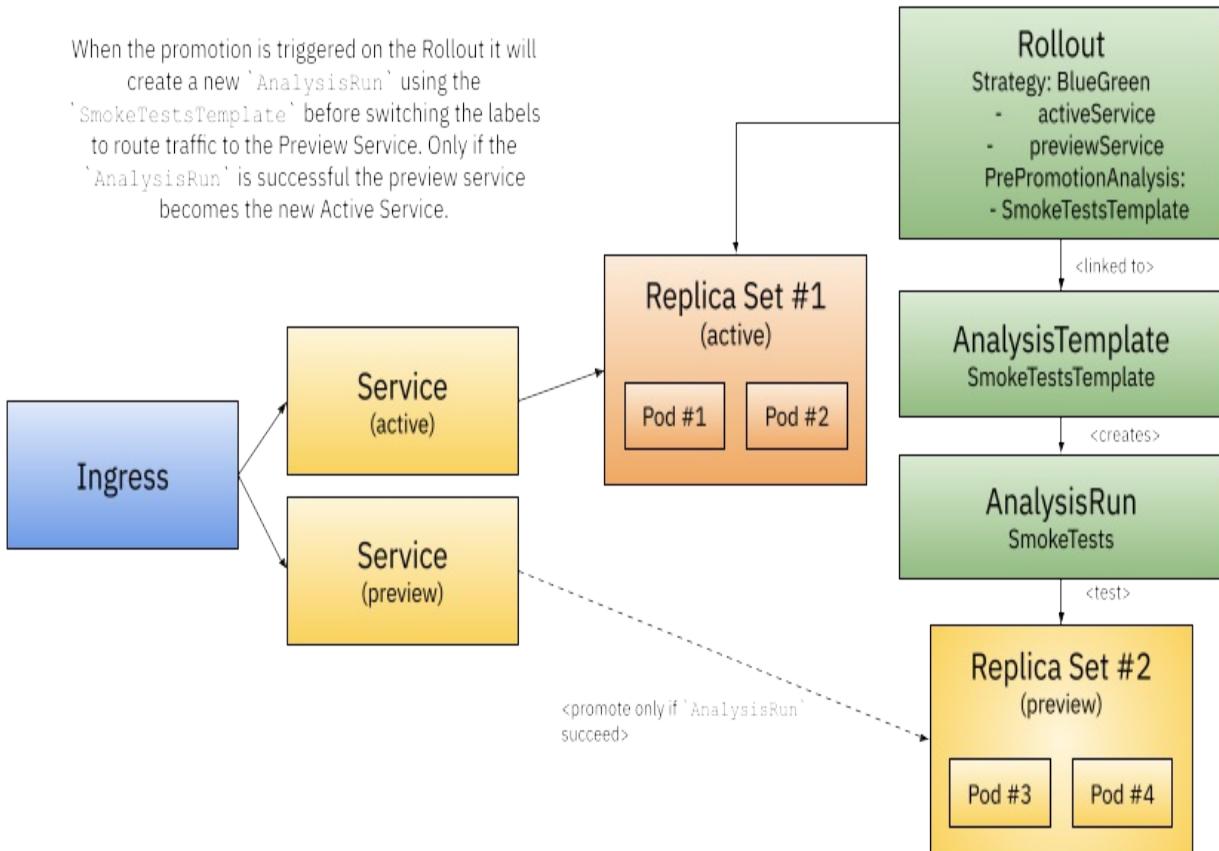
```
strategy:  
  canary:  
    analysis:  
      templates:  
        - templateName: success-rate  
    startingStep: 2 # delay starting analysis run until setWe  
    args:  
      - name: service-name  
        value: email-service-canary.default.svc.cluster.local
```

As mentioned before, the analysis can be also defined as part of the steps, in that case our steps definition will look like this:

```
strategy:  
  canary:  
    steps:  
      - setWeight: 20  
      - pause: {duration: 5m}  
      - analysis:  
          templates:  
            - templateName: success-rate  
          args:  
            - name: service-name  
              value: email-service-canary.default.svc.cluster.local
```

For Rollouts using a BlueGreen strategy we can trigger Analysis runs pre and post promotion.

Figure 7.x Argo Rollouts with blueGreen deployments, PrePromotionAnalysis in action



Here is an example of PrePromotionAnalysis configured in our Rollout:

```

apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: email-service-rollout
spec:
  ...
  strategy:
    blueGreen:
      activeService: email-service-active
      previewService: email-service-preview
      prePromotionAnalysis:
        templates:
        - templateName: smoke-tests
      args:
      - name: service-name
        value: email-service-preview.default.svc.cluster.local
  
```

For PrePromotion tests, a new AnalysisRun a test before switching traffic to the Green Service, and only if the test is successful the labels will be updated.

For PostPromotion, the test will run after the labels were switched to the Green Service and if the AnalysisRun fail the rollout can revert back the labels to the previous version automatically, this is possible because the Blue Service will not be downscaled until the AnalysisRun finishes.

I recommend you to check the Analysis section of the official documentation as it contains a detailed explanation of all the providers and knobs that you can use for making sure that your Rollouts go smoothly:

<https://argoproj.github.io/argo-rollouts/features/analysis/>

7.3.4 Argo Rollouts and Traffic Management

Finally it is worth mentioning that so far Rollouts had been using the number of pods available to approximate the weights that we define for canary releases. While this is a good start and a simple mechanism, sometimes we need more control on how traffic is routed to different revisions. We can leverage the power of service meshes and load balancers to write more precise rules about which traffic is routed to our canary releases.

Argo Rollouts can be configured with different `trafficRouting` rules depending which traffic management tool we have available in our Kubernetes Cluster. Argo Rollouts supports today: Istio, AWS ALB Ingress Controller, Ambassador Edge Stack, Nginx Ingress Controller, Service Mesh Interface (SMI), Traefik Proxy, among others. As described in the documentation, if we have more advanced traffic management capabilities we can implement techniques like:

- Raw percentages (i.e., 5% of traffic should go to the new version while the rest goes to the stable version)
- Header-based routing (i.e., send requests with a specific header to the new version)
- Mirrored traffic where all the traffic is copied and send to the new version in parallel (but the response is ignored)

By using tools like Istio in conjunction with Argo Rollouts we can enable developers to test features that are only available to request setting specific headers, or to forward copies of the production traffic to the canaries to

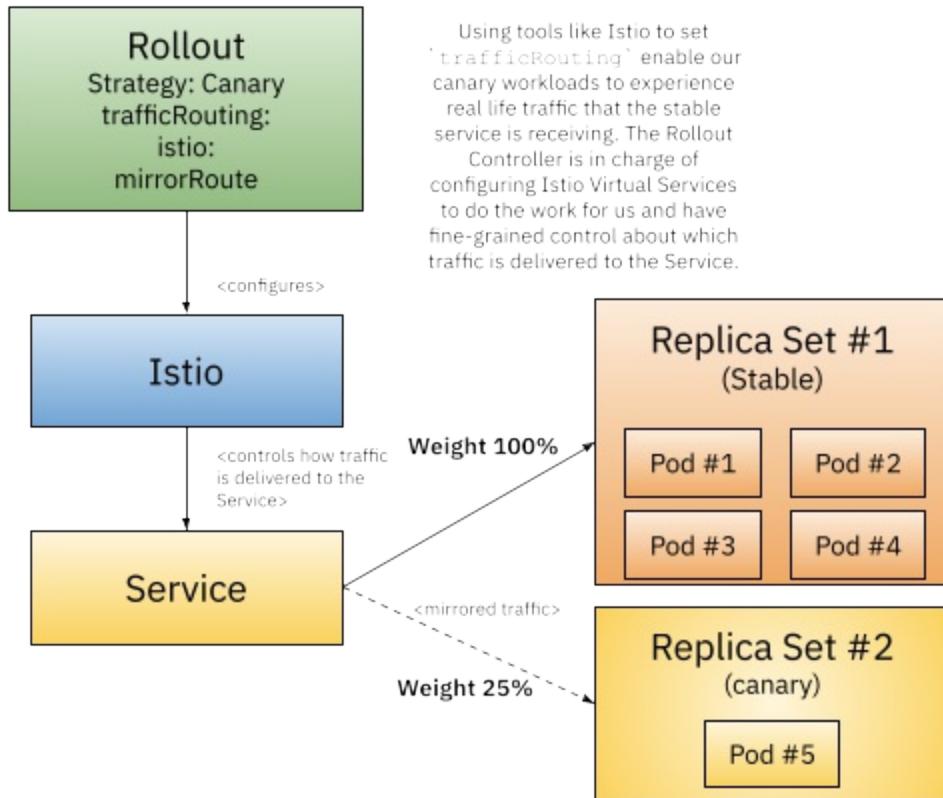
validate that they are behaving as they should.

Here is an example of configuring a Rollout to mirror 35% of the traffic to the canary release which has a 25% weight:

```
apiVersion: argoproj.io/v1alpha1
kind: Rollout
spec:
  ...
  strategy:
    canary:
      canaryService: email-service-canary
      stableService: email-service-stable
      trafficRouting:
        managedRoutes:
          - name: mirror-route
    istio:
      virtualService:
        name: email-service-vs
    steps:
      - setCanaryScale:
          weight: 25
      - setMirrorRoute:
          name: mirror-route
          percentage: 35
          match:
            - method:
                exact: GET
            path:
              prefix: /
      - pause:
          duration: 10m
      - setMirrorRoute:
          name: "mirror-route" # removes mirror based traffic rou
```

As you can see this simple example already requires knowledge around Istio Virtual Services and a more advanced configuration that is out of the scope for this section. I strongly recommend to check the Istio in Action book by Christian Posta & Rinor Maloku (<https://www.manning.com/books/istio-in-action>) if you are interested in learning about Istio.

Figure 7.x Traffic Mirroring to canary release using Istio



Something that you should know is that when using `trafficManagement` features, the Rollout canary strategy will behave differently than when we are not using any rules. More specifically, the Stable version of the service will not be downscaled when going through a canary release rollout. This ensures that the Stable service can handle 100% of the traffic, the usual calculations apply for the Canary replica count.

I strongly recommend checking the official documentation (<https://argoproj.github.io/argo-rollouts/features/traffic-management/>) and follow the examples there as the rollouts need to be configured differently depending on the Service Mesh that you have available.

7.4 Final thoughts

In this chapter, we have seen what can be achieved with basic Kubernetes building blocks and how tools like Argo Rollouts or Knative Serving simplify the life of teams by releasing new versions of their applications to Kubernetes.

It is unfortunate that as of today, in 2023, Argo Rollouts and Knative Serving haven't been integrated yet (<https://github.com/argoproj/argo-rollouts/issues/2186>), as both communities would benefit from a consolidated way of defining release strategies instead of duplicating functionality. I personally like the Knative Serving building blocks that facilitate the way of implementing these release strategies. On the other hand, I like how Argo Rollouts takes things to the next level with the concepts of `AnalysisTemplates` to make sure that we can automatically test and validate new releases.

It is my firm belief that sooner or later in your Kubernetes journey, you will face delivery challenges, and having these mechanisms available inside your clusters will increase your confidence to release more software faster. Hence I don't take the evaluation of these tools lightly. Make sure you plan time for your teams to research and choose which tools they will use to implement these release strategies, many software vendors can assist you and provide recommendations too.

In the next chapter, we will evaluate how the platform can provide shared capabilities to reduce the friction between application code and the application infrastructure needed to run the application services. By providing a set of standard capabilities that applications can rely upon, we can speed up the application delivery process even further.

7.5 Summary

- Kubernetes built-in Deployments and Services can be used to implement rolling updates, canary releases, blue-green deployments, and A/B testing patterns. This allows us to manually move from one version of our service to the next without causing Service disruption.
- Canary Releases can be implemented by having a Kubernetes Service, two Deployments (one for each version), and a shared label that routes traffic to both versions at the same time
- Blue-Green deployments can be implemented by having a Kubernetes a single Kubernetes Service, two independent deployments and switching labels when the new version has been tested and it is ready to be promoted

- A/B Testing can be implemented by having two Services and two Deployments and two Ingress resources
- Knative Serving introduces an advanced networking layer that allows us to have fine-grain control about how traffic is routed to different versions of our services that can be deployed at the same time. This feature is implemented on top of Knative Services and reduces the manual work of creating several Kubernetes resources for implementing canary releases, blue-green deployments and A/B testing release strategies. Knative Serving simplifies the operational burden of moving traffic to new versions and, with the help of the autoscaler can scale up and down based on demand.
- Argo Rollouts integrates with ArgoCD (that we discussed in chapter 4) and provides an alternative to implement different release strategies using the concept of Rollouts. Argo Rollouts also include features to automate the testing of new releases to make sure that we move safely between versions.

8 Platform capabilities II: Shared application concerns

This chapter covers

- Common patterns and capabilities required by 95% of all Cloud-Native applications
- Speeding up delivery by reducing friction between application and infrastructure
- Using standard APIs and components for shared concerns across services and for a simplified developer experience

Now your teams can fully leverage the power of applying different release strategies to release and deploy more code in front of their users. The next step to unlock more velocity is to reduce the dependencies between the application services and the shared infrastructure that these services need to run.

This chapter will look into some mechanisms to provide developers with capabilities closer to their application needs. Most of these capabilities will be accessed by APIs that abstract away the application's infrastructure needed, allowing the platform team to evolve (update, reconfigure, change) infrastructural components without updating any application code. At the same time, developers will interact with these platform capabilities without knowing how they are implemented and without bloating their applications with a load of dependencies.

This chapter is divided into three sections:

- What are most applications doing 95% of the time?
- Standard APIs and abstractions to separate application code from infrastructure
- Updating our Conference Application with Dapr

Let's start by analyzing what most applications are doing. Don't worry; we will also cover edge cases.

8.1 What are most applications doing 95% of the time?

We have been working with our walking-skeleton Conference applications for 8 Chapters now. We have learned how to run it on top of Kubernetes and how to connect the services to databases, key-value stores, and message brokers. There was a good reason to go over those steps and include those behaviors in the walking skeleton. Most applications, like the conference application, will need the following functionality:

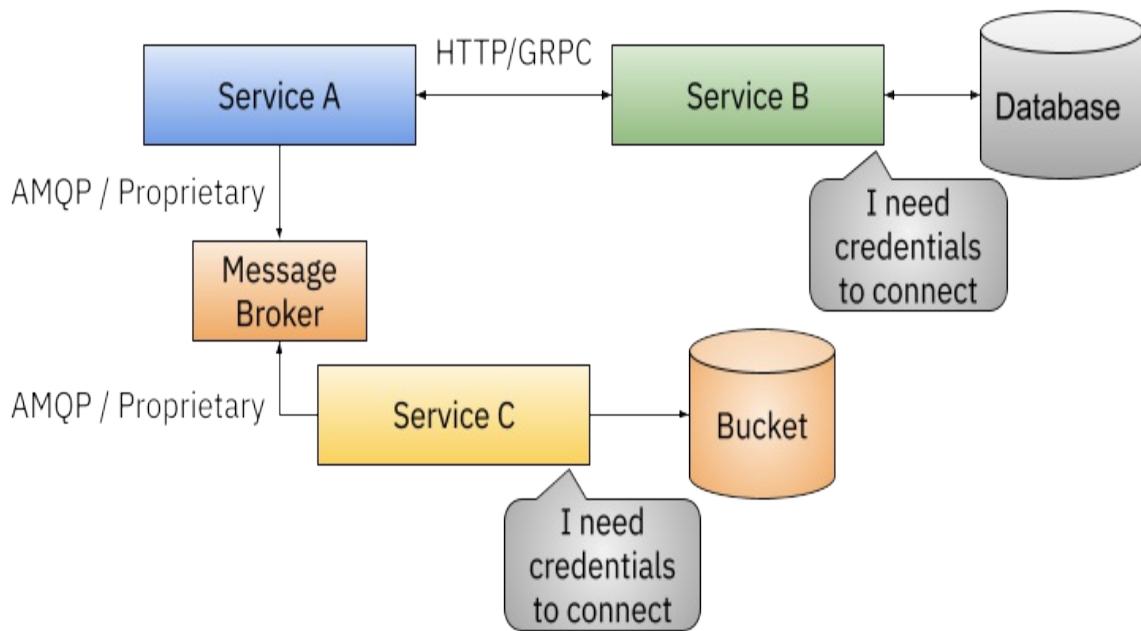
- **Call other services to send or receive information:** application services don't exist on their own. They need to call and be called by other services. Services can be local or remote, and you can use different protocols, the most common being HTTP and GRPC. We use HTTP calls between services for the conference application walking skeleton.
- **Store and read data from persistent storage:** this can be a database, a key-value store, a blob store like S3 buckets, or even writing and reading from files. For the conference application, we are using Redis and PostgreSQL.
- **Emit and consume events or messages asynchronously:** using asynchronous messaging for batch processing or communicating systems implementing an event-driven architecture is quite a common practice in distributed systems. Using tools like Kafka, RabbitMQ, or even Cloud Provider messaging systems is common.
- **Accessing credentials to connect to services:** when connecting to an application's infrastructure components, whether local or remote, most services will need credentials to authenticate to other systems.

Whether we are building business applications or machine learning tools, most applications will benefit from having these capabilities easily available to consume. And while complex applications require much more than that, there is always a way to separate the complex part from the generic parts.

Figure 8.x shows some example services interactions with each other and

available infrastructure. **Service A** is calling **Service B** using **HTTP** (for this topic, **GRPC** would fit similarly). **Service B** stores and reads data from a **database** and will need the right **credentials** to connect. **Service A** also connects to a message broker and sends messages to it. **Service C** can pick messages from the message broker and, using some credentials, connect to a **Bucket** to store some calculations based on the messages it receives.

Figure 8.x Common behaviors in distributed applications



No matter what logic these services are implementing, we can extract some constant behaviors and enable development teams to consume without the hassle of dealing with the low-level details or pushing them to make decisions around cross-cutting concerns that can be solved at the platform level.

To understand how this can work, we must look closely at what is happening inside these services. As you might know already, the devil is in the details. While from a high-level perspective, we are used to dealing with services doing what is described in Figure 8.x, if we want to unlock an increased velocity in our software delivery pipelines, we need to go one level down to understand the intricate relationships between the components of our applications. Let's take a quick look at the challenges the application teams

will face when trying to change the different services and the infrastructure that our services require.

8.1.1 The challenges of coupling application and infrastructure

Fortunately, this is not a programming language competition, independently from your programming language of choice. If you want to connect to a Database or Message broker, you will need to add some dependencies to your application code. While this is a common practice in the software development industry, it is also one of the reasons why delivery speed is slower than it should be.

