Compliments of codefresh

Continuous Delivery for Kubernetes

Mauricio Salatino

MANNING

codefresh

Conquer DevOps

From Zero to GitOps in a Few Clicks

Thousands of DevOps teams depend on Codefresh to deploy their software in a safe and scalable manner. You can easily automate your deployments in minutes using our managed enterprise platform powered by argo. Plus, Codefresh can integrate with best-of-breed tools to support your software delivery end to end for even the most complex scenarios.

Start your **free hosted Argo CD now** and learn what you have been missing!

Visit codefresh.io/free-trial

Continuous Delivery for Kubernetes

Continuous Delivery for Kubernetes

Mauricio Salatino

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Marija Tudor

ISBN: 9781633438590

contents

foreword

Going "cloud-native" is about a lot more than just using Kubernetes. A monolithic service shoved into a container may function but will leave most of the advantages of cloud-native architecture on the cutting room floor. The goal is not just to be "cloud-native" for its own sake. The goal is to realize the benefits! More frequent deployments, fewer regressions, more reliability, and even a more efficient developer experience are all possible when teams embrace a cloud-native approach.

Author Mauricio Salatino goes beyond simple tips to explore application architecture and approaches. More often than not, when teams find their Kubernetes applications misbehaving, they miss one of these critical architectural points. So if you're on your way to a GitOps nirvana with Argo CD and Codefresh, this book will help you bring your applications with you!

—Dan Garfield
Co-Founder and Chief Open Source Officer, Codefresh
Argo Project and Open GitOps Maintainer

1 Cloud-Native continuous delivery

Building and delivering modern Cloud-Native applications is hard. You end up building highly complex distributed applications that are continuously evolving on top of a tech stack that is continuously changing. Delivering software has always been challenging; delivering software efficiently and reliably is still considered a holy grail by many. In today's world, how fast you deliver new features to your users/customers can become a real differentiator from your competition; hence, it is becoming a priority for companies from all industries to change the way they work, how they organize teams, and how they architect and deliver software.

This report focuses on applying Continuous Delivery practices to modern Cloud-Native environments using Kubernetes as the target platform. This report will also provide links to step-by-step tutorials and aims to be very practical in showing how you can use different tools with the primary goal of delivering software reliably and efficiently.

Each of the tools introduced in this report has been chosen to solve specific challenges that you will face when building Cloud-Native applications. Some of these tools solve very technical and architectural challenges. Some cover how teams will improve collaboration. But all the tools presented have a shared goal: to help you to deliver robust and reliable software to your customers.

This report is divided into two parts, the first part covers the basics around Cloud-Native applications and their challenges, Continuous Delivery practices, and Kubernetes as the platform of choice to run our applications. In this first part, we will be looking at a Walking Skeleton, a demo application that will help us try different tools and assumptions about how our distributed applications will work and what is expected of them.

The second part goes over the challenges that we will face when trying to deliver these applications to different environments going from source to our applications being deployed for use by our customers/users. In this second part, the focus will be on figuring out the tools to transform the source code that our developers are creating into running versions of our applications that can be installed in different environments.

Before we begin, let's break down the main things we will be looking at in this first part. We will break down the second part once we come to it.

This first part is divided into the following sections:

1. Are you Cloud-Native?
   - What does this mean for Kubernetes?
   - Kubernetes? Where? How do we choose one?
2. Continuous Delivery Goals
   - Are you doing Continuous Delivery?
3. The need for a Walking Skeleton
   - Use case
   - Installing the application into a Kubernetes cluster
   - Interacting with the application and Kubernetes basics
4. Cloud-Native application challenges
   - Downtime is not allowed
   - Service's built-in resiliency
   - Dealing with the application state is not trivial
   - Data inconsistent data
   - Understanding how the application is working
   - Application security and identity management

Now, let's begin with the fundamentals and answer the question: are you Cloud-Native?

1.1 Are you Cloud-Native?

Let's get straight to the point. You will see a lot about Kubernetes in this report, but you need to understand that you can implement Cloud-Native applications without using Kubernetes. Similarly, you can apply Continuous Delivery practices without Kubernetes. Still, this report aims to deliver a practical experience on a real technology stack that is widely available today, so the reader can experience the advantages of Continuous Delivery first-hand.

Let's get the definitions out of the way, Cloud-Native is a very overloaded term, and while you shouldn't worry too much about it, it is essential to understand why this report makes use of concrete tools that run on top of Kubernetes using containers.

A good definition of the term can be found on the VMWare site by Joe Beda (Co-Founder, Kubernetes and Principal Engineer, VMware) at https://tanzu.vmware.com/cloud-native:

"Cloud-Native is structuring teams, culture, and technology to utilize automation and architectures to manage complexity and unlock velocity."

As you can see, there is much more than the technology associated with the term *Cloud-Native*. There is a people and culture angle to it, that pushes us to reevaluate how we are building software. This report, while covering technology, will make a lot of references to practices that can speed up the process of creating and delivering Cloud-Native applications.

On the technical side, Cloud-Native applications are heavily influenced by the "12-factor apps" principles (https://12factor.net) which were defined to leverage cloud-computing infrastructure. These principles were created way before Kubernetes existed and served to establish recommended practices for building distributed applications. With these principles, you can separate services to be worked by different teams using the same assumptions on how these services will work and interact with each other. These 12 factors are:

- I. Codebase
  One codebase tracked in revision control, many deploys
- II. Dependencies
  Explicitly declare and isolate dependencies
- III. Config
  Store config in the environment
- IV. Backing services
  Treat backing services as attached resources
- V. Build, release, run
  Strictly separate build and run stages
- VI. Processes
  Execute the app as one or more stateless processes
- VII. Port binding
  Export services via port binding
- VIII. Concurrency
  Scale out via the process model
- IX. Disposability
  Maximize robustness with fast startup and graceful shutdown
- X. Dev/prod parity
  Keep development, staging, and production as similar as possible
- XI. Logs
  Treat logs as event streams
- XII. Admin processes
  Run admin/management tasks as one-off processes

By following these principles, you are aiming to manage and reduce the complexity of building distributed applications, for example, by scoping a smaller and more focused

set of functionalities into what is known as a microservice. These principles guide you to build stateless microservices (VI and VIII) that can be scaled by creating new instances (replicas) of the service to handle more load.

By having smaller microservices, you end up having more services for your applications. This forces you to have a clear scope for each microservice, where the source code is going to be stored and versioned (I), and its dependencies (II and IV). Having more moving pieces (microservices), you will need to rely on automation to build, test, and deploy (V) each service, so having a clear strategy becomes a must from day one. Now you need to manage an entire fleet of running services, instead of just one big ship (monolith), which requires you to have visibility on what is going on (XI) and plan accordingly for cases when things go wrong (XII). Finally, to catch production issues early (X), it is highly recommended you work and regularly perform testing on environments that are as close as possible to your production environment.

The term Cloud-Native is also strongly related to container technologies (such as Docker) because containers by design follows best practices from the "12-factor apps" principles as they were designed with Cloud-Native applications and cloud infrastructures in mind. Once again, you can implement Cloud-Native patterns without using containers. Still, for the sake of simplicity, in this report, Cloud-Native services, "12-factor apps", and microservices are all going to be packaged as containers, and these terms will be used as synonyms.

If you are following the "12-factor apps" principles, you and your teams are going to be building a set of services that have a different lifecycle and can evolve independently. No matter the size of your team, you will need to organize people and tools around these services (figure 1.1).