Coordination between different teams is the reason behind most blockers when releasing more software. We have created architectures and adopted Kubernetes because we want to go faster. By using containers, we have adopted an easier and more standard way to run our applications. No matter in which language our application is written or which tech stack is used, if you give me a container with the application inside, I can run it. We have removed the application dependencies on the Operating System and the software that we need to have installed in a machine (or virtual machine) to run your application which is now encapsulated inside a container.

Unfortunately, by adopting containers, we haven't tackled the relationships and integration points between containers (our application's services). We also haven't solved how these containers will interact with application infrastructure components that can be local (self-hosted) or managed by a Cloud Provider.

Let's take a closer look at where these applications heavily rely on other services and can block teams from making changes, pushing them for complicated coordination that can end up causing downtime to our users. We will start by splitting up the previous example into the specifics of each interaction.

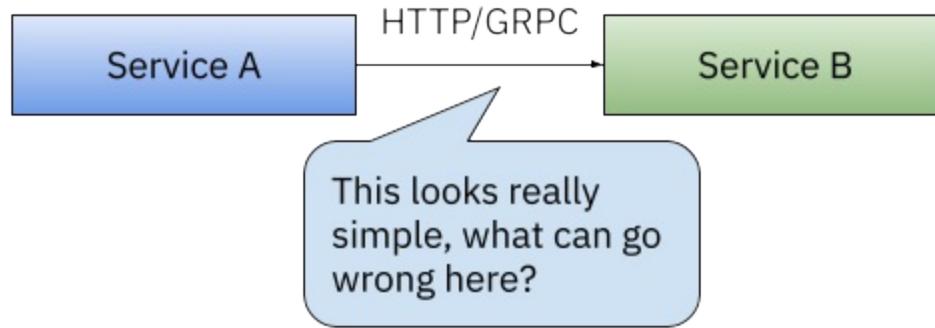
Service-to-Service interactions challenges

To send data from one service to another, you must know where the other

service is running and which protocol it uses to receive information. Because we are dealing with distributed systems, we also need to ensure that the requests between services arrive at the other service and have mechanisms to deal with unexpected network issues or situations where the other services might fail. In other words, we need to build resilience in our services. We cannot trust the network or other Services always to behave as expected.

Let's use Service A and Service B as examples to go a little bit deeper into the details. In Figure 8.x Service A needs to send a request to Service B.

Figure 8.x Service to service interactions challenges



Suppose we leave the fact that Service A depends on the Service B contract (API) to be stable and not change for this to work on the side. What else can go wrong here? As mentioned, development teams should add a resiliency layer inside their services to ensure that Service A requests reach Service B. One way to do this is to use a framework to retry the request if it fails automatically. Frameworks implementing this functionality are available for all programming languages, tools like `go-retryablehttp` (<https://github.com/hashicorp/go-retryablehttp>) or Spring Retry for Spring Boot (<https://www.baeldung.com/spring-retry>) add resiliency to your service to service interactions. Some of these mechanisms also include exponential backoff functionality to avoid overloading services and the network when things are going wrong.

Unfortunately, there is no standard library shared across tech stacks that can provide the same behavior and functionality for all your applications, so even if you configure both Spring Retry and `go-retryablehttp` with similar

parameters, it is quite hard to guarantee that they will behave in the same way when services start failing.

Figure 8.x Service-to-service interactions retry mechanisms

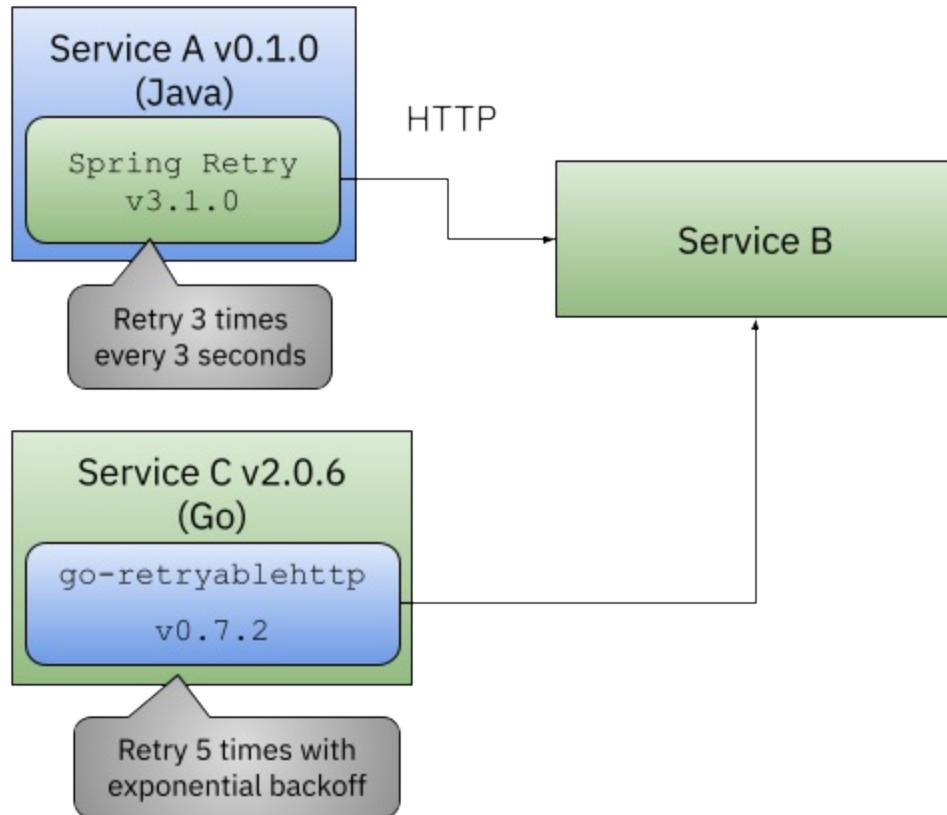


Figure 8.x shows Service A written in Java using the Spring Retry library to retry 3 times with a wait time of 3 seconds between each request when the request fails to be acknowledged by Service B. Service C written in Go is using the `go-retryablehttp` library to retry 5 times but using an exponential backoff (the retry period between requests is not fixed, this can help to wait for the other service to recover and not be flooded with retries) mechanism when things go wrong.

Even if the applications are written in the same language and using the same frameworks, both services (A and B) must have compatible versions of their dependencies and configurations. If we push both Service A and Service B to have the versions of the frameworks, we are coupling them together, meaning we will need to coordinate the update of the other service whenever any of

these internal dependency versions change. This can cause even more slowdowns and complex coordination efforts.

Note

In this section, I've used retrying mechanisms as an example but think about other cross-cutting concerns that you might want to include for these service-to-service interactions, like circuit breakers (also for resiliency) or observability. Consider the frameworks and libraries you will need to add to instrument your application code to get metrics from it.

On the other hand, using different frameworks (and versions) for each service will complicate troubleshooting these services for our operations teams.

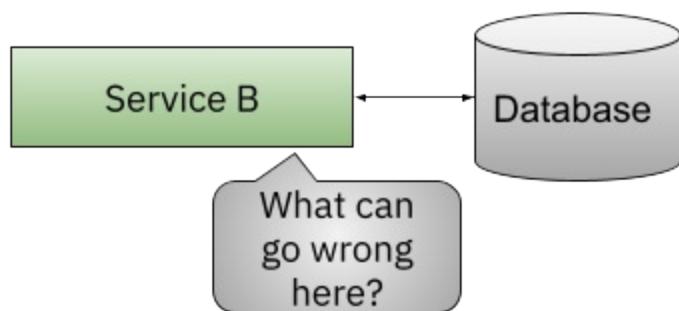
Wouldn't it be great to have a way to add resiliency to our applications without modifying them?

Before answering this question, what else can go wrong?

Storing/Reading state challenges

Our application needs to store or read state from persistent storage. That is quite a common requirement, right? You need data to do some calculations, then store the results somewhere so they don't get lost if your application goes down. In our example, Figure 8.x, Service B needed to connect to a database or persistent storage to read and write data.

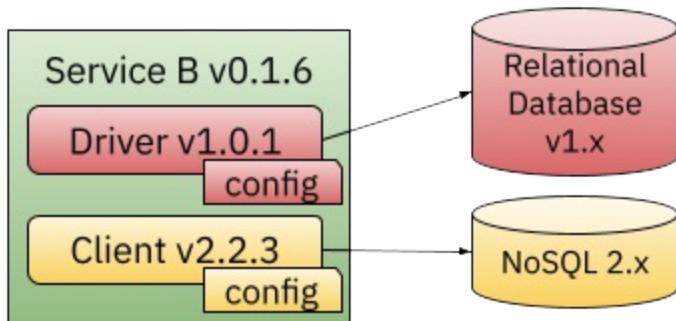
Figure 8.x Storing/Reading state challenges



What can go wrong here? Developers are used to connecting to different kinds of databases (relational, NoSQL, Files, Buckets) and interacting with them—two main friction points slow teams from moving their services forward: Dependencies and credentials.

Let's start by looking at dependencies. What kind of dependencies does Service B need to connect to a database? Figure 8.x shows Service B connecting to both a relational database and a NoSQL database. To achieve these connections Service B needs to include a Driver and a Client library plus the config needed to fine-tune how the application will connect to these two kinds of databases. These configurations define how big is the connection pool (how many application threads can connect concurrently to the database), buffers, health checks and other important details that can change how the application behaves.

Figure 8.x Databases dependencies and clients versions



Connecting to a database will require your application to include a driver or a client dependency to talk to the specific database provider. These drivers or clients can be fine-tuned to behave differently depending on the application's requirements and the database's configurations. Besides their configuration, the version of these drivers and clients needs to be compatible with the version of the databases we are running, and this is where the challenges begin.

Note

It is important to notice that each Driver/Client is specific to the database

(relational or NoSQL) that you are connecting to. This section assumes you used a specific database because it meets your application's requirements. Each database vendor has unique features optimized for different use cases. In this chapter, we are more interested in 95% of the cases that do not use vendor-specific features.

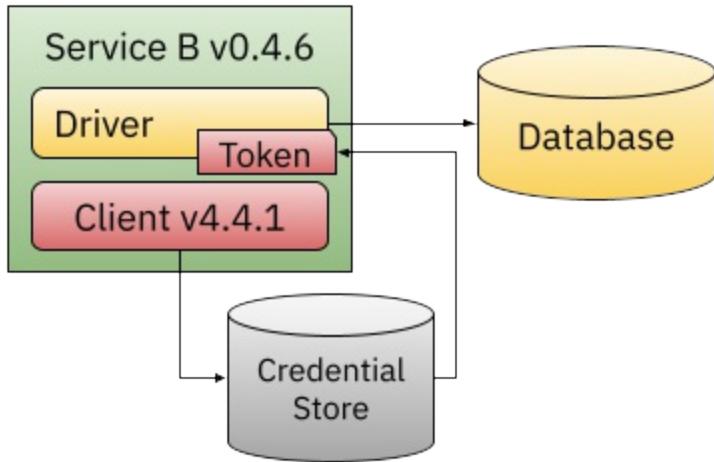
Once the application's service is connected to the database using the client APIs, it should be fairly easy to interact with it. Whether by sending SQL queries or commands to fetch data or using a Key-Value API to read keys and values from the database instance, developers should know the basics to start reading and writing data.

Do you have more than one service interacting with the same database instance? Are they both using the same library and the same version? Are these services written using the same programming language and frameworks? Even if you manage to control all these dependencies, there is still a coupling that will slow you down. Whenever the operations teams decide to upgrade the database version, each service connecting to this instance might or might not need to upgrade its dependencies and configuration parameters. Would you upgrade the database first or the dependencies?

For credentials, we face a similar issue. It is quite common to consume credentials from a credential store like Hashicorp's Vault (<https://www.vaultproject.io/>). If not provided by the platform and not managed in Kubernetes, application services can include a dependency to consume credentials from their application's code easily.

Figure 8.x shows Service B connecting to a Credential Store, using a specific client library, to get a Token to be able to connect to a Database.

Figure 8.x Credentials store dependencies



In Chapter 2 and Chapter 5, we connected the Conference services to different components using Kubernetes Secrets. By using Kubernetes Secrets, we were removing the need for application developers to worry about where to get these credentials from.

Otherwise, if your service connects to other services or components that might require dependencies in this way, the service will need to be upgraded for any change in any of the components. This coupling between the service code and dependencies creates the need for complex coordination between application development teams, the platform team, and the operations teams in charge of keeping these components up and running.

Can we get rid of some of these dependencies? Can we push some of these concerns down to the platform team so we remove the hassle of keeping them updated from developers? If we decouple these services with a clean interface, infrastructure, and applications can be updated independently.

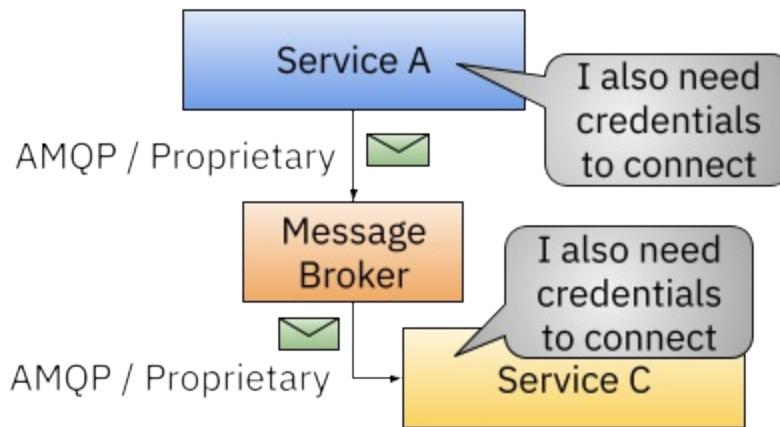
Let's look at how these challenges relate to asynchronous messaging before jumping into some alternatives.

Asynchronous Messaging challenges

With asynchronous messaging, you want to decouple the producer from the consumer. When using HTTP or GRPC, Service A needs to know about Service B. When using asynchronous messaging, Service A doesn't know

anything about Service C. You can take it even further, Service C might not even run when Service A places a message into the Message Broker. Figure 8.x shows Service A placing a message into the Message Broker, at a later point in time Service C can connect to the Message Broker and fetch messages from it.

Figure 8.x Asynchronous messaging interactions



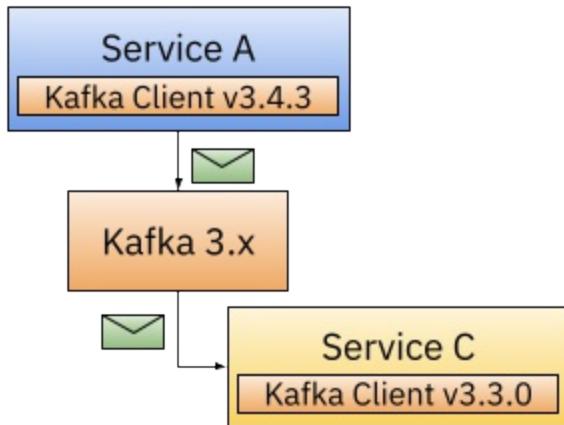
Similarly to HTTP/GRPC service-to-service interactions, for using a Message Broker, we need to know where the message broker is to send messages to or to subscribe to get messages from. Message Brokers also provide isolation to enable applications to group messages together using the concept of topics. Services can be connected to the same Message Broker instance but send and consume messages from different topics.

When using Message Brokers, we face the same issues described with databases. We need to add a dependency to our applications depending on which Message Broker we have decided to use, its version, and the programming language that we have chosen. Message brokers will use different protocols to receive and send information. A standard increasingly adopted in this space is the CloudEvent specification (<https://cloudevents.io/>) from the CNCF. While CloudEvents is a great step forward, it doesn't save your application developers from adding dependencies to connect and interact with your message brokers.

Figure 8.x shows Service A including the Kafka Client library to connect to Kafka and send messages. Besides the URL, port and credentials to connect

to the Kafka Instance, the kafka client also receives configurations on how the client will behave when connecting to the broker, similarly to databases. Service C uses the same client, but with different version, to connect to the same broker.

Figure 8.x Dependencies and APIs challenges



Message brokers face the same issue as with Databases and persistent storage. But unfortunately, with message brokers, developers will need to learn specific APIs that might not be that easy initially. Sending and consuming messages using different programming languages present more challenges and cognitive load on teams without experience with the specifics of the message broker at hand.

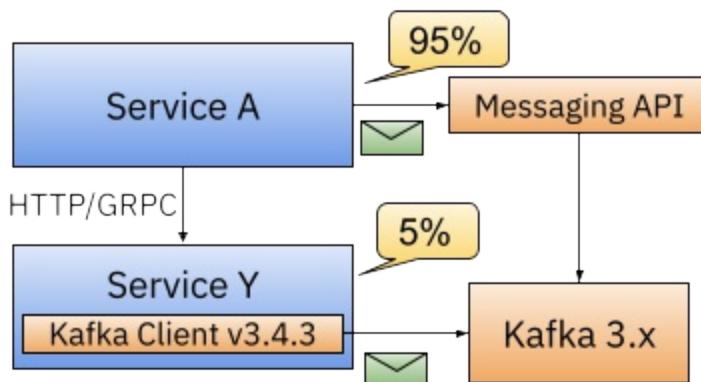
Same as with databases, if you have chosen Kafka, for example, it means that Kafka fits your application requirements. You might want to use advanced Kafka features that other Message Brokers don't provide. However, let me repeat it here; we are interested in 95% of the cases where application services want to exchange messages to externalize the state and let other interested parties know. For those cases, we want to remove the cognitive load from our application teams and let them emit and consume messages without the hassle of learning all the specifics of the selected message broker.

Dealing with edge cases (the remaining 5%)

There is always more than one good reason to add libraries to your

application's services. Sometimes these libraries will give you the ultimate control over how to connect to vendor-specific components and functionalities. Other times we add libraries because it is the easiest way to get started or because we are instructed to do so. Someone in the organization decided to use PostgreSQL, and the fastest way to connect and use it is to add the PostgreSQL driver to our application code. We usually don't realize that we are coupling our application to that specific PostgreSQL version. For edge cases, or to be more specific, scenarios where you need to use some vendor-specific functionality, I would ask you to consider wrapping up that specific functionality as a separate unit from all the generic functionality you might be consuming from a database or message broker.

Figure 8.x Common vs. edge cases encapsulation



I've chosen to use async messaging as an example in Figure 8.x, but the same applies to databases and credential stores. If we can decouple 95% of our services to use generic capabilities to do their work and encapsulate edge cases as separate units, we reduce the coupling and the cognitive load on new team members tasked to modify these services. Service A in Figure 8.x is consuming a Message API provided by the Platform Team to consume and emit messages asynchronously. We will look deeper into this approach in the next section. But more importantly, the edge cases, where we need to use some Kafka-specific features (for example) are extracted into a separate service that Service A can still interact with using HTTP or GRPC. Notice that the Messaging API also uses Kafka to move information around. Still, for Service A, that is no longer relevant, as a simplified API is exposed as a platform capability.