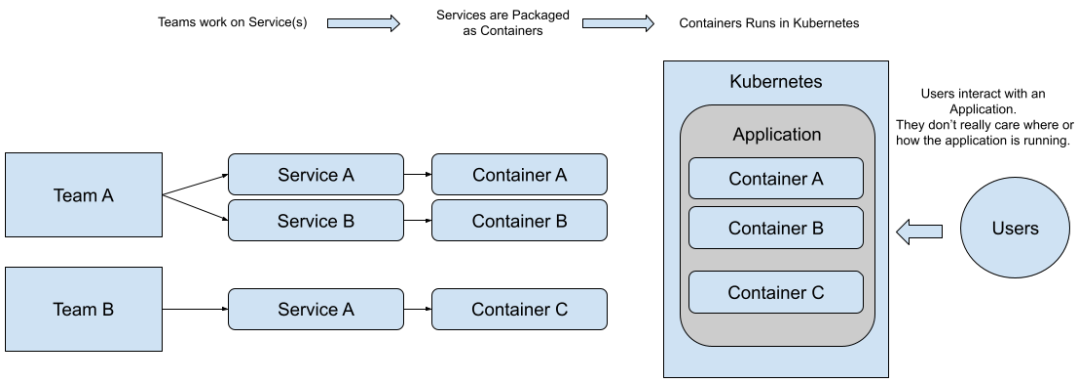

Figure 1.1 Teams, services, and containers composing applications for end-users.

When you have multiple teams working on different services, you will end up with tens or even hundreds of services. When you have three or four containers and a set of computers to run them, it is possible to decide where these services will run manually, but

when the amount of services grows, and your data center needs to scale, you will need to automate this job. That is precisely the job of a container orchestrator, which will decide for you based on the size and utilization of your cluster (machines in your data center) where your services will run.

The industry already chose Kubernetes to become the de facto standard for container orchestration. You will find a Kubernetes managed service in every major cloud provider and on-prem service offered by companies such as Red Hat, VMWare, and others.

Kubernetes provides a set of abstractions to deal with a group of computing resources (usually referred to as a cluster; imagine a data center) as a single computer. Dealing with a single computer simplifies the operations because you can rely on Kubernetes to make the right placements of your workloads based on the state of the cluster. Developers can focus on deploying applications, and Kubernetes will take the burden of placing them where it is more appropriate.

Kubernetes provides a developer and operations-friendly declarative REST API to interact with these abstractions (figure 1.2). Developers and Operations can interact with different Kubernetes Clusters by using a CLI (command-line interface) called `kubectl` or directly calling the REST APIs exposed by each cluster.

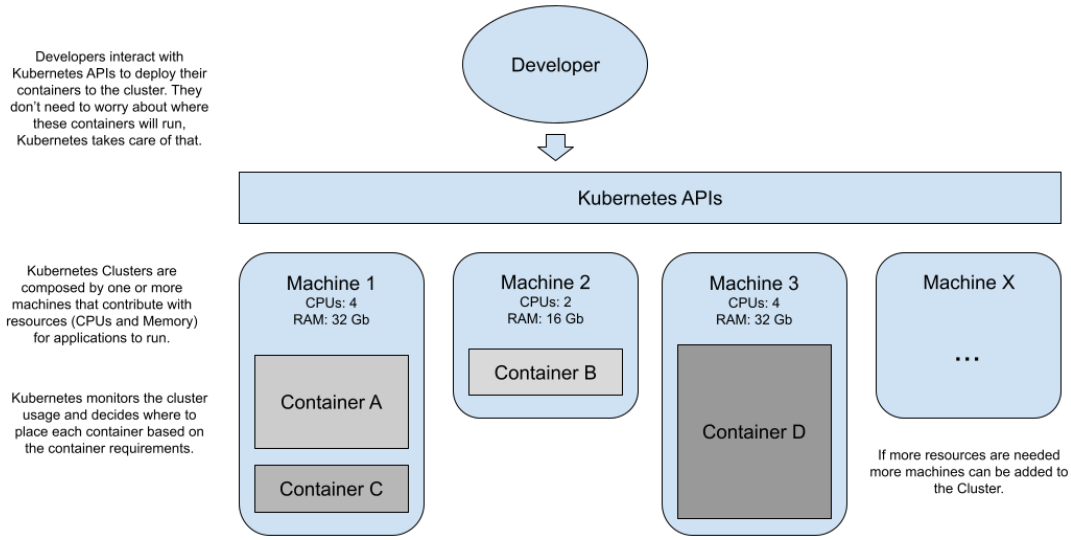

Figure 1.2 **Kubernetes lets you focus on running your services.**

Kubernetes is in charge of deciding where your containers will run (in which specific machine), based on the cluster utilization and other characteristics that you can tune. Most of the time, as a developer, you are only interested in having your applications running, not where they run.

Both *Kubernetes in Action, Second Edition* and *Core Kubernetes* are highly recommended books if you are interested in learning how to work with Kubernetes and how Kubernetes works internally.

For the sake of space, and in the name of focus, this report is fully centered around the idea of using Kubernetes as our target platform where our application will be deployed. Before moving forward, I wanted to make sure that it is clear that someone will need to run and manage Kubernetes Clusters. Normally, this is done by an operations team which is in charge of dealing with the hardware or virtual machines that will compose a cluster. This operations team can be a team inside your company, or you can use a managed service, which basically means that another company will charge you for running and maintaining those clusters.

As a developer you have several options:

- *Run a Kubernetes Cluster locally in your environment*: This is a great option to get started if you want to start learning and gaining experience with the Kubernetes APIs and the basic concepts. But it has several limitations, such as your local environment CPUs and memory, that you can allocate for the cluster to run. If you aim to run large applications, this is probably not going to be enough.
- *Use a cloud provider-managed Kubernetes service*: The easiest and quickest way to access a real Kubernetes Cluster is by using a cloud provider such as Google, Azure, AWS, Linode, Digital Ocean, etc. All these providers will give you a free trial to get started, but most of them require you to enter a credit card. You can find a community-maintained list of free Kubernetes Trials at https://github.com/learnk8s/free-kubernetes. If your company is already using any of these cloud providers, requesting access to them might be easier. The big downside of this approach is that someone will need to pay for those services.
- *Request a Kubernetes cluster to your company operation team*: If your company is already working with Kubernetes, they might have already an operations team that can provision new clusters on-demand. In most of the cases, you will need to justify why your team needs a Kubernetes cluster because the operations team will need to allocate hardware for that cluster to run.

No matter which option you choose, all the examples covered in this report should work for all the options listed.

Be aware that no matter if you choose Kubernetes, 12-factor principles, and containers you are not guaranteed to succeed, you need to use them wisely to rip the benefits that these tools were designed to provide. Every decision along the way introduces its own opinions, challenges, and restrictions that you will need to follow to make the best out of your stack of choice. By choosing Kubernetes you are buying into a set of best practices, de-facto standards, and very non-opinionated solutions that enable you with very flexible tools to implement a wide range of scenarios. For these reasons in the following sections we will evaluate the (Continuous Delivery) practices that we need to rely on to navigate the technical challenges that we will face in our Cloud-Native journey.

Let's start by looking at the Continuous Delivery Goals to then go and get hands on by installing and interacting with a demo application that we can use to test our assumptions and tools.

1.2 Continuous Delivery goals

Delivering valuable software to your customers/users in an efficient way should be your main goal. While building Cloud-Native applications, these become challenging as you are not dealing with a single application, you are now dealing with complex distributed applications and multiple teams delivering features at different paces.

For the remainder of the report, the following goal of Continuous Delivery is going to be used, to guide the selection of different projects and tools for your teams to use:

> *"Goal: Deliver useful, working software to users as quickly as possible.*
>
> *Focus on Reducing cycle time (The time from deciding to make a change to having it available to users.)"*

This goal definition comes from the Continuous Delivery book[1] written by Jez Humble and David Farley. The book, on purpose, doesn't go deep into any technologies besides naming them, and because it was written more than 10 years ago, the cloud and Kubernetes didn't exist in the way that exists today.

There are some significant areas covered by Humble and Farley's book that you will read about here, such as:

- *Deployment pipeline*: All the steps needed to create and publish the software artifacts for our application's services.
- *Environment Management*: How to create and manage different environments to develop, test, and host the application for our customers/users.
- *Release Management*: The process to verify and validate new releases for your services.
- *Configuration Management*: How to manage configuration changes across environments in an efficient and secure way.

This report aims to be a practical guide where you can experience the concepts described by Humble and Farley in their book first-hand, with simple tools and a working example that you can modify to test different aspects of Continuous Delivery.

To benefit (and have some return on investments) from adopting Kubernetes, re-architecting your applications and running your workloads is not enough. You can only fully leverage Kubernetes design principles if your organization delivers more and better software to your users faster.

In such a way, the Cloud-Native Continuous Delivery goal can be stated as follows:

> *"Deliver useful, working software to users as quickly as possible by organizing teams to build and deploy in an automated way Cloud-Native applications that run in cloud-agnostic setup."*

This goal implies multiple teams working on different parts of these Cloud-Native applications that can be deployed to different cloud providers to avoid vendor lock-in. It also means the fact that Cloud-Native applications are more complex than old monoliths, but this inherent complexity also unlocks velocity, scalability, and resilience if managed correctly.

At this point, you might be wondering: Am I already doing Continuous Delivery?

1.2.1 Are you doing Continuous Delivery already?

Continuous Delivery is all about speeding up the feedback loop from the moment you release something to your users until the team can act on that feedback and implement the change or new feature requested. To produce efficient and reliable high-quality software, you need to automate a big part of this process.

I often hear people stating that they are already doing Continuous Delivery; hence, this section gives a quick overview of what the remainder of the report will be covering so you can map your current situation with some of these points:

- *Every change needs to trigger the feedback loop:* There are four main things that you need to monitor for changes and verify that these changes are not breaking the application:
  - *Code:* If you change the source code that will be built and run, you need to trigger the build, test, and release process for every change.
  - *Configuration:* If something in the configuration changes, you need to re-test and make sure that these changes broke nothing.
  - *Environment:* If the environment where you run the application changes, you need to re-test and verify that the application is still behaving as expected. Here is where you control and monitor which version of the operating system you are using, which version of Kubernetes is being used in every node of your cluster, etc.
  - *Data structures:* If a data structure changes in your application, you need to verify that the application keeps working as expected, because data represents a very valuable asset, every change needs to be correctly verified. This also involves a process for deciding how backward-compatible the change is and how the migration between the old and new data structure will work.
- *The feedback loop needs to be fast:* The faster the feedback loop is, the quicker you can act on it, and the smaller the changes are (figure 1.3). To make the feedback loop faster, most of the verifications need to be automated by applying a Continuous Integration approach. Usually, you will find the following kinds of tests required to verify these changes:

  - *Unit Tests:* At each project/service level, these tests can run fast (under 10 seconds) and verify that the internal logic of the services works. Usually, you avoid contacting databases or external services here, to prevent long-running tests. A developer should run these tests before pushing any changes.
  - *Integration/Component Tests:* These tests take longer because they interact with other components. But you need to verify that these interactions are still working. For these tests, components can be mocked, and this report covers "Consumer Contract Testing" to verify that new versions of the services are not breaking the application when their interfaces (contracts) change.
  - *Acceptance Tests:* Verify that the application is doing what it is supposed to do from a business perspective. Usually, this is verified at the service level, instead of at the user interface level, but there are different techniques to cover different angles. These tests are executed on top of the entire application. This requires a whole environment to be created and configured with the version of the service that includes the new change, and it can take more time to run.
  - *Manual Testing:* This is performed by a team that is going to test the application in an environment similar to production. Ideally, these testers should be testing what the users are going to get. These tests are prolonged, because they require people to go over the application.
- *Everything needs to be measured:* To make sure that we are going in the right direction, and you keep delivering high-quality software to your users, you need to track how much time and resources this feedback loop is taking you from start to finish. Here are some key measurements that will help you to understand how good you are. Based on the DORA report about the State of DevOps 2021 you can measure:
  - *How frequent your code deployments are:* How often are you deploying new versions to your production environment?
  - *Lead time from committing changes to deploying your service:* How much time does it take you from committing a change to version control to having those changes deployed in your production environment?
  - *Time to recover from incidents:* How much time does it take you to fix a service (or a set of services) that are misbehaving between when the issue is reported until the system is again in a stable state?
  - *Change failure rate:* How often do you deploy new versions that cause problems in the production environment?

A typical feedback loop looks like figure 1.3, where after a change in Configuration, Code, Data, or the Environment a build and test step is triggered followed by a new deployment including the new changes. Once we have the changes applied, we will take some measurements that will let us know if the new changes are performing as expected or if we need to roll back to the previous working version. If we are happy with the measurements, we can mark the release as done!

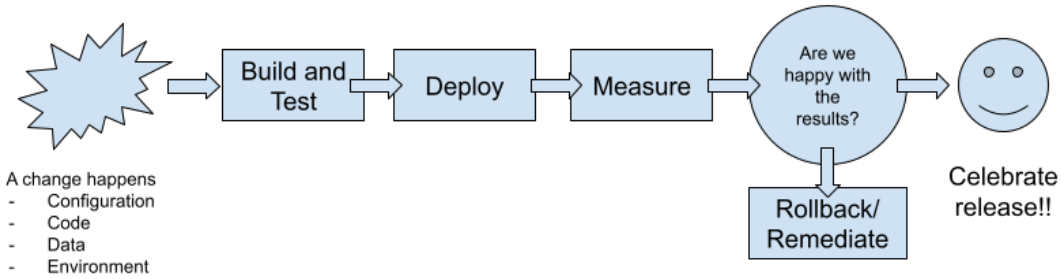

Figure 1.3 Fast feedback loops help you to accelerate your deliveries.

There is a high chance that you are doing some of these things already, but unless you are measuring, it becomes impossible to assert whether your changes are useful to your users or not.

The main objective of this report is to show how Continuous Delivery can be achieved in a Cloud-Native environment. This helps you to deal with the complex nature of distributed teams working on distributed applications. For this exact same reason, an example is needed, and by example, I don't mean a typical "hello world" for each technology that I will mention in the following sections.

In order to be convinced that continuous delivery can be achieved for your company or scenario, you need to see an example that you can map almost one-to-one with your daily challenges. The following section introduces an example that will be used throughout this report. To highlight some tools and frameworks, we need more than a simple example; hence, we use the term "walking skeleton" which represents a fully functional application that contains enough components and functionality to work end-to-end. A "walking skeleton" is supposed to highlight the defined architecture and how components interact with each other. This "walking skeleton" pushes you to define which frameworks, target platforms, and tools are you going to use to deliver your software. The following chapters will deep dive into different characteristics of the walking skeleton, which was created in an open-source way, for you to run in your own environment and use as a playground for testing new technologies before applying them to your own projects.

1.3 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 in order to deliver a simple PoC (proof of concept). A PoC explains how you build these projects into containers that can run inside Kubernetes to how to route traffic to the REST endpoints provided in each of these containers. If you want to experiment with new projects to see if they fit into your own ecosystem, you end up building a PoC to validate your understanding of how this shiny new project works and how it is going to save your and your teams' time.