When we need to change these services, 95% of the time, we don't need team members to worry about Kafka. The "Messaging API" removes that concern from our application development teams. For modifying Service Y, you will need Kafka experts, and the Service Y code will need to be upgraded if Kafka is upgraded because it directly depends on the Kafka Client.

For this book, Platform Engineering teams should focus on trying to reduce the cognitive load on teams for the most common cases while at the same time allowing teams to choose the appropriate tool for edge cases and specific scenarios that don't fit the common solutions.

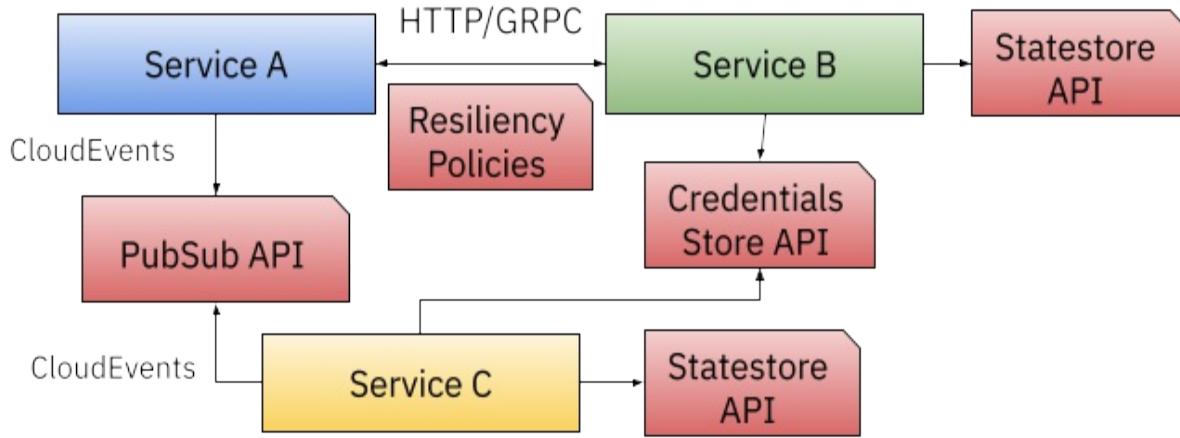
The following section will look at some approaches to address some of the challenges we have been discussing. However, keep in mind that these are generic solutions, and further steps may be required within your own specific context

8.2 Standard APIs to separate applications from infrastructure

What about if we encapsulate all these common functionalities (storing and reading data, messaging, credential stores, resiliency policies) into APIs that developers can use from within their applications to solve common challenges while, at the same time, enabling the Platform team to wire infrastructure up in a way that doesn't require applications code to change?

In Figure 8.x we can see the same services but now instead of adding dependencies to interact with infrastructure they use HTTP/GRPC requests instead.

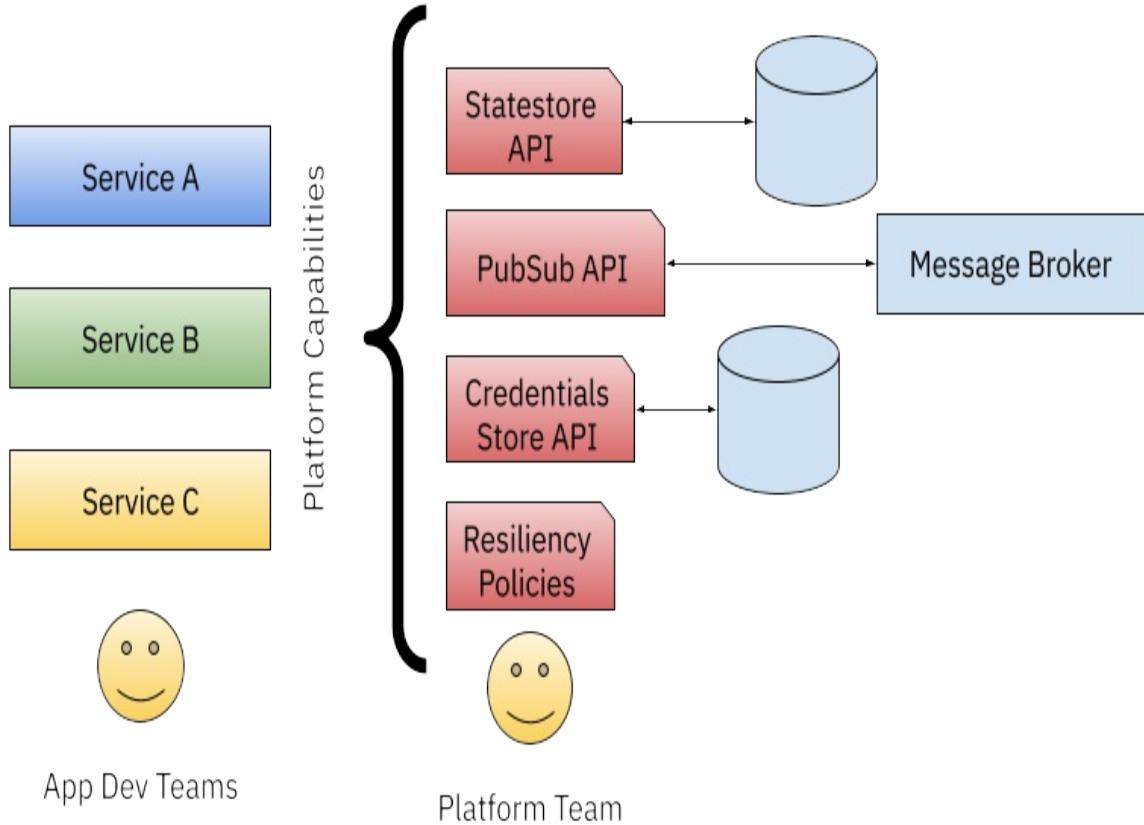
Figure 8.x Platform Capabilities as APIs



Suppose we expose a set of HTTP/GRPC APIs that our applications services can consume. In that case, we can remove vendor-specific dependencies from our application code and consume these services using standard HTTP or GRPC calls.

This separation between application services and platform capabilities enables separate teams to handle different responsibilities. The platform can evolve independently from applications, and application code will now only depend on the platform capabilities interfaces but not the version of the components running under the hood. Figure 8.x shows the separation between application code (our three services) managed by application development teams and platform capabilities managed by the platform team.

Figure 8.x Decoupling responsibilities from App Dev Teams and Platform Capabilities



When using an approach like the one suggested here, the Platform Team can expand the platform capabilities, introducing new services for application development teams. More importantly, they can do so without impacting the existing applications or forcing them to release new versions. This enables teams to decide when to release new versions of their services based on their features and the capabilities that they want to consume.

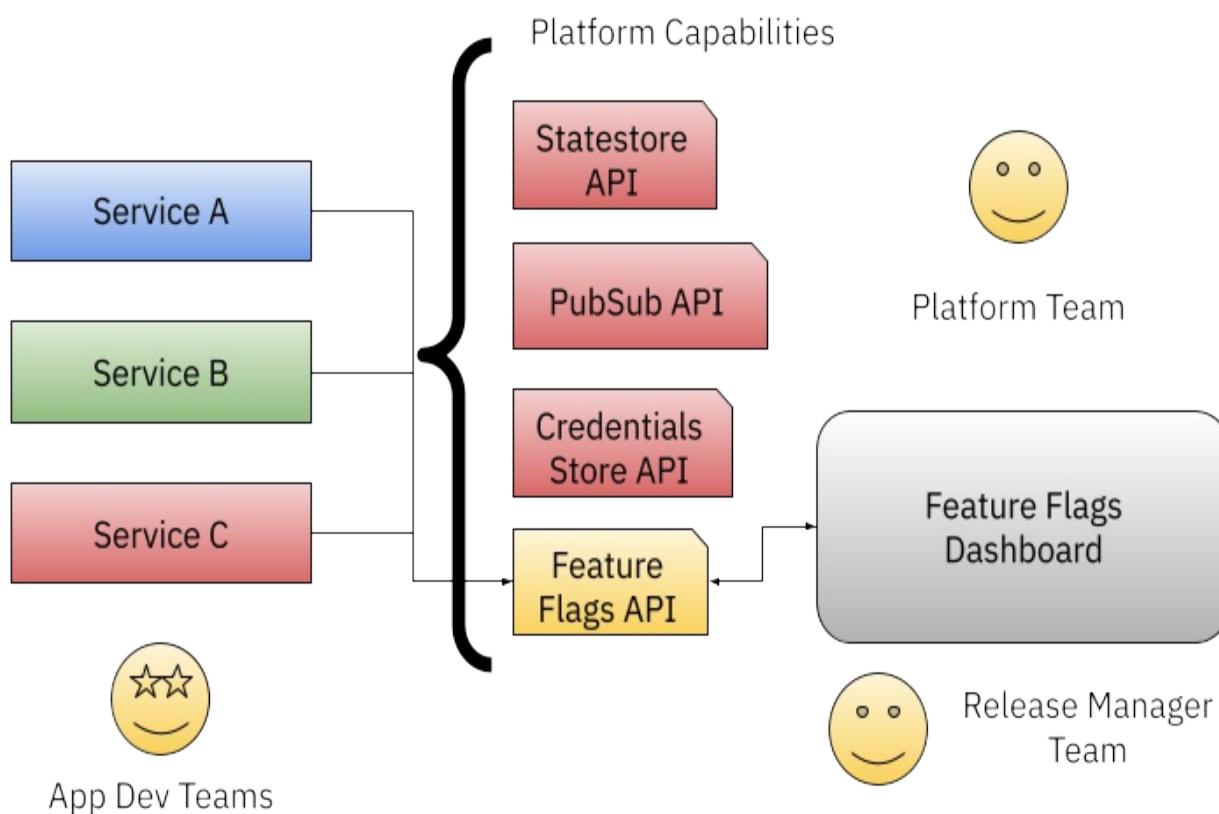
By following this approach, the Platform team can make new capabilities available for Services to use and promote best practices. Because these platform capabilities are accessible to all services, they can promote standardization and implement best practices behind the covers. Each team can decide which capabilities are needed to solve their specific problems based on the available ones.

For the sake of argument, imagine that the Platform team decides to expose a consistent Feature Flagging capability to all the services. Using this capability, all services can consistently define and use feature flags without

adding anything to their code except the feature flag conditional checks. Teams then can manage, visualize and toggle on and off all their flags consistently. A capability like Feature Flags introduced and managed by the platform team directly impacts developers' performance, as they don't need to worry about defining how feature flags will be handled under the hood (persistence, refresh, consistency, etc.), and they know for sure that they are doing things aligned with other services.

Figure 8.x shows how the platform team can add extra capabilities, like for example feature flags, directly enabling teams to use this new capability uniformly in all the services. No new dependencies needed.

Figure 8.x Enabling teams by providing consistent and unified capabilities such as feature flags



Before moving forward, a word of caution. Let's look at some challenges that you will face when externalizing capabilities as APIs, as suggested in the previous figure.

Exposing platform capabilities challenges

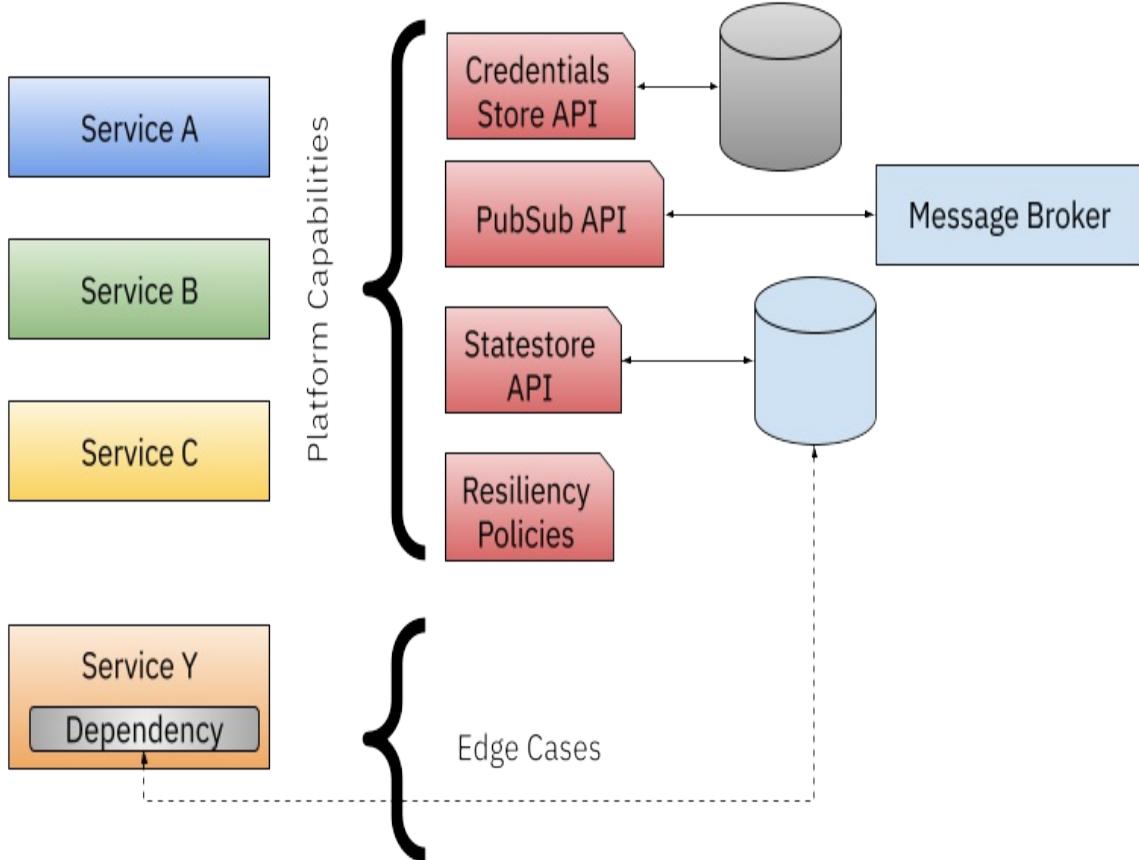
Externalizing APIs for teams to use will require, first of all, stable contracts that Application Teams can trust. When these APIs change, all applications consuming those APIs will break and must be updated.

One of the main advantages of adding dependencies to your application code and, for example, using containers is that for local development, you can always start a PostgreSQL instance using Docker or Docker Compose and connect your application locally to it. If you move towards platform-provided capabilities, you must ensure that you can provide a local development experience for your teams unless your organization is mature enough to always work against remote services.

Another big difference is that the connection between your Services and the Platform provided APIs will introduce latency and require security by default. Before, calling the PostgreSQL driver APIs was a local call in the same process as your application. HTTPS, or a secure protocol, established the connection to the Database itself, but setting that secure channel between your application and the database was the responsibility of the operations team.

It is also essential to recognize all the edge cases we can find when applying this approach to real-life projects. If you want to build these platform capabilities and push for your teams to consume them, you need to make sure that there is always a door open for edge cases, that doesn't force teams (or even the Platform Team) to make the common cases more complex in favor of an obscure feature that will be used only 1% of the time. Figure 8.x shows Service A, B and C using the capabilities exposed by the platform via the capabilities APIs. Service Y, on the other hand, has very specific requirements on how to connect to the database and the team maintaining the service has decided to bypass the Platform Capabilities APIs to connect directly to the database using the database client.

Figure 8.x Handling edge cases, do not ignore them



Treating edge cases separately allows Service A, B and C evolve separately from the platform components (database, message brokers, credential stores) while Service Y is now heavily depending on the Database that is connecting to with a specific version of the client. While this sound bad, in practice it is totally acceptable and it should be considered as a feature of the platform.

The following section will examine a couple of CNCF initiatives that took these ideas forward and helped us implement the platform capabilities that most of our applications require.

8.3 Providing Application-Level Platform Capabilities

In this last section, I wanted to cover two projects that will save us quite some time in standardizing these generic APIs that most of our applications

will need. We will start by looking at the Dapr Project, what it is, how it works, and how we can apply it to our Conference Application. Then we will look into Open Feature, a CNCF initiative that provides our applications with the right abstractions to define and use feature flags without being tied to a specific feature flag provider.

Once we get a bit of an understanding of how these two projects work and complement each other by helping us to provide application-level platform capabilities, we will look into how these projects can be applied to our Conference Application, what changes are needed, the advantages of following this approach and some examples showing edge cases.

Let's start with Dapr, our Distributed Application Runtime.

8.3.1 Dapr in action

Dapr provides a set of consistent APIs to solve common and recurrent distributed application challenges.

The Dapr project has spent a significant amount of time implementing a set of APIs (called components) to abstract away common challenges and best practices that distributed applications will need 95% of the time. Created by Microsoft and donated to the CNCF in 2021, the Dapr project has a large community contributing with extensions and improvements to the project APIs, making it the 9th fastest growing project in the CNCF.

Dapr defines a set of components that provide concrete APIs to solve different challenges and swappable implementations that the Platform team can define which to use. If you visit the <https://dapr.io> website you will see the list of components including Publish & Subscribe, Service Invocation, Secret Management, Input/Output Bindings, State Management and Virtual Actors.