For this report, 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 to change your own projects for the sake of experimentation.

The main purpose of this walking skeleton is to highlight how to solve very specific challenges from the architectural point of view and from 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 what is the minimum viable product that you need and deploy it quickly to a production environment where you can improve from there. By taking the walking skeleton all the way to a production environment, you can get 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 in my opinion. It is more important to understand how the pieces fit together and what tools and practices can be used to enable each team behind a service (or a set of services) to evolve in a safe and efficient way.

1.3.1 Building a conference platform

During this report, you will be working with a conference platform application. This conference platform can be deployed in a different environment to serve different conference events when needed. This platform relies on containers, Kubernetes, and tools that will work in any major Cloud-Providers as well as on-prem Kubernetes installations.

This is what the application's main page looks like (figure 1.4):

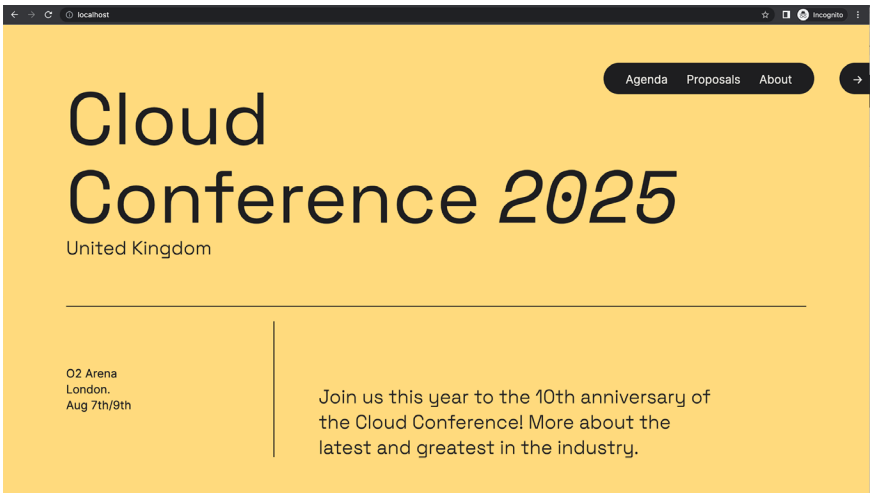

Figure 1.4 **Conference platform main site.**

The conference application lists all the approved submissions on the Agenda page. The main page will also allow potential speakers to submit proposals in the Proposal section while the "Call for Proposals" window is still open.

When we start the application for the first time, there are no confirmed talks in the agenda section (figure 1.5).

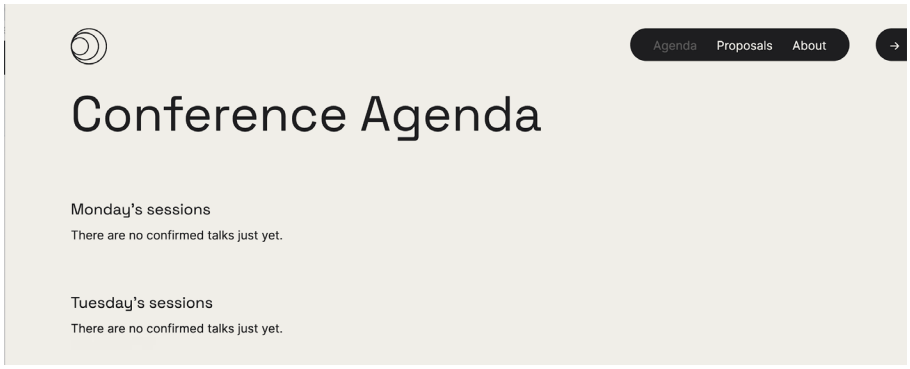

Figure 1.5 Conference Agenda with no confirmed sessions.

There is also a Back Office section for the organizers to review proposals and do admin tasks while organizing the conference (figure 1.6).

Figure 1.6 Conference platform back office page.

This application is composed of a set of services that have different responsibilities. Figure 1.7 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.

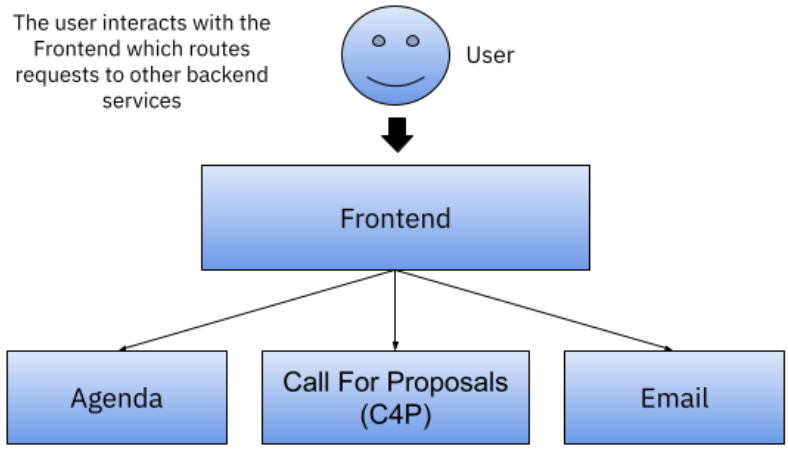

Figure 1.7 **Conference platform services.**

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

- *Frontend:* This service serves as 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, because 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 SMPT 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 will see in the following diagram, the C4P service calls both the Agenda and the Email services, hence these two services are considered "downstream" services from the C4P service perspective (figure 1.8).

From the end-user perspective, we are coordinating the interactions of two different Personas, the Potential Speaker and the Conference Organizer. The sequence of interactions between these two personas is explained in figure 1.8, but to summarize it first the Potential Speaker uses the website to submit a new proposal for a talk. The organizers are in charge of reviewing all the incoming proposals on the Conference Platform back office page. If a proposal gets approved the talk is automatically added to the conference Agenda section. No matter the decision an email is sent notifying the Potential Speaker of the approval or rejection of his/her proposal.

It is quite 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 Frontend service is acting as an API Gateway, because it is accepting all the requests from the client side and routing them to the different backend services as needed.

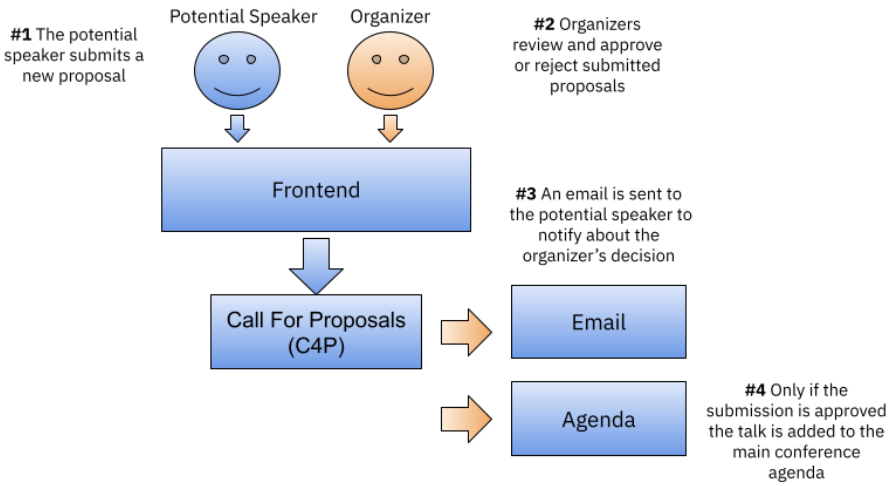

Figure 1.8 Call for Proposals use case.