Figure 8.x Dapr components for building distributed applications

Publish & Subscribe

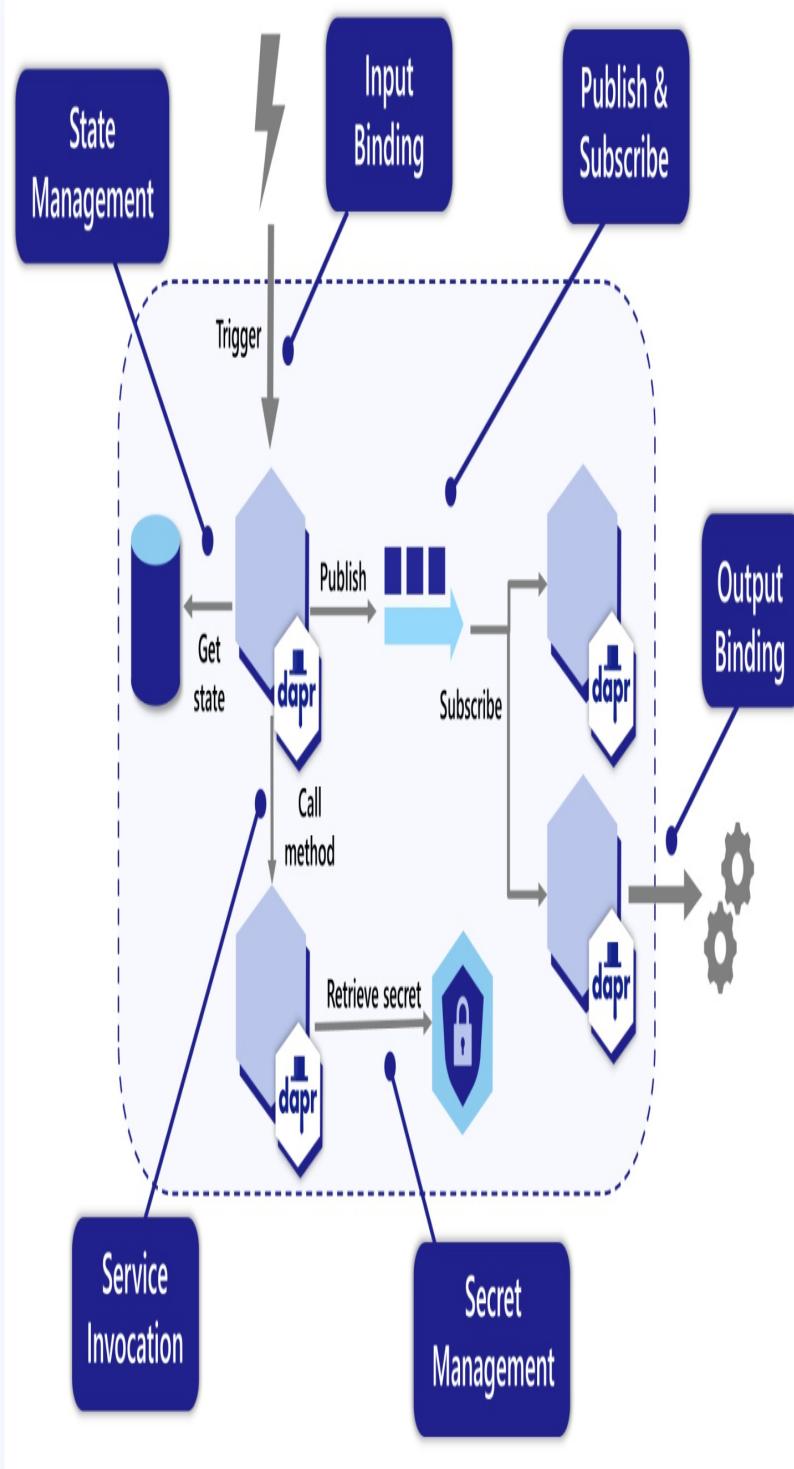
↑↓ Service Invocation

₩ Secret Management

⇄ Input/Output Bindings

⌚ State Management

■■■ Virtual Actors



While Dapr does much more than just exposing APIs, in this chapter, I wanted to focus on the APIs provided by the project and the mechanisms used by the project to enable applications/services to consume these APIs.

Because this is a Kubernetes book, we will look at Dapr in the context of Kubernetes, but the project can also be used outside of Kubernetes Clusters, making Dapr a generic tool to build distributed applications no matter where you are running them.

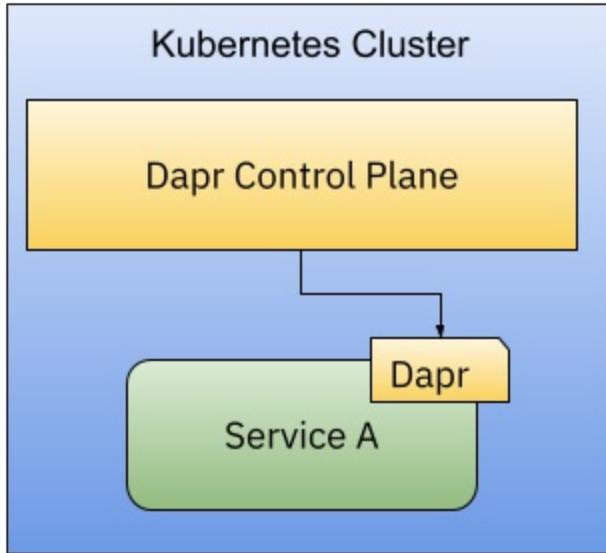
Dapr in Kubernetes

Dapr works as a Kubernetes extension or add-on. In the same way, we did with Knative Serving in the previous chapter. You must install a set of Dapr controllers (Dapr Control Plane) on your Kubernetes Clusters. Figure 8.x shows Service A deployed in a Kubernetes Cluster that has Dapr installed in it. Service A is also annotated with the following annotation:

```
`dapr.io/enabled: "true"'
```

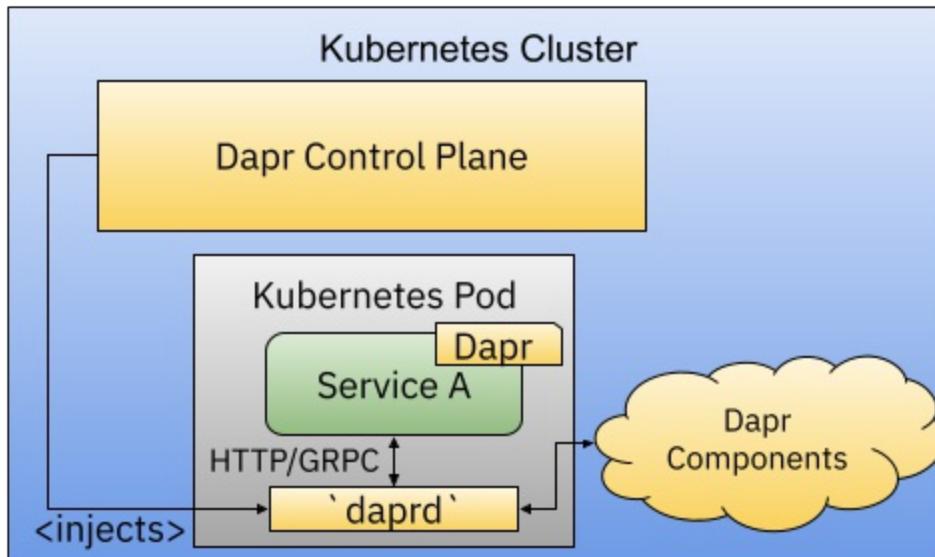
Once Dapr is installed in your clusters, your applications deployed in the cluster can start using the Dapr Component APIs by adding a set of annotations to your workloads. This enables the Dapr controllers to understand that your application wants to use Dapr Components.

Figure 8.x The Dapr control plane monitor for applications with Dapr annotations



By default, Dapr will make all the Dapr Component APIs available to your applications/services as a sidecar (`daprd` is the container that will run beside your applications/services) that runs beside your application's containers. Using the sidecar pattern, we enable our application to interact with a co-located API that runs very close to the application's container and avoids network round trips. Figure 8.x shows how the Dapr Control Plane injects the `daprd` sidecar to the application annotated with the Dapr annotations. This enables the application to access the configured Dapr Components.

Figure 8.x Dapr sidecars (`daprd`) give your applications local access to Dapr Components



Once the Dapr sidecar is running beside your applications’/service container can start making use of the Dapr Component APIs by sending requests (using HTTP or GRPC) to `localhost`, as the `daprd` sidecar runs inside the same Pod as the application, hence sharing the same networking space.

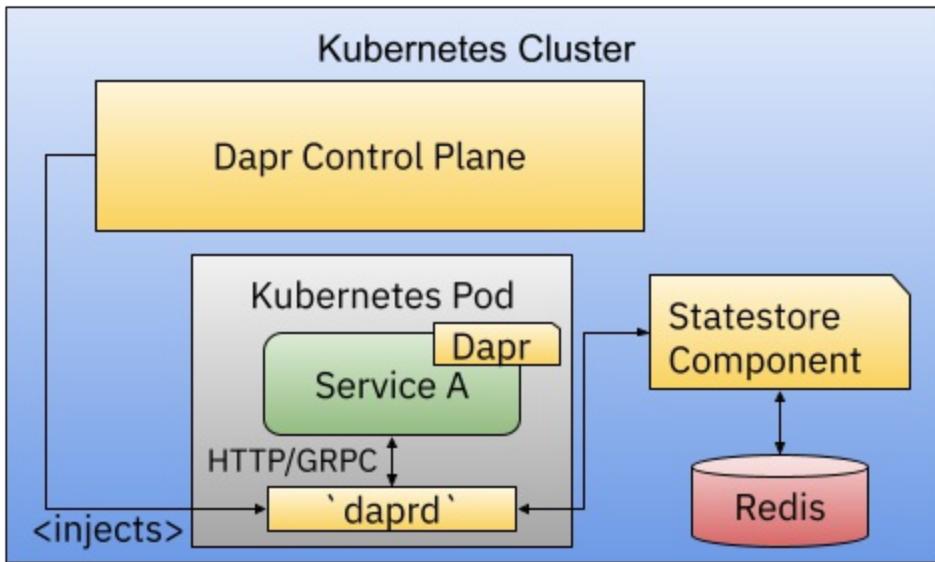
Now for the Dapr Component APIs to be of some use, the Platform team needs to configure the implementation (or backing mechanisms) for these APIs to work. For example, if you want to use the Statestore component APIs from your applications/services you will need to define and configure a Statestore component.

In Dapr, you configure Components using Kubernetes Resources. For example, configuring a Statestore component that uses Redis as the implementation will look like this:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: statestore
spec:
  type: state.redis
  version: v1
  metadata:
    - name: keyPrefix
      value: name
    - name: redisHost
      value: redis-master:6379
    - name: redisPassword
      secretKeyRef:
        name: redis
        key: redis-password
auth:
  secretStore: kubernetes
```

If the component resource is available in the Kubernetes Cluster, the `daprd` sidecar can read its configurations and connect to the Redis instance for this example. From the application perspective, there is no need to know if Redis is being used or if any other implementation for the Statestore component. Figure 8.x shows how Dapr Components are wired so Service A can use the Statestore component APIs. For this example, Service A by calling a local API will be able to store and read data from the Redis Instance.

Figure 8.x Dapr sidecars uses Component configurations to connect to the component's implementations

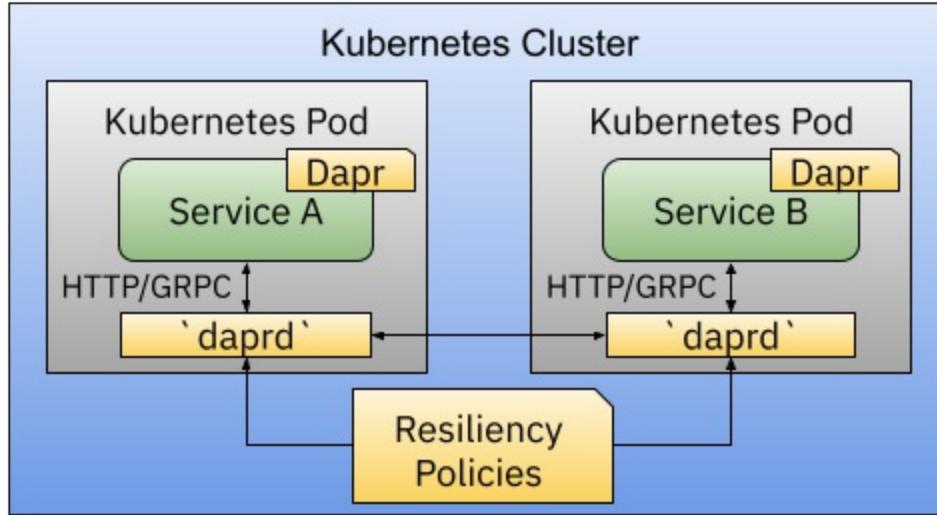


Dapr makes it easy to build your application using a local/self-hosted Redis Instance but then move it to the Cloud where a managed Redis service can be used. No code or dependencies changes are needed, just a different Dapr Component configuration.

Do you want to emit and consume messages between different applications? You just need to configure a Dapr PubSub Component and its implementation. Now your service can use a local API to emit asynchronous messages. Do you want to make service-to-service calls but leverage Dapr resiliency policies to avoid writing custom logic inside your application code? Just use the Dapr service to service invocation APIs and then provide Dapr resiliency policies configurations according to your needs.

Figure 8.x shows how Service A and Service B can send requests to each other but using the Service Invocation APIs, in contrast to calling the other service directly. By using these APIs (that send traffic through the `daprd` sidecar) enable the platform team to configure resiliency policies at the platform level, in an uniform way, without adding any dependencies or changing the application code.

Figure 8.x Dapr-enabled services can leverage service-to-service communications and resiliency policies



Ok, so the Dapr control plane will inject the Dapr sidecars (`daprd`) to the applications that are interested in using Dapr Components. But how does this look from the application point of view?

Dapr and your applications

If we go back to the example introduced in the previous section where Service A wants to use the Statestore Component to store/read some data from persistent storage like Redis, the application code is straightforward. No matter which programming language you use, as soon as you know how to create HTTP or GRPC requests, you have all you need to work with Dapr.

For example, to store data using the Statestore APIs your application code need to send an HTTP/GRPC request to the following endpoint

``http://localhost:<DAPR_PORT>/v1.0/state/<STATESTORE_NAME>``

Using `curl` the request will look like this, where `-d` shows the data we want to persist and 3500 is the default DAPR_PORT and our Statestore component is called `statestore`

```
> curl -X POST -H "Content-Type: application/json" -d '[{"key":
```

To read the data that we have persisted, instead of sending a POST request, we just write a GET request. With `curl`, it would look like this:

```
curl http://localhost:3500/v1.0/state/statestore/name
```

But of course, you will not be using `curl` from inside your applications. You will use your programming language tools to write these requests. So if you are using Python, Go, Java, .Net or Javascript, you can find tutorials online on how to use popular libraries or built-in mechanisms to write these requests.

Alternatively, and as an option, you can choose to use one of the Dapr SDKs (Software Development Kits) available for programming languages. Adding the Dapr SDK to your application as a dependency allows you to make your developers' life easier, so they don't need to craft HTTP or GRCP requests manually. It is crucial to notice that while you are now adding a new dependency to your application, this dependency is an option and only used as a helper to speed things up, as this dependency is not tied to any of the infrastructural components that the Dapr APIs are interacting with.

Check the Dapr website for examples of how your code will look if you use the Dapr SDK. For example, for a multi-programming language example on how to use the Statestore component using the SDKs, you can visit:
<https://docs.dapr.io/getting-started/quickstarts/statemanagement-quickstart/>

8.3.2 Feature Flags in action

Feature Flags enable teams to release software that includes new features without making those features available. New features can be hidden away behind feature flags that can be enabled later. In other words, feature flags allow teams to keep deploying new versions of their services or applications, and once these applications are running, features can be turned on or off based on the company's needs.

While most companies might build their mechanisms to implement feature flags, it is a well-recognized pattern to be encapsulated into a specialized service or library. In the Kubernetes world, you can think about using `ConfigMaps` as the simplest way to parameterize your containers. As soon as your container can read Environment Variables to turn on and off features, you are ready to go.

Unfortunately, this approach is too simplistic and doesn't work for real-world scenarios. One of the main reasons is that your containers will need to be restarted to reread the content of the ConfigMap. The second reason is that you might need tons of flags for your different services. Hence you might need multiple ConfigMaps to manage your feature flags. Third, if you use Environment Variables, you will need to come up with a convention to define the status of each flag, their default values, and maybe their type, as you cannot get away with just defining variables as plain strings.

Because this is a well-understood problem, several companies have come up with tools and managed services like LaunchDarkly, and Split among others, which enable teams to host their feature flags in a remote service that offers simplified access to view and modify feature flags without the need for technical knowledge. For each of these services, to fetch and evaluate complex feature flags, you will need to download and add a dependency to your applications. As each feature flag provider will offer different functionalities, switching between providers would require many changes.

Open Feature is a CNCF initiative to unify how feature flags can be consumed and evaluated in Cloud-Native applications. In the same way that Dapr is abstracting how to interact with Statestores or PubSub (async message brokers) components, Open Feature provides a consistent API to consume and evaluate feature flags no matter which features flag provider we use.

<https://openfeature.dev/>

In this short section, we will look at only the simplest example in Kubernetes, using a ConfigMap to hold a set of feature flag definitions. We will also be using the `flagd` implementation provided by OpenFeature, but the beauty of this approach is that you can then swap the provider where the feature flags are stored without changing any single line of code in your application.

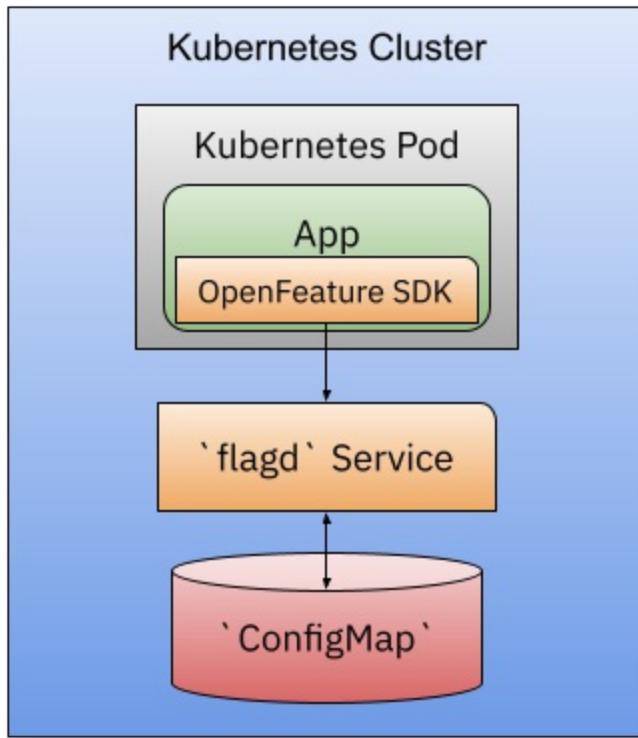
You can find a set-by-step tutorial at the following link:

<https://github.com/salaboy/from-monolith-to-k8s/tree/main/openfeature>

In this tutorial, we deploy a simple application that consumes feature flags

from the `flagd` Service. Figure 8.x shows the simple architecture of this example.

Figure 8.x Consuming and evaluating feature flags from our application services



For this simple example, our App is written in Go and uses the OpenFeature Go SDK to fetch feature flags from the `flagd` service. The `flagd` Service for this example is configured to watch a Kubernetes `ConfigMap` that contains some complex feature flags definitions.

While this is a simple example, it allows us to see how a service like `flagd` can allow us to abstract away all the complexities of the storage and implementation of the mechanisms needed to provide a feature flag capability as part of our platform.

8.3.3 Updating our Conference Application to consume application-level Platform capabilities

Conceptually and from a Platform perspective, it will be great to consume all

these capabilities without the Platform leaking which tools are being used to implement different behaviors. This would enable the Platform team to change/swap implementations and reduce the cognitive load from teams using these capabilities. But, in the same way, as we discussed with Kubernetes, understanding how these tools work, their behaviors, and how their functionalities were designed to influence how we architect our applications and services. In this last section of the chapter, I wanted to show how tools like Dapr and Open Feature can influence your application architecture and, at the same time, mention that tools like Dapr, Open Feature, and Knative offer the building blocks to create higher-level abstractions that can avoid teams to be aware that these tools are being used.

For our Conference Application, we can leverage the following Dapr components, so let's focus on these:

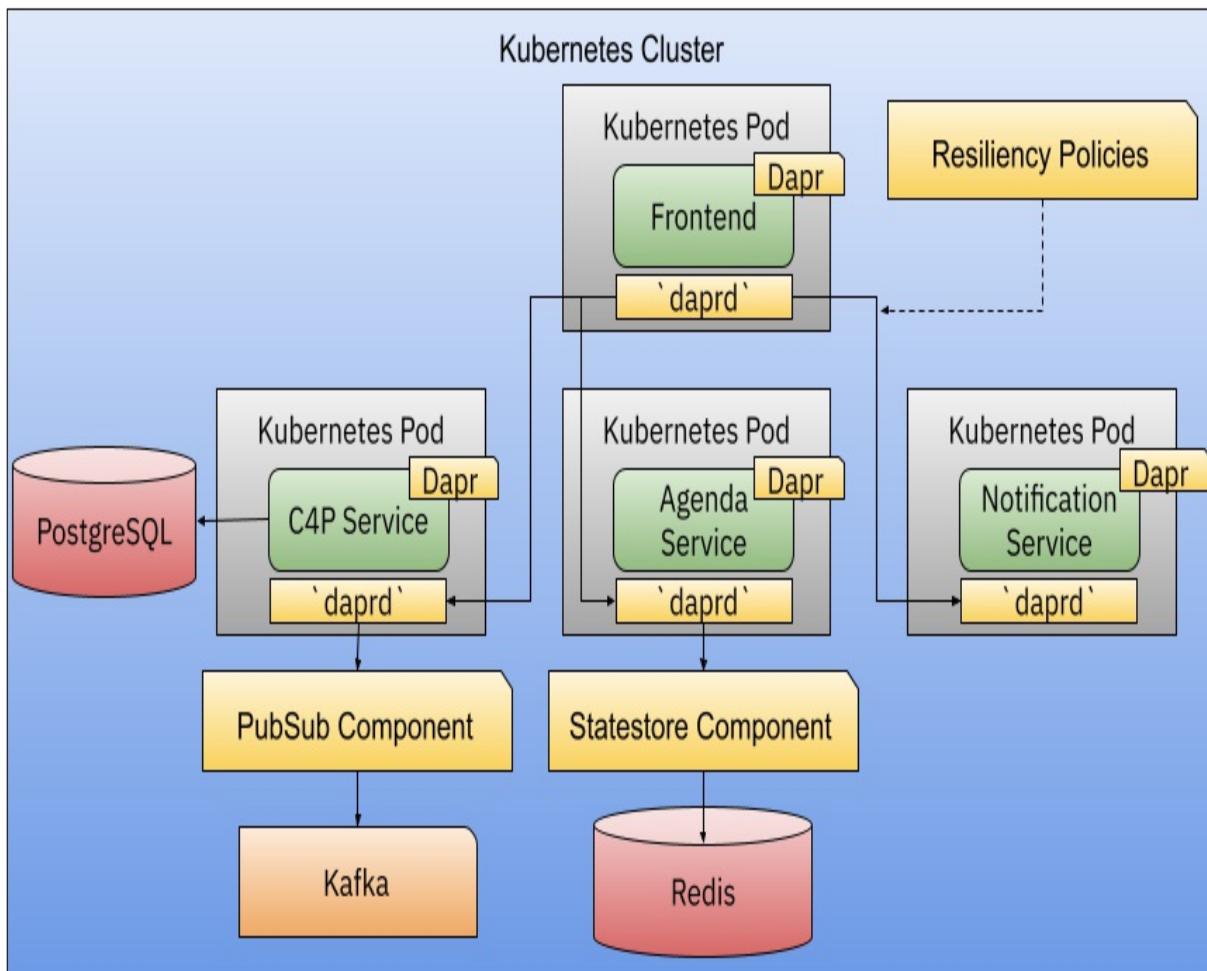
- **Dapr Statestore component:** using the Statestore component APIs will enable us to remove the Redis dependency from the Agenda Service included in the Conference Application. If for some reason, we want to swap Redis for another persistent store we will be able to do so without changing any of the application code.
- **Dapr PubSub component:** For emitting events, we can replace the Kafka client from all the services to use the PubSub component APIs, allowing us to test different implementations, such as RabbitMQ or a Cloud Provider services to exchange asynchronous messages between applications.
- **Dapr Service-to-Service invocations & Dapr Resiliency Policies:** if we use the service-to-service invocation APIs, we can configure resiliency policies between the services without adding any library or custom code to our services code.

While we can choose to use the Statestore component APIs to also remove the PostgreSQL dependency in our “Call for Proposals” Service, I have chosen not to do so to support the use of SQL and PostgreSQL features that the team needed for this service. When adopting Dapr, you must avoid an “all or nothing” approach.

Let's take a look at how the application, as a whole, will change if we decide to use Dapr. Figure 8.x shows the application services using Dapr

Components, as all the Services are annotated to use Dapr, the `daprd` sidecar has been injected to all services. A PubSub and Statestore components had been configured and can be accessed by the Call For Proposals Service, Agenda Service and Notifications Service. Resiliency Policies had been also configured and defined for the Frontend service to interact with all the backend services, making the Frontend more resilient to failure.

Figure 8.x Using Dapr Components for our Walking Skeleton / Conference Application



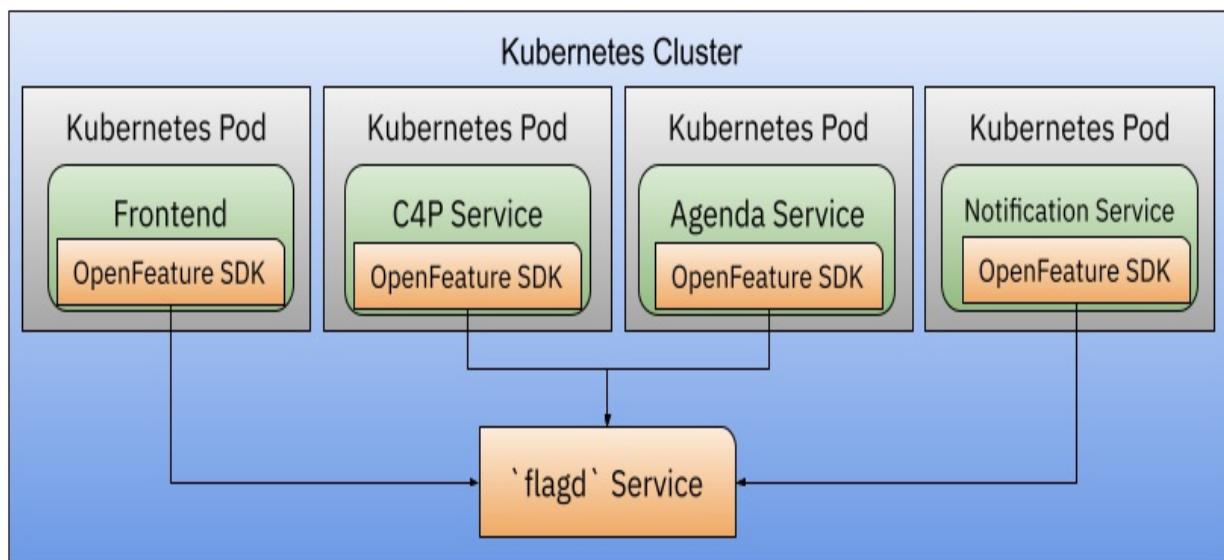
It is also important to notice that the application code needs to change slightly, as when services want to talk to each other, they need to use the Dapr API to be able to use Resiliency Policies. In Figure 8.x, you can see that the Frontend service interacts with all the backend services going through the

Dapr Service-to-Service invocation APIs.

Finally, because we wanted to enable all the services to use feature flags, each service now includes the Open Feature SDK, which allows the platform team to define which feature flag implementation all services will use.

In Figure 8.x each service has included the OpenFeature SDK library and configured to point to the `flagd` Service that enables the platform team to configure the mechanism used to store, fetch and manage all the feature flags used by all the services.

Figure 8.x Services using the `flagd` feature flag provider.



Using the OpenFeature SDK, we can change the feature flag provider without changing our application code. All the feature flag consumption and evaluation is now standardized by the OpenFeature SDK on our service code.

You can find a tutorial that expands on our Platform walking skeleton, which now uses Dapr and OpenFeature flags to enable application teams to keep changing the application services. At the same time, the Platform team is free to configure and wire up application infrastructure and define all the backing mechanisms and implementations for feature flags, storage, messaging, configuration, managing credentials, resiliency, and other common challenges they don't want to expose directly to developers.

8.4 Some final thoughts about platform capabilities

In these last two chapters, we have seen how we can enable teams with platform-wide capabilities. Providing teams with autoscaling, release strategies, and common APIs to solve everyday challenges they will face when building distributed applications and mechanisms such as feature flags, we aim to speed their process to write and deliver more software more efficiently.

For the sake of space, topics such as observability, metrics, and logs purposefully haven't been covered in these sections, as these capabilities are currently more mature and more operations-focused. I've decided to focus on capabilities that build on top of the operation and infrastructure teams to speed up development teams and solve everyday challenges. I do see tools like Knative, ArgoRollouts, Dapr, and OpenFeature leading the way in this space, and still, even if teams need to figure out all the details of each of these tools, patterns are emerging.

In the next chapter, to close the book, I've decided to talk about how we can measure the Platforms we are building on top of Kubernetes. The platform capabilities described in these last two chapters and the combination tools described so far in this book are as good as we are improving our team's velocity to deliver software. Hence using metrics that focus on how efficient our teams are in delivering software directly correlates with the tools offered by the platform for these teams to use.

8.5 Summary

- Removing dependencies to application infrastructure enable application code to stay agnostic to platform-wide upgrades. Separating the lifecycle of the applications and the infrastructure enable teams to rely on stable APIs instead of dealing with provider-specific clients and drivers for everyday use cases
- Treating edge cases separately allows experts to make more conscious cases based on their application requirements. This also allows common scenarios to be handled by less experienced team members, who don't

need to understand the specifics of tools like vendor-specific database features or low-level message broker configurations when they only want to store or read data or emit events from their application's code.

- Dapr solves common and shared concerns when building distributed applications. Developers that can write HTTP/GRPC requests can interact with infrastructure that the platform team will wire up.
- OpenFeature standardizes the way applications consume and evaluate feature flags. Relying on OpenFeature abstractions allows Platform teams to decide where feature flags are stored and how they are managed. Different providers can offer non-technical people dashboards where they can see and manipulate flags.

9 Measuring your platforms

This chapter covers

- The importance of measuring our continuous delivery performance
- DORA metrics and the secret of continuous improvement
- Collecting metrics using an event-driven approach using CloudEvents and CDEvents

In Chapter 8, we covered the principles of how to build a platform that helps you to deliver software and enables your teams to have the tools that they need when they need it. This last chapter is all about making sure that the platform is working, not only for application development teams but for the entire organization. To understand how the platform is performing, we need to be able to measure it. There are different ways of taking measurements on the software we run. Still, in this chapter, we will focus on the DORA (DevOps Research and Assessment) metrics, which provide a good foundation for understanding our organization's software delivery speed and how good we are at recovering from failures when they happen.

This chapter is divided into three main sections.

- What to measure: DORA metrics and high-performant teams
- How to measure our platform: CloudEvents and CDEvents to the rescue
- A platform that can measure itself is a healthy platform

Let's get started by understanding what we should be measuring, and for that, we will need to look at the DORA metrics.

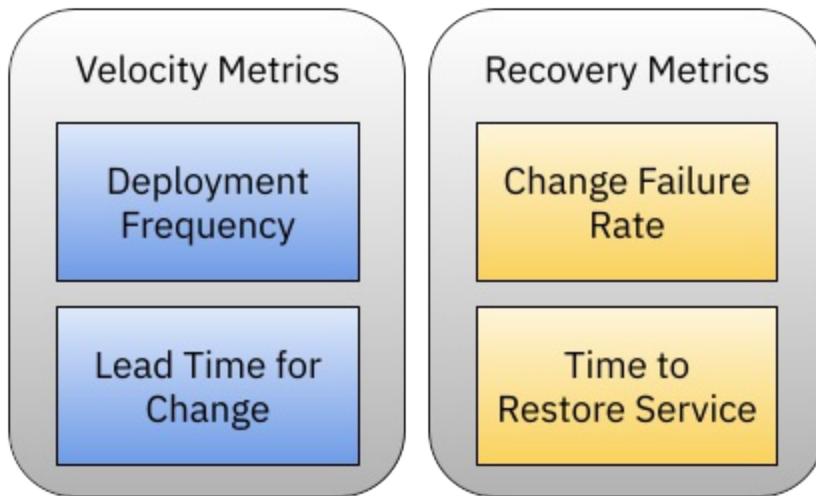
9.1 What to measure: DORA metrics and high-performant teams

The DevOps Research and Assessment (DORA) team, after performing thorough research in the industry, has identified five key metrics that

highlight the performance of software development teams delivering software. Initially, in 2020, only four keys were defined, so you might find references to the “DORA four keys” metrics. After surveying hundreds of teams, they found out which indicators/metrics separated high-performant / elite teams from the rest, and the numbers were quite shocking. They used the following four keys to rank teams and their practices:

- **Deployment Frequency:** how often an organization successfully releases software in front of their customers
- **Lead Time for Change:** the time that it takes a change produced by an application team to reach live customers
- **Change Failure Rate:** the number of issues that are created by new changes being introduced to our production environments
- **Time to Restore Service:** how long takes to recover for an issue in our production environments.

Figure 9.1 DORA metrics by category



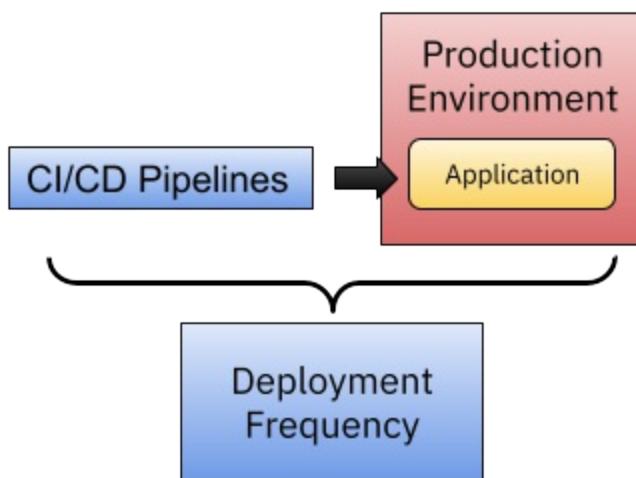
In 2022, a fifth key metric was introduced focused on reliability to make sure that operational performance is covered too. We will only focus on the four software delivery metrics, as this book focuses on application development teams and not operation teams.

These five key metrics, as shown in the reports, establish a clear correlation between high-performing teams and their velocity expressed by these

metrics. If you manage your teams to reduce their deployment frequency, that is, how often they deploy new versions in front of your users and reduce the time caused by incidents, your software delivery performance will increase.