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 in order 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. For improving collaboration, you need visibility, and you need to understand how the platform is working. The last few chapters of this report goes into integrating tools for monitoring, visualizing, and orchestrating your services to understand and collect business metrics from your applications.

You also need to take into consideration the operation side of this Cloud-Native application. You can imagine that 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 the application will open the attendee registration page, etc.

In section 1.2.3 when you deploy the application to a Kubernetes cluster, I will encourage you to inspect how these services are configured to work, how the data is flowing between the different services, and how to scale the services.

By playing around with a fictional application, you are free to change each service 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 for you to deploy these services to your own environment.

By having this example application up and running, you will be able to understand and experience with a concrete example of how to measure your Continuous Delivery practices such as:

- *Every change needs to trigger the feedback loop*: What kind of setup do you need to have in place to trigger these feedback loops? How do you reconcile different service feedback loops, and how do you aggregate these changes when they happen? Part 2 of the report covers how to define and implement an end-to-end deployment pipeline.
- *The feedback loop needs to be fast*: Where do you test? What do you test? And how do you stop promoting artifacts when tests go wrong? Part 2 of the report covers tools like Argo CD which will monitor our environments to make sure that our services are up, but we need to have a clear strategy on how to process issues when we find them.
- *Everything needs to be measured*: What do you measure? When do you measure? And how do you make sure that new changes are not taking your application in the wrong direction? Shifting-left observability and monitoring is something that can help us to have the right data to remediate when things are not going as we expected. Collecting the right measurements is key to understanding if we are improving or not.

Let's go ahead and get this application up and running on a Kubernetes Cluster. In my experience, the only way to fully understand how to optimize our development and delivery practices is by actually having something running and then looking at the steps to start the optimizations and automation whenever possible.

1.3.2 Installing Kubernetes KinD locally

There are several options to create Local Kubernetes Clusters for development. This report has chosen KinD because it supports different platforms and has ease of customization to run your clusters with minimum dependencies, because 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 full-fledged Kubernetes cluster, I encourage you to use that one instead, and you can skip the following section on KinD and move straight to Installing the walking skeleton using 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 on 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 on their website: https://helm.sh/docs/intro/install/.

Once you 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 on your laptop/pc and set 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 1.9. We want to be able to see in which nodes our application services are placed inside our Kubernetes cluster.

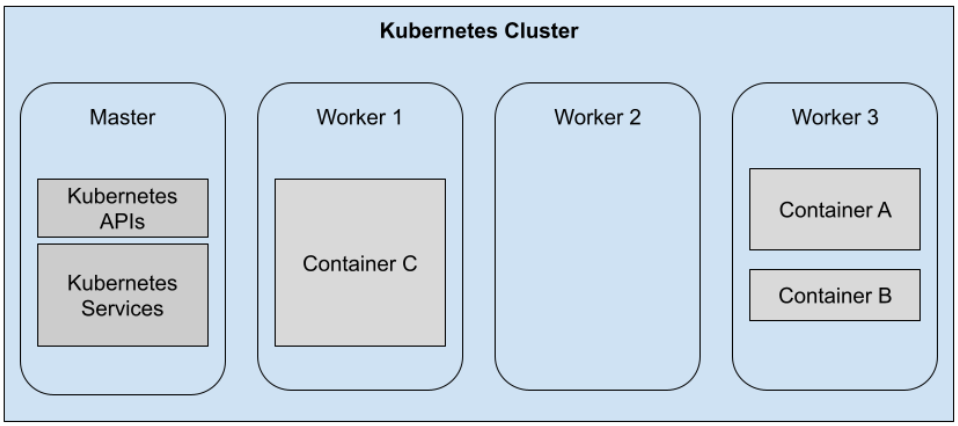

Figure 1.9 **Kubernetes cluster topology.**

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 application's downtimes. Because you are running a local Kubernetes cluster, this has limitations, such as your laptop/pc's available CPUs and memory. In real-life clusters, each of these nodes is a different physical or virtual machine that can run in different locations to maximize resilience.

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

```
cat <<EOF | kind create cluster --name dev --config=-
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
      kubeletExtraArgs:
        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 the 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 1.10 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/
```

Figure 1.10 **KinD cluster is created.**

To connect your kubectl CLI tool with this newly created, you might need to run:

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

You should see something similar to figure 1.11.

```
Kubernetes control plane is running at https://127.0.0.1:50084
CoreDNS is running at https://127.0.0.1:50084/api/v1/namespaces/kube-system/services/
kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Figure 1.11 Setting the context for kubectl.

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 figure 1.12.

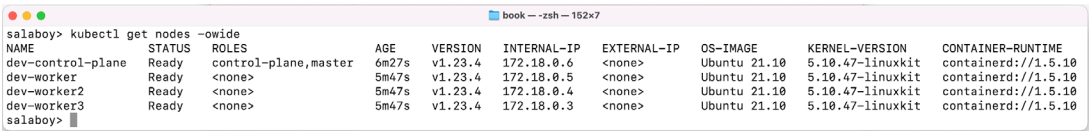

Figure 1.12 Listing all Kubernetes nodes.

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 "-owide" 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 Controller 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/ingress-nginx/master/deploy/
     static/provider/kind/deploy.yaml
```

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" (figure 1.13):

```
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/ingress-nginx-
     admission created
```

Figure 1.13 **Installing the NGINX Ingress Controller.**

Figure 1.14 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 figure 1.14.

```
NAME                                        READY   STATUS      RESTARTS   AGE   IP           NODE
ingress-nginx-admission-create-dwrgc        0/1     Completed   0          17h   10.244.1.2   dev-worker
ingress-nginx-admission-patch-mm5jg         0/1     Completed   2          17h   10.244.2.2   dev-worker3
ingress-nginx-controller-8695d45448-p6v2w   1/1     Running     0          17h   10.244.0.5   dev-control-plane
```

Figure 1.14 **Ingress Controller running in the control-plane node.**

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.

1.3.3 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 allow us to choose which container (or containers) 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 report to cover an in-detail view of these resources, as there are other resources available. In this report, 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 (figure 1.15) 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 1.15 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.

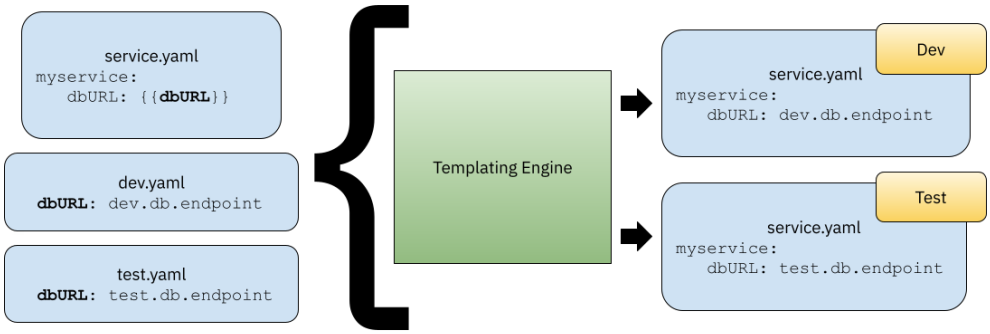

Figure 1.15 Templating engines render YAML resources by replacing variables.

Using a templating engine can save you a lot of time maintaining different copies of the same file; 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 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 me a way to browse available packages and their metadata so I can decide which package I want to install. Once I've decided which package I want to use, I should be able to download it and then install it. Once the package is installed, I would expect, as a user, to be able to upgrade to a newer version of the package if it becomes available. Upgrading/updating a package is something that usually requires manual intervention, meaning that as a user I would implicitly 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 three 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 1.16).