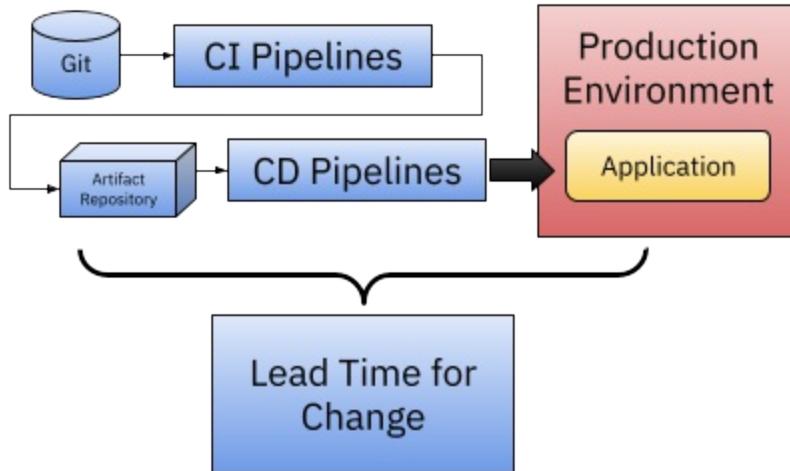
In this chapter, we will be looking at how to calculate these metrics for the platforms that we are building to make sure that these platforms are improving our continuous delivery practices. To be able to collect data and calculate these metrics, you will need to tap into different systems that your teams are using to deliver software. For example, if you want to calculate **Deployment Frequency**, you will need access to data from the production environment every time a new release is deployed. Another option would be to use data from the Environment pipelines performing the releases to our production environment.

Figure 9.2 Deployment frequency data sources



If you want to calculate **Lead Time for Change** you will need to aggregate data coming from your source code version control system like GitHub and have a way to correlate this information with the artifacts that are being deployed into production.

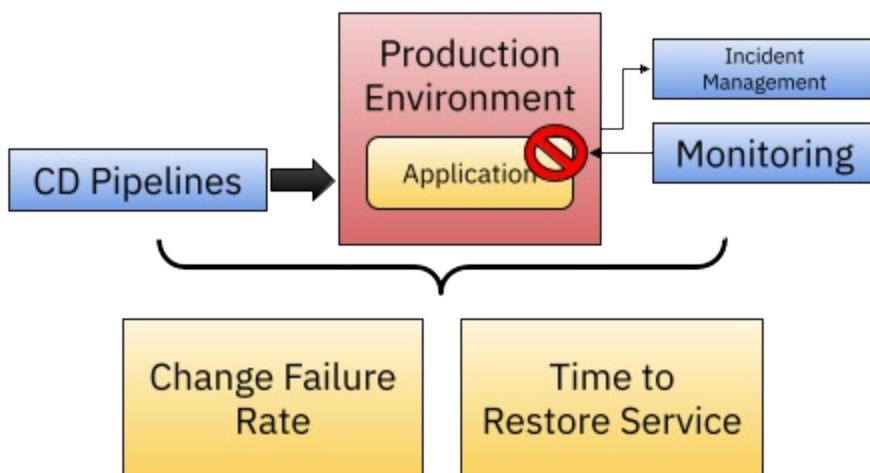
Figure 9.3 Lead Time for change data sources



Suppose you have a straightforward way to correlate commits to artifacts and later on to deployments. In that case, you can rely on a few sources, but if you want to have a more detailed understanding of where the bottlenecks are, you might choose to aggregate more data to be able to see where time is being spent.

To calculate **Change Failure Rate** and **Time to Restore Service**, you might need to tap into an incident management tool and monitoring tools.

Figure 9.4 Recovery metrics data sources



For recovery metrics (Change Failure Rate and Time to Restore Service),

data collection can be more challenging as we need to find a way to measure the time when the application performance is degraded or when there is downtime. This might involve reports from actual users experiencing issues with our applications.

The integration problem

This quickly becomes a system integration challenge. In general terms, we need to observe the systems that are involved in our software delivery process, capture relevant data and then have the mechanisms to aggregate this information. Once we have this information available, we can use these metrics to optimize our delivery processes and find and solve bottlenecks.

While some projects are already providing DORA metrics out of the box, you will need to evaluate if they are flexible enough to plug your systems into them. The Four Keys project by Google provides an out-of-the-box experience to calculate these metrics based on external outputs. You can read more about it here: <https://cloud.google.com/blog/products/devops-sre/using-the-four-keys-to-measure-your-devops-performance>.

Unfortunately, the Four Keys project requires you to run on the Google Cloud Platform as it uses BigData and Google Cloud run to do the calculations. Following the principles of this book, we need a solution that works across Cloud Providers and uses Kubernetes as the baseline. Other tools like LinearB (<https://linearb.io/>) offer a SaaS solution to track different tools, and I recommend this blog post by Codefresh which also explains the challenges of calculating these metrics and the data points that you will need to do so: <https://codefresh.io/learn/software-deployment/dora-metrics-4-key-metrics-for-improving-devops-performance/>.

To have a Kubernetes-native way to calculate these metrics, we need to standardize in a way how we consume information from different systems, transform this information into a model that we can use to calculate these metrics, and make sure that different organizations can extend this model with their metrics and their very diverse sources of information.

In the next section, we will look at two standards that can help us with this

mission: CloudEvents and CDEvents. We have seen CloudEvents already in Chapter 6, but here we will look at them from a more practical perspective and not go into routing them. We will focus more on transforming them to serve our purpose.

9.2 How to measure our platform: CloudEvents and CDEvents

More and more tools and service providers are adopting CloudEvents (<https://cloudevents.io>) as a standard way to emit and consume events from other sources. In this book, we have covered Tekton (<https://tekton.dev>) and Knative Eventing (<https://knative.dev>), two open-source projects that provide native support for CloudEvents, but if you look into the official CloudEvents website, you can find all the projects that already support the standard. I find it interesting to see Cloud Provider services such as GoogleCloud Eventarc (<https://cloud.google.com/eventarc/docs>) or Alibaba Cloud EventBridge (<https://www.alibabacloud.com/help/en/eventbridge>) in the list, which indicates that CloudEvents are here to stay.

While seeing more adoption is an excellent indicator, there is still much work to do when you receive or want to emit a CloudEvent. We must remember here that CloudEvents are just very simple and thin envelopes for our events data. This means that if we want to use these events to calculate metrics, we will need to open the envelope, read the data, and based on that, aggregate and correlate these events together.

This has proven challenging, as each tool that generates CloudEvents can define its schemas for the CloudEvent payload. We would need to understand how each system is encoding the payload to be able to extract the data that we need to calculate our metrics. Wouldn't it be great to have some kind of standard model to quickly be able to filter and consume these events based on what these events mean for our Continuous Delivery needs?

Welcome CEvents: <https://cdevents.dev>

CloudEvents for Continuous Delivery: CDEvents

CDEvents are just CloudEvents, but with a more specific purpose. They map to different phases of our continuous delivery practices. CDEvents are an initiative that the Continuous Delivery Foundation (<https://cd.foundation>) drives, and as its website defines, they focus on enabling interoperability across different tools that are related to Continuous Delivery.

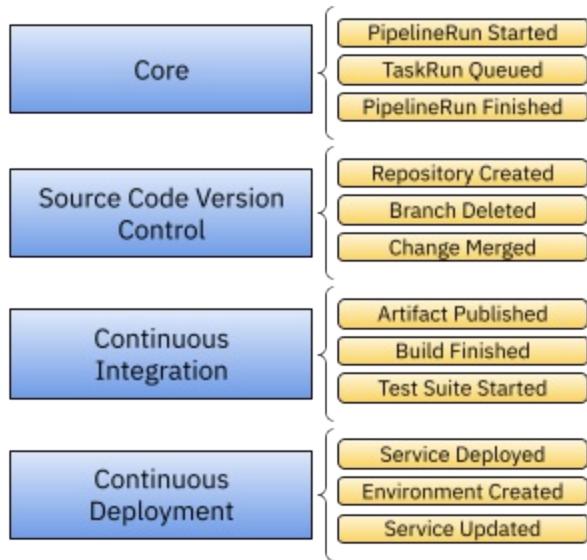
“CDEvents is a common specification for Continuous Delivery events, enabling interoperability in the complete software production ecosystem.”

-- <https://cdevents.dev>

To provide interoperability, the CDEvents specification defines four stages (<https://github.com/cdevents/spec/blob/v0.1.0/spec.md#vocabulary-stages>). These stages are used to group events that are conceptually related to different phases and tools in our software delivery ecosystem:

- **Core:** events related to the orchestration of tasks usually coming from pipeline engines. Here you will find the specification of the events around the subjects “taskRun” and “pipelineRun”. Events like “PipelineRun started”, or “TaskRun queued” can be found at this stage.
- **Source Code Version Control:** events related to changes associated with your source code, the specification focuses on covering the subjects: “repository”, “branch” and “change”. Events like “Change created” or “Change Merged” can be found at this stage.
- **Continuous Integration:** events related to building software, producing artifacts, and running tests. This stage covers the subjects “artifact”, “build”, “testCase” and “testSuite”. Events like “Artifact published” or “Build finished” can be found at this stage.
- **Continuous Deployment:** events related to deploying software in different environments. The subjects covered in this stage are “services” and “environments”. Events like “Service deployed” or “Environment modified” can be found at this stage.

Figure 9.5 The four stages defined by the CDEvents specification

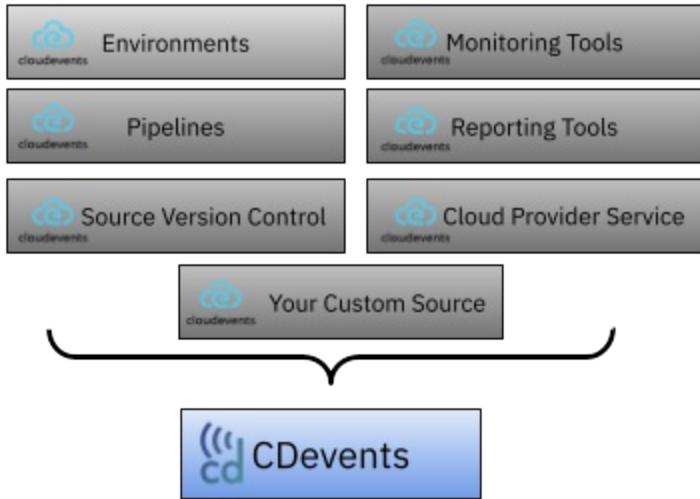


We can easily use CDEvents to calculate our software delivery metrics, as they are already covering the subjects that these metrics are interested in. For example, we can use events from the Continuous Deployment stage to calculate the **Deployment Frequency** metric. We can combine Continuous Deployment events and Source Code Version Control events to calculate **Lead Time for Change**.

The question then becomes, where do we get CDEvents from? CDEvents is a much newer specification that is currently being incubated at the CD Foundation, and it is my firm belief that as part of the interoperability story, this specification can serve as a hook mechanism for different tools and implementations to map their tools to a standard model that we can use to calculate all these metrics while allowing legacy systems (and tools that are not emitting cloud events) to benefit from them too.

For the purposes of this chapter, we will use the CDEvents specification to define our standardized data model. We will collect information from a wide range of different systems using CloudEvents and rely on CDEvents to map the incoming events into the different stages of our software delivery practice.

Figure 9.6 CDEvents are more specialized CloudEvents for Continuous Delivery



Tools like Tekton are already providing experimental support for CDevents (<https://www.youtube.com/watch?v=GAm6JzTW4nc>) and as we will see in the next section, we can transform CloudEvents into CDevents using functions. More importantly, the CDevents Working Group is also focused on providing SDK (software development kits) in different languages so you can build your own applications that consume and emit CDEvents no matter the programming language that you are using.

In the next section, we will examine how a Kubernetes-based solution for calculating DORA metrics can be built and extended to support different metrics and event sources. This is important to ensure that different platforms using different tools can leverage their performance and detect early bottlenecks and improvement points.

9.3 A platform that can measure itself is a healthy platform

To be able to calculate the metrics proposed by the DORA team (Deployment Frequency, Lead Time for Change, Change Failure Rate, Time to Restore Service) we need to collect data. Once we have the data coming from different systems, we need to transform the data into a standardized model that we can use to calculate the metrics. Then we need to process the data to calculate the values for each metric. We need to store the results of these calculations, and then we need to make them available to everyone interested,

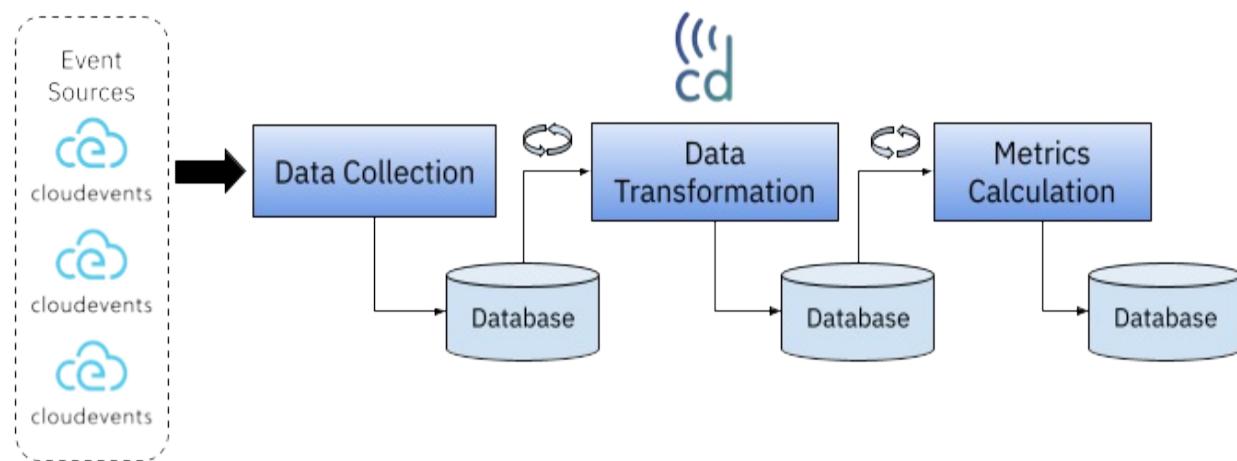
probably using a graphical dashboard that summarizes the data collected and the calculated metrics.

Different tools can be used to build this data collection, transformation, and aggregation pipeline. Still, to build a simple yet extensible solution, we will use the same tools covered in the previous chapters, such as Knative Eventing, Knative Functions, CloudEvents, and CDEvents. This section is divided into three subsections:

- Data Collection from Event Sources
- Data Transformation to CDEvents
- Metrics calculations

These sections map one to one with the proposed architecture that, from a high-level looks like the following diagram:

Figure 9.7 Collecting and transforming data to calculate DORA metrics



From a high-level perspective, we need to architect our data collection and transformation pipeline in a way that we can support any number of event sources as different companies and implementations will collect data from systems that we cannot anticipate. We are imposing the data to be in the form of CloudEvents before it enters our system. This means that if you have event sources that are not following the CloudEvents specification you will need to adapt their data to follow the specification. This can be easily achieved by using the CloudEvents SDK to wrap your existing events to follow the

specification.

Once the data enters our system, we will store it in persistent storage, in this case, we have chosen to use a PostgreSQL database to store all the incoming data and calculations. The next stage (Data Transformation) is not directly called by any component, instead, it fetches data from the database periodically and processes all the data that hasn't been processed yet. This stage (Data Transformation) is in charge to transform incoming CloudEvents that are already stored in the database into CDEvents structures that will be used to calculate the metrics. Once the transformation to CDEvents structure happens, the result is also stored in a separate table in our PostgreSQL database. Finally, the "Metrics Calculation" stage reads periodically from the database all new CDEvents that haven't been processed and calculates the metrics that we have defined.

This simple architecture allows us to plug in new data sources, new transformation logic depending on the data that we are receiving, and finally new metrics calculation logic for your domain-specific metrics (not only DORA metrics). It is also important to notice that as soon as we guarantee that the incoming data is correctly stored, all the transformations and calculations can be recalculated if the metrics data is lost.

Let's take a deeper look at the stages required to calculate the simplest of the DORA four key metrics, "**Deployment Frequency**".

9.3.1 Data collection from Event Sources

As shown in figure 9.x, we want to consume data from multiple sources, but we have set CloudEvents as the standard input format. While CloudEvents has been widely adopted, many systems still don't support the standard yet. In this section, we will look into Knative Sources as a mechanism that can declaratively define our event sources and transform non-CloudEvent data into CloudEvents.

The proposed solution then exposes a REST Endpoint to receive incoming CloudEvents. As we have seen in Chapter 6, we can use Knative Eventing to route events using HTTP to this new REST endpoint. Once we have

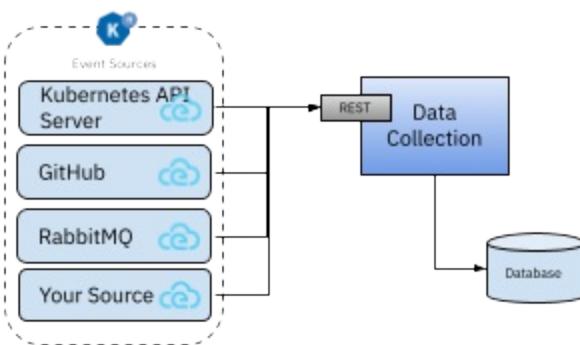
CloudEvents, we will validate the data and store it in a PostgreSQL table called `cloudevents_raw`.

Let's take a quick look at Knative Sources, as we must support different event sources.

Knative Sources

With Knative Sources, you can declaratively define which components are producing CloudEvents, you can use the components we discussed in Chapter 6, such as Brokers and Triggers, to route these events to different components.

Figure 9.8 Knative Sources and Data Collection



Several Knative Sources are provided out of the box by the Knative community and different software vendors. The following list is not exhaustive, but it covers some of the sources that you might want to use to calculate your metrics:

- APIServerSource
- PingSource
- GitHubSource
- GitLabSource
- RabbitMQSource
- KafkaSource

Check the complete list of third-party sources here:

<https://knative.dev/docs/eventing/sources/#third-party-sources>.

These sources transform events, for example, from the Kubernetes API Server, GitHub or RabbitMQ AMQP messages into CloudEvents.

If you want to use one of the available Knative Sources for example the `APIServerSource` you just need to make sure that the source is installed in your cluster and then configure the source according to your needs. For calculating the Deployment Frequency metric we will be tapping into Kubernetes Events related to deployments. You can declaratively configure the source and where the events will be sent by defining an `APIServerSource` resource:

Listing 9.1 Knative Source APIServerSource definition