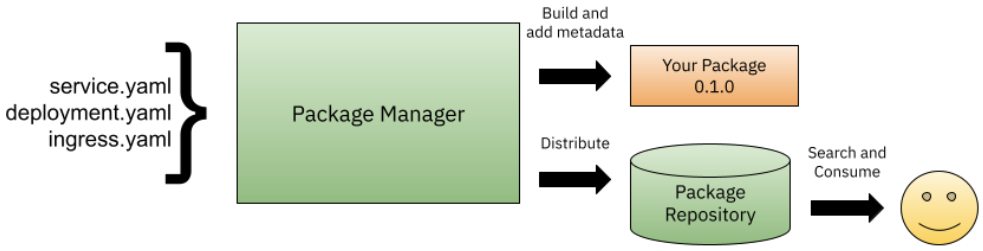

Figure 1.16 **Package Managers' responsibilities: build, package, and distribute.**

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 1.17).

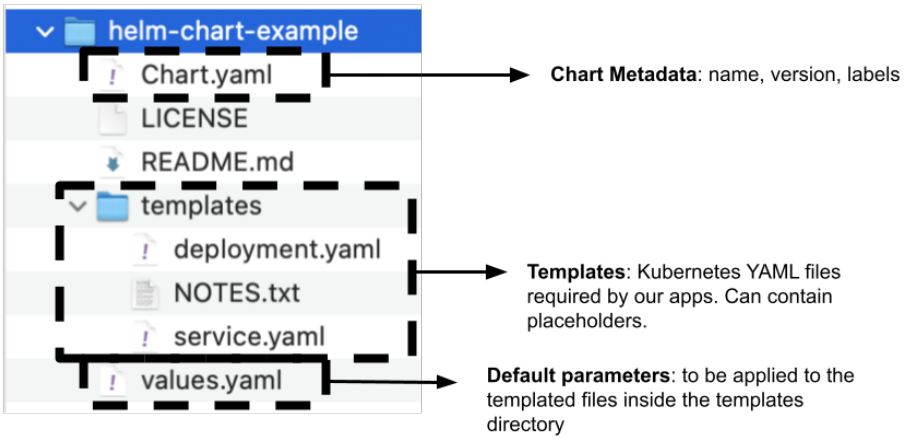

Figure 1.17 A simple Helm chart.

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 could be Maven Central, Nexus, or Artifactory.

```
helm repo add fmtok8s https://salaboy.github.io/helm/
helm repo update
```

You should see the output shown on figure 1.18.

```
salaboy> helm repo add fmtok8s https://salaboy.github.io/helm/
"fmtok8s" has been added to your repositories
salaboy> 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!⎈
```

Figure 1.18 Adding a custom Helm repository.

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 the same way I am doing it in the following repository: https://github.com/salaboy/helm. Check the repository README.md for more details about how this works.

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 that 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
```

Now, let's install our walking skeleton.

Installing the Conference Platform 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 earlier in this section. 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 use this application throughout the report 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 it 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 (figure 1.19).

```
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
```

Figure 1.19 **Helm installed the chart fmtok8s-conference version 0.1.0.**

`helm install` creates a Helm Release, which means that you have created an application instance, in this case, the instance is called "app". 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 figure 1.20.

```
NAME        NAMESPACE  REVISION  UPDATED                                  STATUS    CHART                          APP VERSION
conference  default    1         2022-06-22 16:07:02.129083 +0100 BST     deployed  fmtok8s-conference-chart-v0.1.0  0.1.0
```

Figure 1.20 **List Helm releases.**

NOTE If instead of using `helm install` you run `helm template <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 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 figure 1.21:

```
NAME                                              READY   STATUS    RESTARTS      AGE    IP            NODE          NOMINATED NODE
conference-fmtok8s-agenda-service-57576cb65c-hzfxv 1/1     Running   0             112s   10.244.2.11   dev-worker3
conference-fmtok8s-c4p-service-6c6f9449b5-z22jk    1/1     Running   3 (86s ago)   112s   10.244.2.10   dev-worker3
conference-fmtok8s-email-service-6fdf958bdd-8622z  1/1     Running   0             112s   10.244.1.11   dev-worker
conference-fmtok8s-frontend-5bf68cf65-s7rn2        1/1     Running   0             112s   10.244.3.13   dev-worker2
conference-postgresql-0                           1/1     Running   0             112s   10.244.3.15   dev-worker2
conference-redis-master-0                         1/1     Running   0             112s   10.244.1.14   dev-worker
conference-redis-replicas-0                       1/1     Running   0             112s   10.244.1.15   dev-worker
```

Figure 1.21 Listing application pods.

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 because the C4P and Agenda services need persistent storage. Besides the services, we will have these two databases running inside our Kubernetes Cluster. 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 1.22:

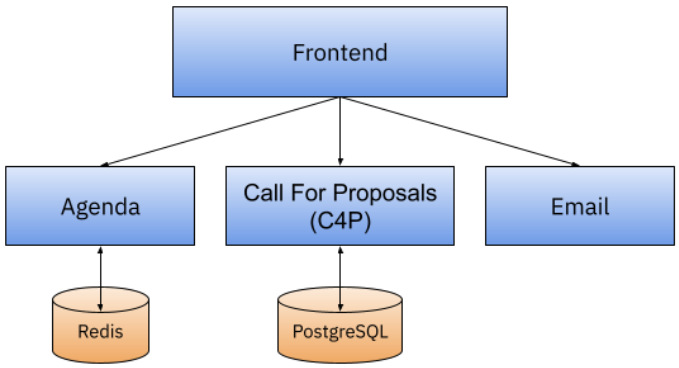

Figure 1.22 Application services and databases.

On figure 1.21, 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 because 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.

Notice that pods 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 pods 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 require, there is a step-by-step tutorial on how to deploy this application to a Kubernetes cluster using Helm, which you can find at the following repository: https://github.com/salaboy/from-monolith-to-k8s/tree/main/helm.

1.3.4 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 figure 1.23:

Figure 1.23 **Conference main page.**

If you switch to the Agenda section now you should see something like figure 1.24:

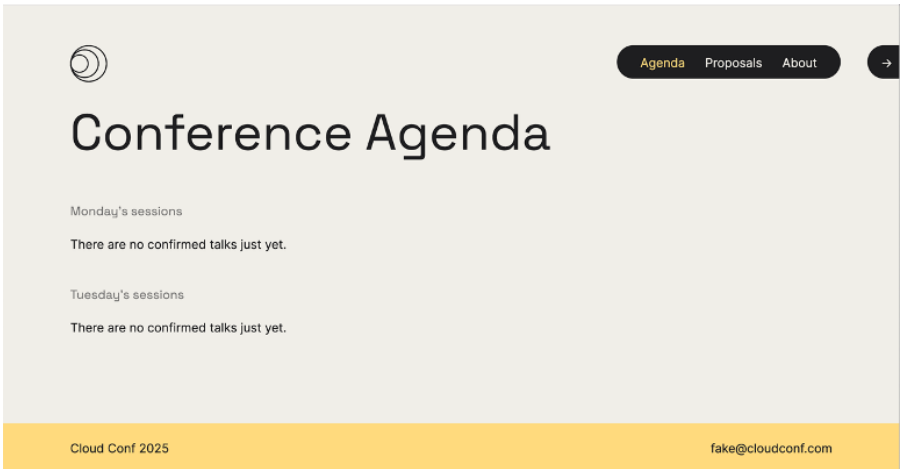

Figure 1.24 **Conference empty Agenda when we first install the application.**

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 (figure 1.25). 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.

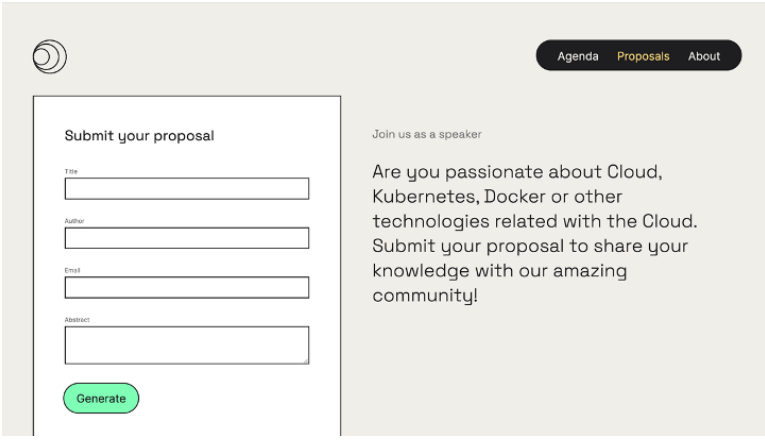

Figure 1.25 **Submitting a proposal for organizers to review.**

Notice that there are four fields (Title, Author, Email, and Abstract) in the form that you need to fill in 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 in figure 1.26:

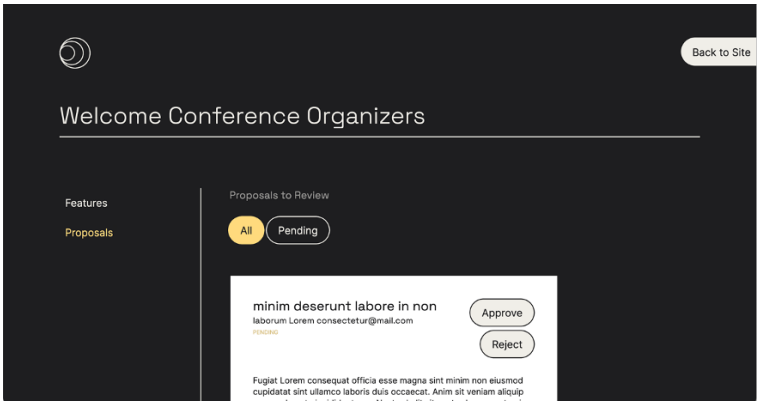

Figure 1.26 Conference organizers can Accept or Reject incoming proposals.

Accepted proposals will appear on the Main Page. Attendees who visit the page at this stage can see the conferences main speakers (figure 1.27).

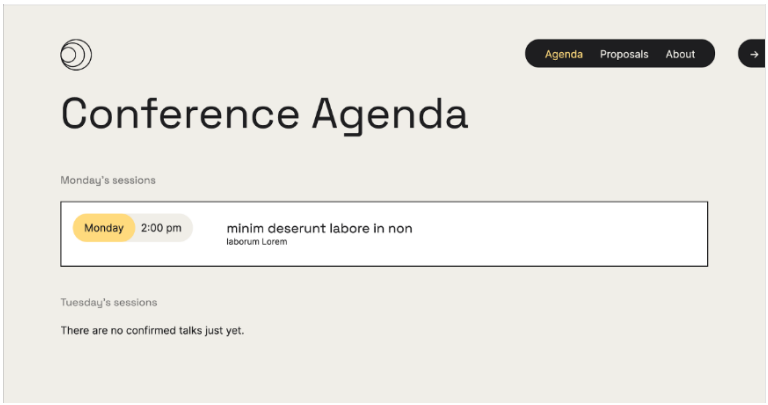

Figure 1.27 Your proposal is now live on the agenda!

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 (figure 1.28):

```
kubectl logs -f app-fmtok8s-email-rest-<POD_ID>
```

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

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.7.0)

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 aliquip ipsum ipsum offi
cia esse dolore consequat qui cillum 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
```

Figure 1.28 **Email Service logs.**

If you made it so far, congrats, the Conference Platform 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 installed the Conference Platform 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, and by using Helm we can parameterize some applications configurations, in the second part of this report we will see how to tweak some of the available configurations. But before moving forward, and now that we have an application running, we need to dig deeper into what kind of challenges we will face when building distributed applications like the one that we have just installed.

1.4 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's infrastructure requirements to efficiently allow Kubernetes to scale up and down our services.
- *Data 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 of 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.

1.4.1 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-frontend
```

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 1.29).

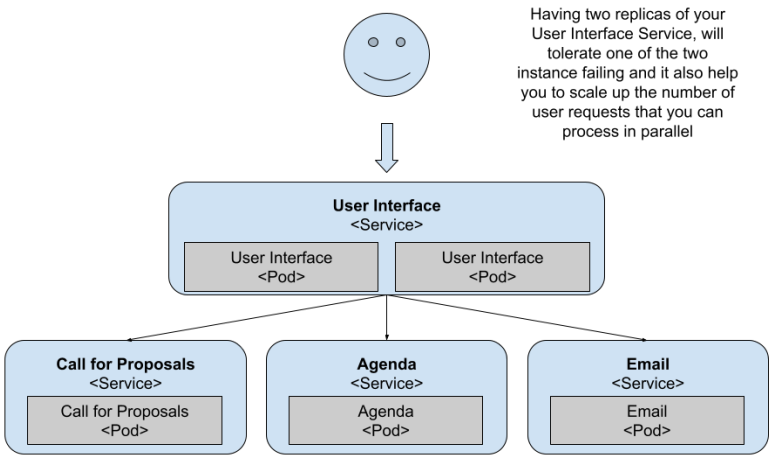

Figure 1.29 Two replicas for the Frontend service.

You can quickly try this self-healing feature of Kubernetes by killing one of the two pods of the application Frontend (figure 1.30). You can do this by running the following commands:

```
kubectl get pods
```

```
NAME                                                READY   STATUS    RESTARTS   AGE
conference-fmtok8s-agenda-service-57576cb65c-sl27p  1/1     Running   0          6h18m
conference-fmtok8s-c4p-service-6c6f9449b5-j6ksv     1/1     Running   0          6h18m
conference-fmtok8s-email-service-6fdf958bdd-4pzww   1/1     Running   0          6h18m
conference-fmtok8s-frontend-5bf68cf65-pwk5d         1/1     Running   0          2m9s
conference-fmtok8s-frontend-5bf68cf65-zvlzq         1/1     Running   0          6h18m
conference-postgresql-0                             1/1     Running   0          6h18m
conference-redis-master-0                           1/1     Running   0          6h18m
conference-redis-replicas-0                         1/1     Running   0          6h18m
```

Figure 1.30 Checking that the two replicas are up and running.

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

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

Then list the pods again:

```
kubectl get pods
```

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 1.31).