```
apiVersion: sources.knative.dev/v1
kind: APIServerSource #A
metadata:
  name: main-api-server-source #B
spec:
  serviceAccountName: api-server-source-sa #C
  mode: Resource
  resources:
    - apiVersion: v1
      kind: Event #D
  sink: #E
  ref:
    apiVersion: v1
    kind: Service
    name: cloudevents-raw-service
    namespace: four-keys
```

As you can imagine, the `APIServerSource` will generate tons of events and all these events will be sent to the ``cloudevents-raw-service`` Service which will store them in the PostgreSQL database. More complex routing and filtering can be configured to only forward events that we are interested in, but we can also apply filtering in the next stages, allowing for an approach that can enable us to add more metrics as we evolve our data collection process.

With this source, every time a new Deployment resource is created, modified,

or deleted we will receive one or more CloudEvents and store them in the database.

If you have a system already producing events but not CloudEvents you can create your own Custom Knative Source. Take a look at the following tutorial for more information on how to do this:

<https://knative.dev/docs/eventing/custom-event-source/custom-event-source/>

. At the time of writing this book, the Knative Source for RabbitMQ (<https://knative.dev/blog/articles/eventing-rabbitmq-announcement/>) reached a stable maturity level. The RabbitMQ Source transforms RabbitMQ AMQP messages into CloudEvents allowing existing applications that are already using RabbitMQ to interact with components that are supporting CloudEvents already.

The big advantage of declaring and managing your event sources using Knative Sources is that you can query your sources as any other Kubernetes resource, monitor and manage their state, and troubleshoot when issues arise using all the tools available in the Kubernetes ecosystem.

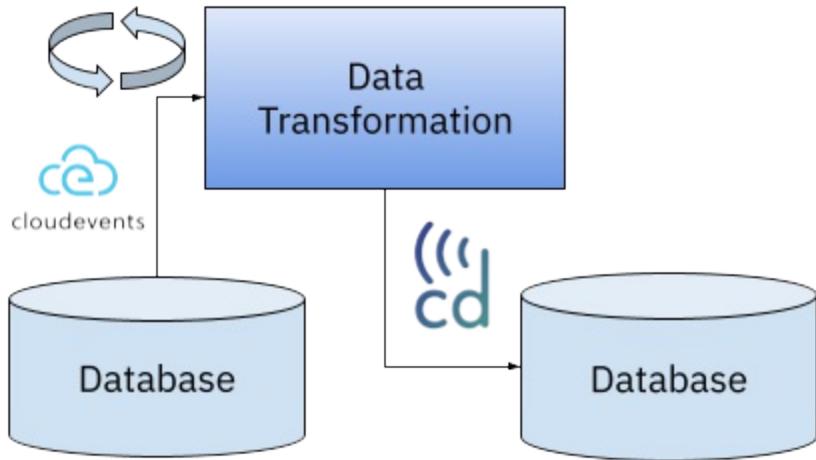
Once we have CloudEvents stored in our database, we can proceed to analyze them and map them into CDEvents for further calculations.

9.3.2 Data Transformation to CDevents

Now we have CloudEvents in our PostgreSQL database, we have validated that they are valid CloudEvents, and now we want to transform some of these very generic CloudEvents into CDEvents which we will use to calculate our metrics.

As explained in the introduction, these transformations will depend on what kind of metrics you are trying to calculate. For this example, we will look into internal Kubernetes events related to Deployment resources to calculate the Deployment Frequency metric, but completely different approaches can be used. For example you can, instead of looking into Kubernetes internal events, look into ArgoCD events or Tekton Pipeline events to monitor when deployments are triggered but from outside the cluster.

Figure 9.9 Mapping and transforming from CloudEvents to CDevents

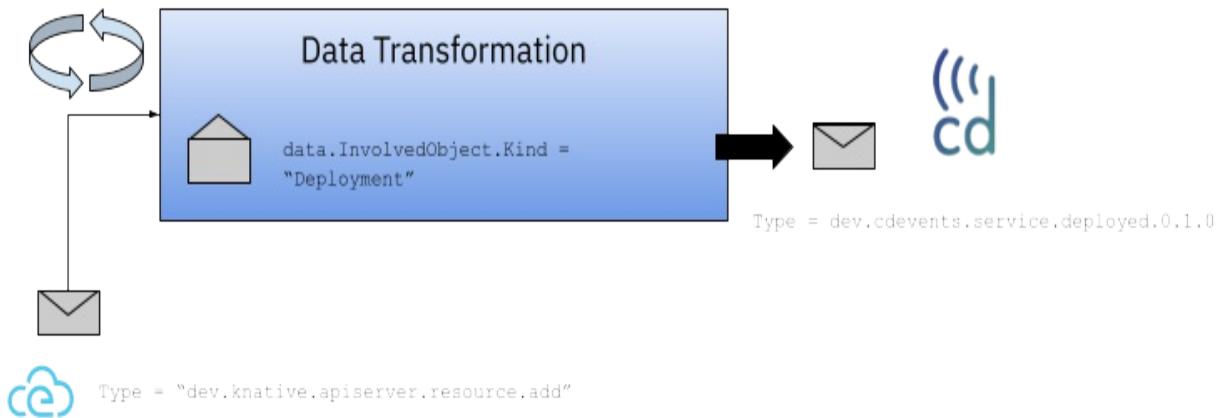


We need a way to map a very generic CloudEvent to a concrete CDEvent that indicates that a Service Deployment has happened or has been updated.

This mapping and transformation logic can be written in any programming language as we only deal with CloudEvents and CEvents. Because of the volume of events that we might be receiving, it is essential to block and process events as they arrive. For this reason, a more asynchronous approach has been chosen here. The data transformation logic is scheduled to happen at fixed periods, which can be configured depending on how often we want/can process the incoming events.

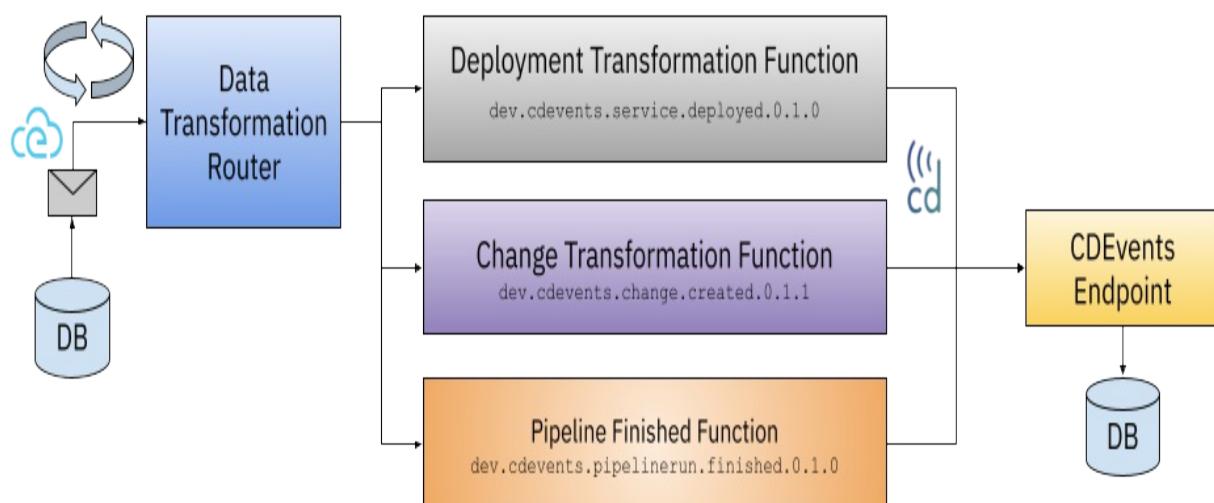
For this example, we will map and translate incoming events with `type` equal to `dev.knative.apiserver.resource.add` and `data.InvolvedObject.Kind` equal to "Deployment" to CEvents of the type `dev.cdevents.service.deployed.0.1.0`. This transformation is particular to our needs as it correlates events from the Knative API Server Source to the events defined in the CEvents specification.

Figure 9.10 Concrete mapping and CDEvent creation for Deployments



To calculate different metrics, we will need more of these transformations. One option would be to add all the transformation logic into a single container. This approach would allow us to version all the transformations together as a single unit, but at the same time, it can complicate or limit teams writing new transformations, as they have a single place to change code. An alternative way that we can take is to use a function-based approach, we can promote the creation of single-purpose functions to do these transformations. By using functions, only functions that are currently transforming events are going to be running. All the ones that are not being used can be downscaled. If we have too many events to process, functions can be upscaled on demand based on traffic.

Figure 9.11 Using functions to map CloudEvents to CDEvents



As shown in figure 9.x a new component is needed to route the CloudEvents being read from the database to concrete functions. Each transformation function can transform the incoming CloudEvent by inspecting its payload, enriching the content with an external data source, or simply wrapping the entire CloudEvent into a CDevent.

The data transformation router component needs to be flexible enough to allow new transformation functions to be plugged into the system and multiple functions to process the same event (that is, the same CloudEvent being sent to one or more transformation functions).

Transformation and mapping functions don't need to care about how the CDevents will be persisted. This allows us to keep these functions simple and focused on transformations only. Once the transformation is done and a new CDEvent is produced, the function will send the event to the CDEvents endpoint component, which is in charge of storing the CDEvent in our database.

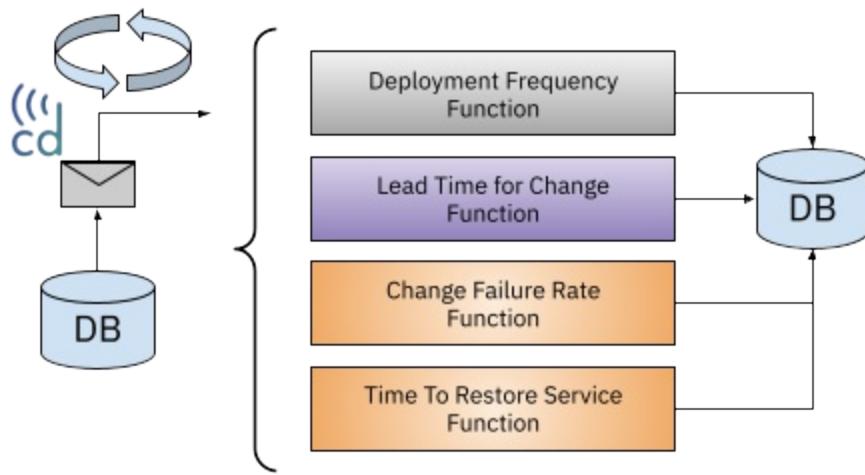
By the end of the transformations, we will have zero or more CDevents stored in our database. These CDevents can be used by the metric calculation functions that we will look at in the following section.

9.3.3 Metrics Calculation

To calculate our metrics (DORA or custom metrics), we will use the same function-based approach we used for the CDevents transformation and mapping.

In this case, we will write functions to calculate different metrics. Because each metric requires aggregating data from different events and maybe systems, so each metric calculation function can implement a different logic. The mechanisms used to calculate a metric are up to the developers who write the code to perform the calculation.

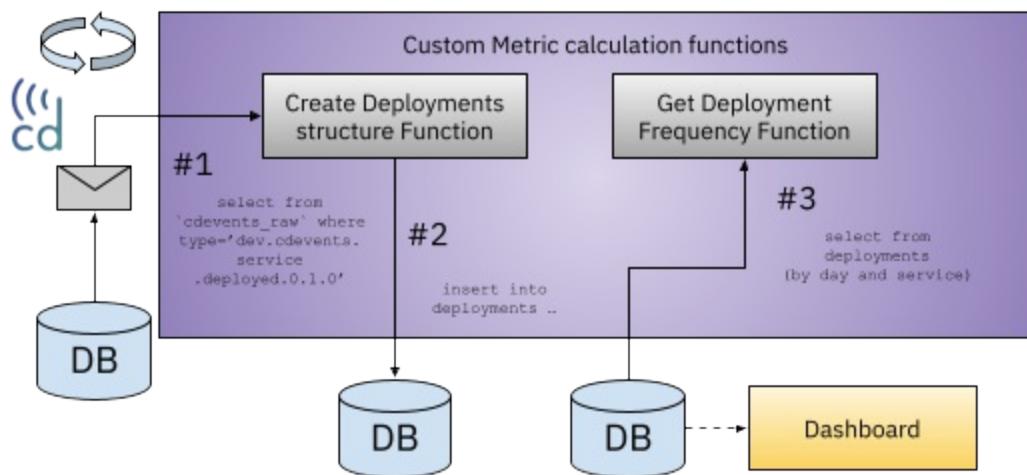
Figure 9.12 Using functions to calculate DORA metrics



To calculate metrics, each function can be configured to fetch very specific CDEvents from the database and with different periods depending on how often we need to get updates for a particular metric. The metric result can be stored in the database or sent to an external system, depending on what you want to do with the calculated data.

If we look at calculating the Deployment Frequency metric to have a more concrete example, we need to implement a couple of custom mechanisms and data structures to keep track of the metric.

Figure 9.13 Deployment Frequency calculation flow



A simplified flow for calculating the Deployment Frequency metric is shown in figure 9.x where the first step (#1) is to get CDEvents related to

deployments from the `cdevents_raw` table. The “Create Deployments structure function” is in charge of reading CDEvents with type `dev.cdevents.service.deployed.0.1.0`, inspecting the payload and metadata, and creating a new structure that can be later queried. Step #2 is in charge of persisting this new structure in our database. The main reason for this structure to exist is to make our data easier and more performant to query for the metric that we are implementing. In this case, a new `deployment` structure (and table) are created to record data we want to use to calculate our Deployment Frequency metric. For this simple example, the deployment structure contains the service's name, the timestamp, and the deployment's name. In step #3 we can use this data to get our deployment frequency by service and divide it into buckets per day, week, or month.

These functions need to be idempotent, meaning that we should be able to retrigger the calculation of the metrics again using the same CDEvents as input, and we should obtain the same results.

Optimizations can be added to this flow; for example, a custom mechanism can be created to avoid reprocessing CDEvents that have already been processed. These customizations can be treated as internal mechanisms for each metric, and developers should be able to add integration with other systems and tools as needed. For the sake of the example, the “Get Deployment Frequency Function” can fetch the metrics from the database. Still, in a more realistic scenario, you can have a Dashboard directly querying the database where the simplified structures are stored, as many dashboard solutions provide an SQL connector out of the box.

Now that we have covered the flow to calculate the Deployment Frequency metric, let's take a look at a working example, where we will install all the components required for Data Collection, Data transformation, and Metrics calculation.

9.3.4 Working example

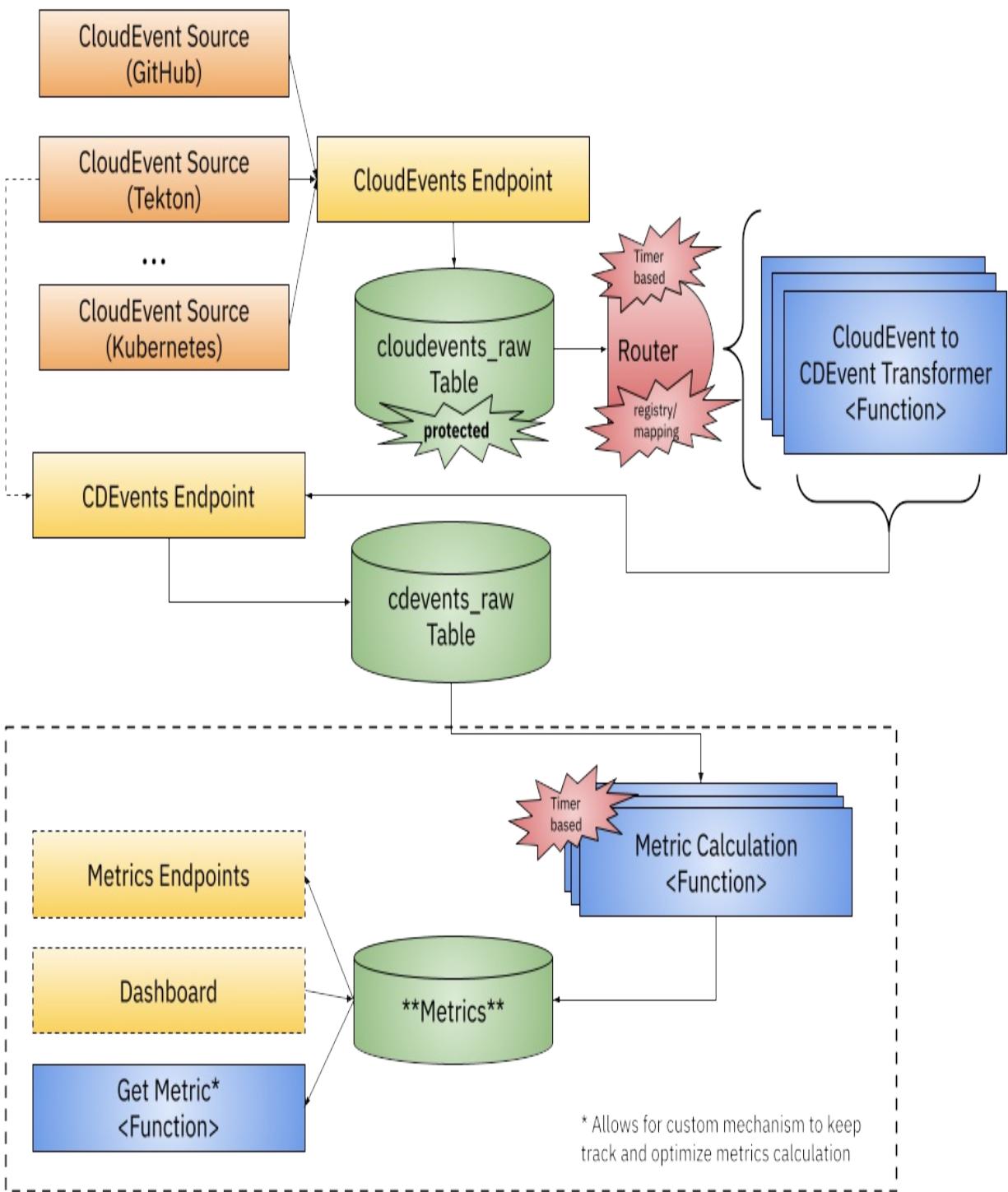
In this section we will be looking at a working example, showing how we can put Data Collection, Data Transformation to CDEvents, and Metrics calculation together for our Kubernetes-based platforms.

This section covers a very implementation that can be found in the following repository, alongside a step-by-step tutorial on how to install and how run the components needed as well as how to produce information and extract the metrics (<https://github.com/salaboy/from-monolith-to-k8s/blob/main/four-keys/README.md>).

The architecture implemented in this example puts together the stages defined in the previous sections such as Data Collection, Data Transformation, and Metrics calculation. One of the main aspects covered by this architecture is the extensibility and pluggability of components for Data Transformation and Metrics Calculation. This architecture assumes that we will be collecting data as CloudEvents, so it is the user's responsibility to transform their event sources to CloudEvents to be able to leverage this architecture.

The following diagram shows how all the components are tied together to provide the functionality of deciding which events we want to collect, and how to transform these events into CDevents so then we can calculate DORA metrics with them.

Figure 9.14 Example architecture for capturing and calculating DORA metrics



While the architecture might look complicated at first, it was designed to allow custom extensions and mappings necessary to support collecting and processing events from various sources.

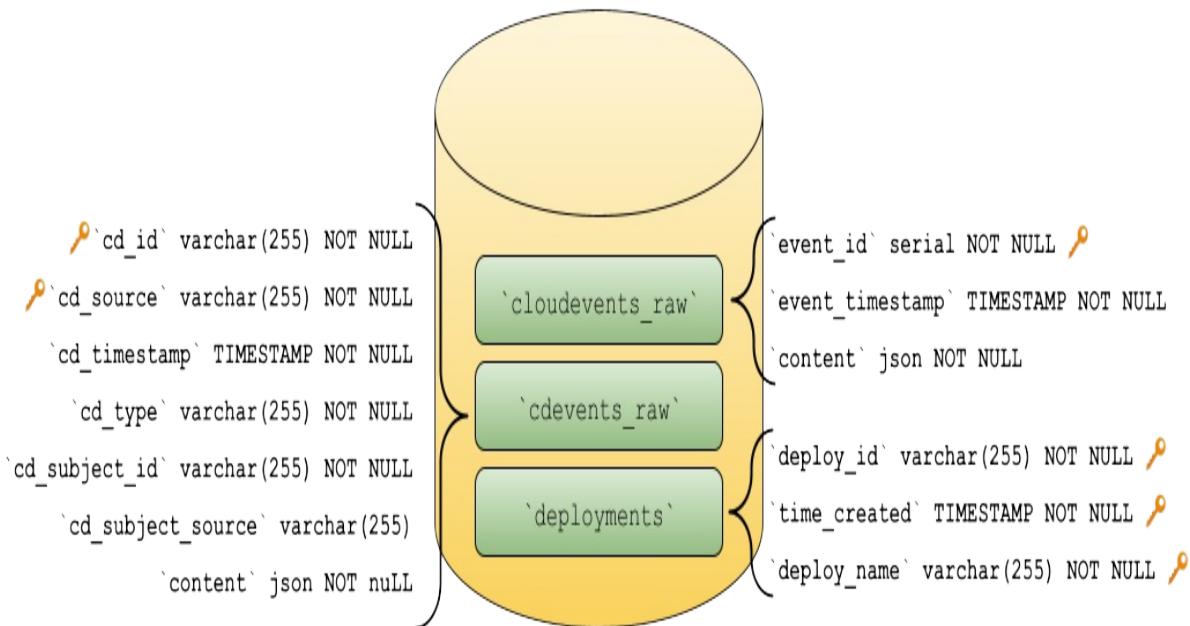
Following the step-by-step tutorial, you will create a new Kubernetes Cluster to install all the components needed to collect CloudEvents and calculate the metrics. Still, the architecture is in no way limited by a single cluster. After you create and connect to a cluster, you will install a set of tools such as Knative Serving for our functions runtime and Knative Eventing only for our Event Sources.

Once we have the cluster ready, you will create a new `namespace` to host all the components actively processing the data collected. But first, we will install a PostgreSQL database that we will use as our Event store.

Storing Events and Metrics

Once we have our database to store events and metrics information, we need to create the tables for our components to store and read events, for this example, we will create the following three tables: `cloudevents_raw`, `cdevents_raw` and `deployments`.

Figure 9.15 Tables CloudEvents, CDEvents and metrics calculations



Let's take a look at what information we are going to be storing in these three tables:

- `cloudevents_raw` Table: This table stores all the incoming CloudEvents from different sources. The main purpose of this table is data collection.
 - `event_id`: this value is generated by the database,
 - `event_timestamp`: stores the timestamp of when the event is received. This can be later used to order events for reprocessing.
 - `content`: stores the serialized JSON version of the CloudEvent in a JSON column.

This table is kept as simple as possible because we don't know what kind of cloud events we are getting, and at this point, we don't want to unmarshal and read the payload, as this can be done in the data transformation stage.
- `cdevents_raw` Table: This table stores all the CDEvents we are interested in storing after filtering and transforming all the incoming CloudEvents. Because CDEvents are more specific, and we have more metadata about these events, this table has more columns:
 - `cd_id`: stores the CloudEvent Id from the original CloudEvent.
 - `cd_timestamp`: stores the timestamp of when the original CloudEvent was received.
 - `cd_source`: stores the source where the original CloudEvent was generated.
 - `cd_type`: stores and allows us to filter by different CDEvents types. The types of CDEvents that will be stored in this table are defined by the transformation functions that we have running in our setup.
 - `cd_subject_id`: stores the id of the entity associated with this CDEvent. This information is obtained when our transformation functions analyze the content of the original CloudEvent.
 - `cd_subject_source`: stores the source of the entity associated with this CDEvent.
 - `content`: the JSON serialized version of our CDEvent, which includes the original CloudEvent as a payload.
- `deployments` table: This table is custom to calculate the Deployment

Frequency metric, there are no rules to what you store in these custom tables that are used to calculate different metrics. For the sake of simplicity, this table only has three columns:

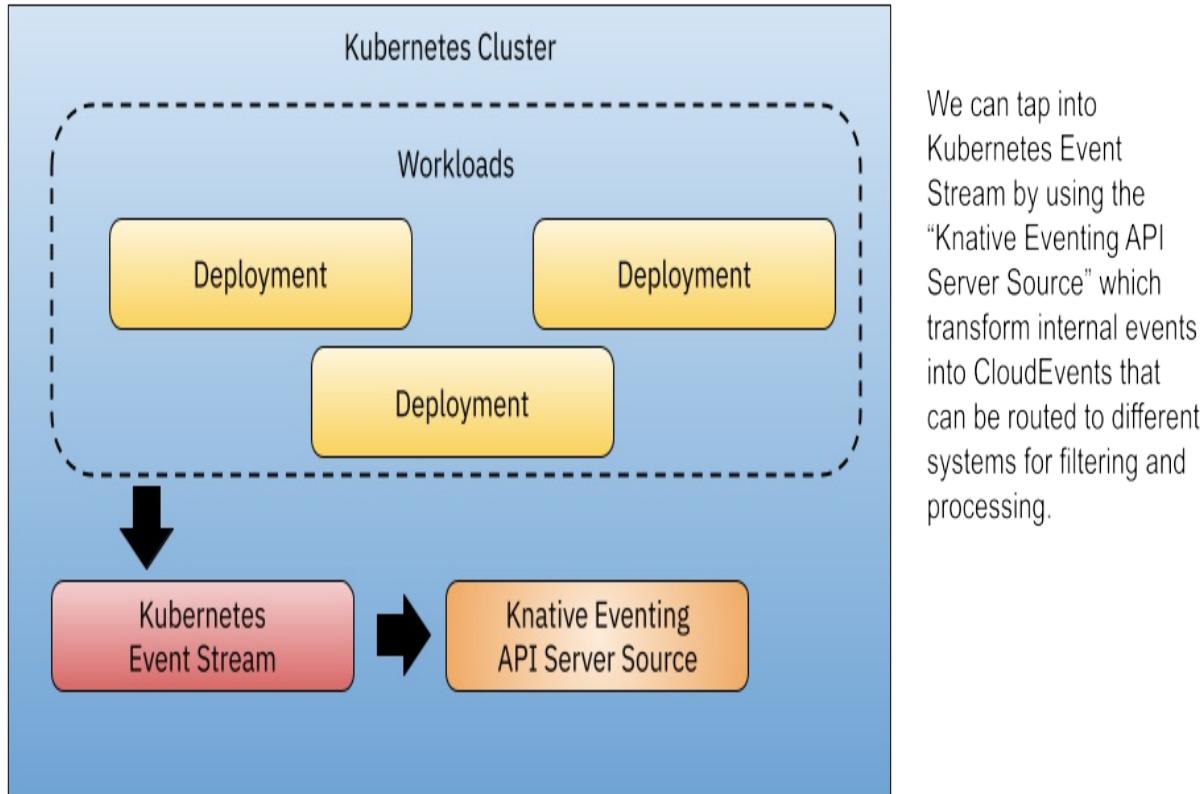
- `deploy_id`: the id used to identify a service deployment
- `time_created`: when the deployment was created or updated
- `deploy_name`: the deployment name used to calculate the metrics.

Once we have the tables ready to store our events and metrics data, we need to have events flowing into our components, and for that, we will need to configure Event Sources.

Configuring Event Sources

Finally, and before installing the data transformation functions or the metrics calculation functions, we will configure the Kubernetes API Server Event Source from Knative Eventing to be able to detect when new deployments are being created.

Figure 9.16 Example using the Knative Eventing API Server Source



We can tap into Kubernetes Event Stream by using the "Knative Eventing API Server Source" which transform internal events into CloudEvents that can be routed to different systems for filtering and processing.

Here, you can use any CloudEvent-enabled data source, the Knative API Server Source is used as an example to demonstrate how easy it is to consume and route events for further processing.

Check projects like ArgoEvents (<https://argoproj.github.io/argo-events/>) and other Knative Eventing sources (<https://knative.dev/docs/eventing/sources/>) to familiarize yourself with what is available out of the box. Also, check the CloudEvents specification adopters list (<https://cloudevents.io/>), as all these tools are already generating CloudEvents that you can consume and map to calculate metrics.

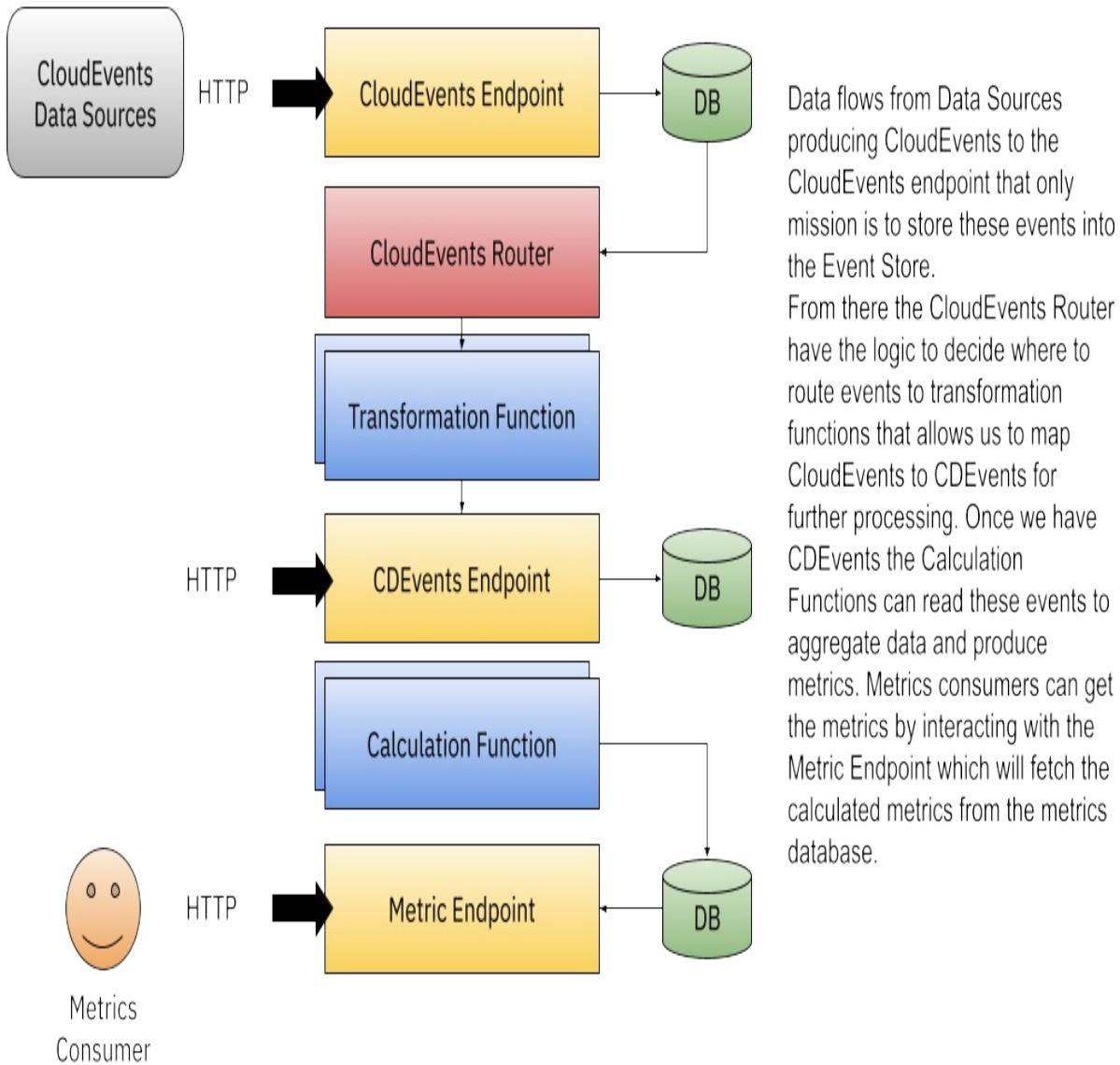
Deploying Data Transformation & Metrics Calculation Components

Now that we have a place to store our events and metrics data, event sources configured and ready to emit events when users interact with our cluster, we can deploy the components that will take these events, filter them and transform them to finally calculate our metrics.

The tutorial deploys the following components:

- **CloudEvents Endpoint:** expose an HTTP endpoint to receive CloudEvents and connects to the database to store them.
- **CDEvents Endpoint:** expose an HTTP endpoint to receive CEvents and connects to the database to store them.
- **CloudEvents Router:** reads CloudEvents from the database and routes them to the configured Transformation Functions. This component allows users to plug their own transformation functions that can transform a CloudEvent into a CEvent for further processing. The CloudEvents router runs periodically, by fetching unprocessed events from the database.
- **(CEvents) Transformation Function:** transformations functions can be defined by users and map CloudEvents to CEvents. The idea here is to enable users to add as many functions as needed to calculate DORA and other metrics.
- **(Deployment Frequency) Calculation Function:** metrics calculation functions provide a way to calculate different metrics by reading CEvents from the database. These functions can store the calculated metrics into custom database tables if needed.
- **(Deployment Frequency) Metric Endpoint:** these metric endpoints can be optionally exposed for applications to consume the calculated metrics. Alternatively, dashboards can query the data directly from the database.

Figure 9.17 Components and data flow



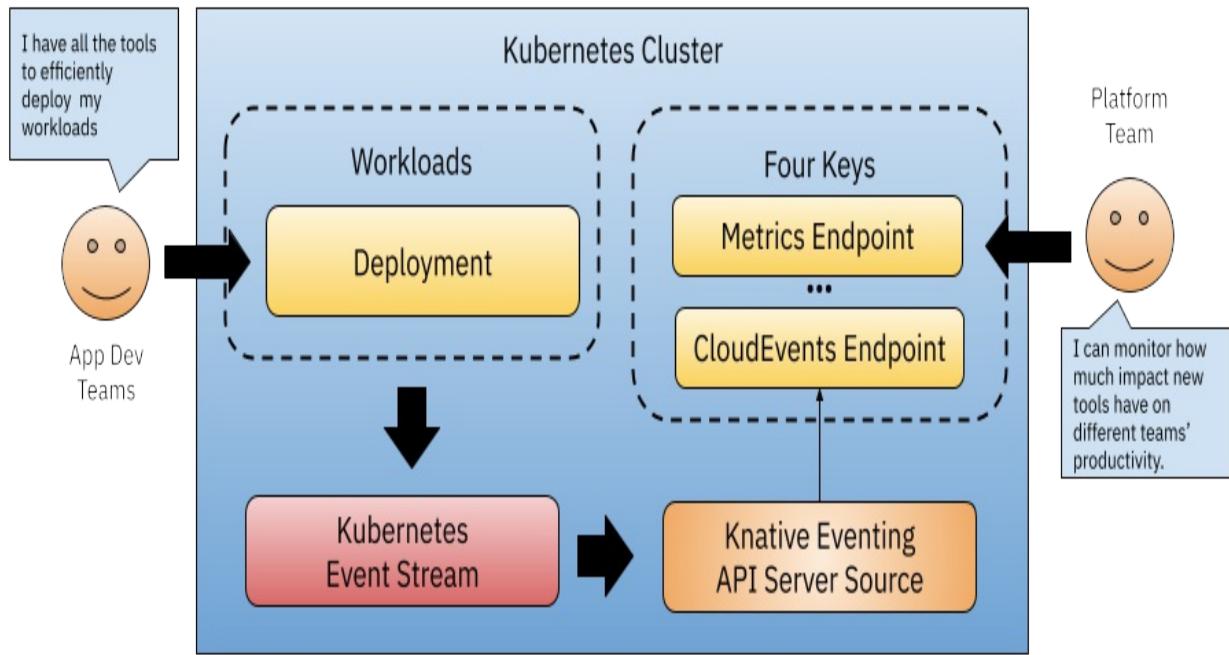
As soon as we have our components up and running we can start using our cluster to generate events that will be filtered and processed by these components to produce the Deployment Frequency Metric.

Deployment Frequency Metric for your deployments

To calculate the Deployment Frequency metric we need to deploy new workloads to our cluster. The tutorial includes all the transformation functions and metric calculation functions to monitor events coming from Deployment Resources.

While development teams can create and update their existing deployments the platform team can transparently monitor how efficient the platform is to enable teams to perform their work.

Figure 9.18 Components and data flow



Finally, you can `curl` the following endpoint if you are running the example on KinD:

```
> curl http://fourkeys-frequency-endpoint.four-keys.127.0.0.1.ssl  
You should see something like this:  
[  
  {  
    "DeployName": "nginx-deployment-1",  
    "Deployments": 3,  
    "Time": "2022-11-19T00:00:00Z"  
  },  
  {  
    "DeployName": "nginx-deployment-3",  
    "Deployments": 1,  
    "Time": "2022-11-19T00:00:00Z"  
  }  
]
```

Transformation and metrics calculation functions are scheduled to run every

minute. Hence these metrics will be only returned after the functions have been executed.

Alternatively, you can use a Dashboard solution like Grafana to connect to our PostgreSQL database and configure the metrics. Dashboard tools can be focused on the tables that store data about particular metrics. For our deployment frequency example, the table called `deployments` is the only one relevant for displaying the metrics.

I strongly recommend you check the example and try to run it locally, follow the step-by-step tutorial and get in touch if you have questions or if you want to help improve it.

Modifying the example to calculate the metrics differently or to add your custom metrics will give you a good overview of how complex these metrics calculations are but, at the same time, how important it is to have this information available to our application development and operations teams so they can understand how things are going almost in real-time.

These metrics built into our platforms can help us measure improvement and justify investing in tools that facilitate our software delivery practices. If we want to include a new tool in our platform, you can test your assumptions and measure the impact of each tool or adopted methodology. It is quite a common practice to have these metrics accessible and visible for all your teams so when things go wrong or when a tool is not working out as expected you will have hard evidence to back up your claims.

I hope that by showing this example, I managed to highlight the importance of measuring how good or bad our technical decisions are and the impact that these decisions have on all the teams involved in delivering software in front of your customers.

9.4 Closing words and next steps

Unfortunately I don't have an unlimited number of pages or unlimited time to keep adding content to this book, but I did my best to include the topics and challenges that I've seen organizations and communities trying to solve in the

Cloud Native space. We have reached a point in the Kubernetes ecosystem where tools are maturing and more projects are graduating which indicates that more and more companies are reusing tools instead of building their own.

I've intentionally left out an appendix on the topic of extending Kubernetes with your own custom controllers, because that should be left to very special cases where no tools exist to solve a problem that you are trying to solve. For the most common cases, as we have seen in this book, CI/CD, GitOps, Event Driven Architectures, Functions-as-a-Service platforms, provisioning infrastructure in the cloud, developer tools and platform building there are tools that are mature enough for you to use and extend if necessary.

I strongly recommend you to engage with your local Kubernetes communities and be active in the Open Source ecosystem, this not only gives you a great playground to learn, but it helps you to make the right informed decisions about which technologies to adopt. Understanding how strong the communities behind these projects are is key to validating that they are solving a problem that first needs a solution and that is common enough to be solved in a generic (and non company) specific way.

Tools like OSS Insight (<https://ossinsight.io/>) provide an enormous value for decision making and for making sure that if you invest time and resources on an Open Source project your changes and improvements will be maintained by an active community.

9.5 Summary

- By using DORA metrics, you can have a clear picture of how the organization is performing in delivering software in front of your customers. This can be used to understand bottlenecks resulting in improvements on the platforms we are building.
- CloudEvents brings standardization about how we consume and emit events. Over the last couple of years, we have seen a rise in the adoption of CloudEvents by different projects in the CNCF landscape. This adoption allows us to rely on CloudEvents to get information about components and other systems that we can aggregate and collect helpful

information that can be used for decision-making.

- CDEvents provides a CloudEvents extension, a set of more specific CloudEvents related to Continuous Delivery. While I expect the adoption of CDEvents to grow over time, we have seen how we can map CloudEvents to CDEvents to calculate the “four keys” metrics proposed by the DORA project. By using CDEvents as the base types to calculate these metrics, we can quickly adapt other event sources to leverage all the tooling and, in this case, all the metrics that are calculated.
- If we can measure our platform, we will know what needs improvement and where the organization is struggling with its delivery practices. This feedback loop provided by the metrics gives valuable information to platform teams in charge of continuously improving the tools and processes our teams use daily.