```
NAME                                              READY   STATUS        RESTARTS   AGE
conference-fmtok8s-agenda-service-57576cb65c-sl27p  1/1     Running       0          6h22m
conference-fmtok8s-c4p-service-6c6f9449b5-j6ksv     1/1     Running       0          6h22m
conference-fmtok8s-email-service-6fdf958bdd-4pzww   1/1     Running       0          6h22m
conference-fmtok8s-frontend-5bf68cf65-8j7lt         0/1     Running       0          2s
conference-fmtok8s-frontend-5bf68cf65-fc295         1/1     Terminating   0          52s
conference-fmtok8s-frontend-5bf68cf65-zvlzq         1/1     Running       0          6h22m
conference-postgresql-0                           1/1     Running       0          6h22m
conference-redis-master-0                         1/1     Running       0          6h22m
conference-redis-replicas-0                       1/1     Running       0          6h22m
```

Figure 1.31 A new replica is automatically created by Kubernetes as soon as one goes down.

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 (figure 1.32). You can revert back to a single replica with:

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

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

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

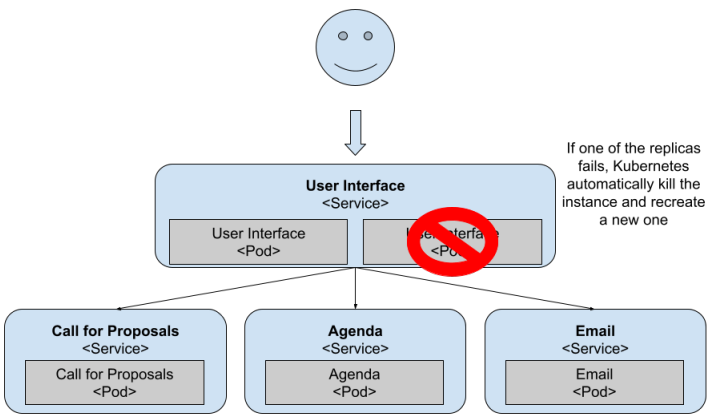

Figure 1.32 With a single replica being restarted, there is no backup to answer user requests.

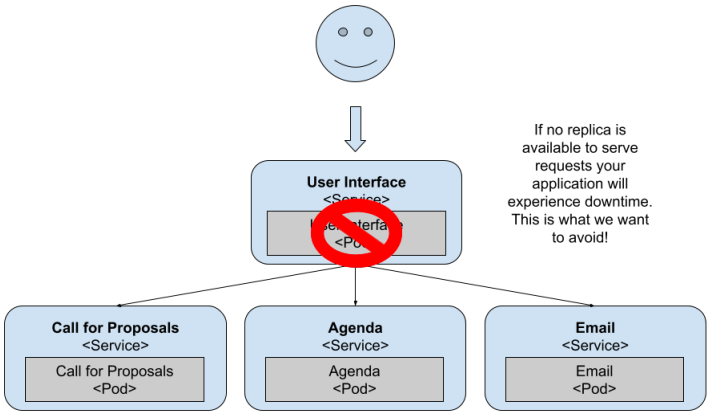

Figure 1.33 With a single replica being restarted, there is no backup to answer user requests.

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, because the Ingress Controller (not shown in the previous figure for simplicity) cannot find a replica running behind the API Gateway service (figure 1.34). If you wait for a bit, you will see the application come back up.

This behavior is to be expected, because 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 API Gateway / FrontEnd service is the most important service of the entire application because 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 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.

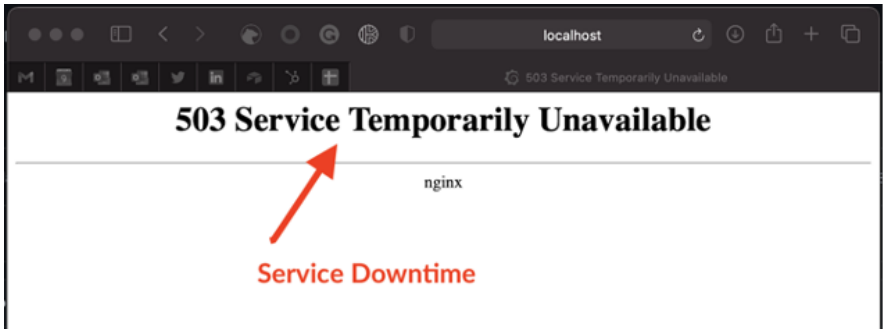

Figure 1.34 With a single replica being restarted, there is no backup to answer user requests.

1.4.2 Building in a service's resilience

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, because the Agenda List is right there on the main page of the application (figure 1.35). So, let's scale the service down:

```
kubectl scale --replicas=0 deployments/conference-fmtok8s-agenda-service
```

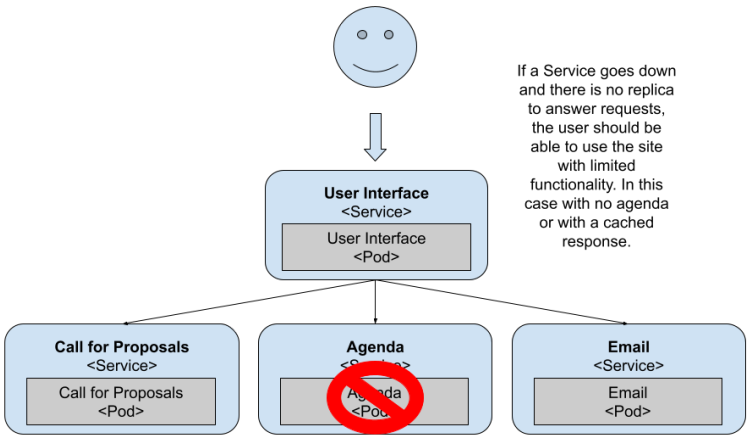

Figure 1.35 No pods for the Agenda service.

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 1.36):

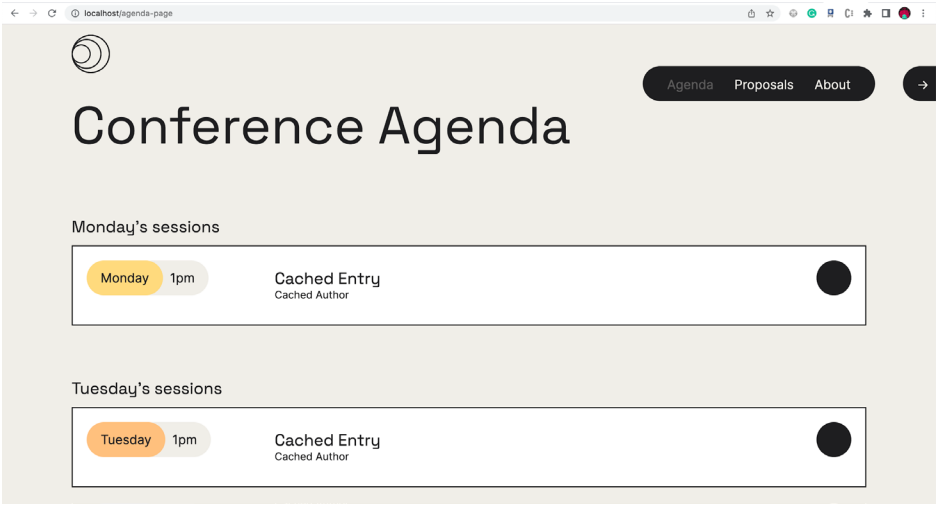

Figure 1.36 If the Agenda service has no replica running, the Frontend is wise enough to show the user some cached entries.

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 because 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 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.

1.4.3 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-agenda-service
```

If you 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 are storing all the proposals and agenda items in external databases. In this context, external means outside of the pod memory.

Now, what do you think will happen if we scale the Agenda Service up to two replicas (figure 1.37):

```
kubectl scale --replicas=2 deployments/conference-fmtok8s-agenda-service
NAME                                                READY   STATUS    RESTARTS   AGE
conference-fmtok8s-agenda-service-57576cb65c-7pfx2  1/1     Running   0          45s
conference-fmtok8s-agenda-service-57576cb65c-cq75h  1/1     Running   0          8m4s
conference-fmtok8s-c4p-service-6c6f9449b5-j6ksv     1/1     Running   0          6h38m
conference-fmtok8s-email-service-6fdf958bdd-4pzww   1/1     Running   0          6h38m
conference-fmtok8s-frontend-5bf68cf65-zvlzq         1/1     Running   0          6h38m
conference-postgresql-0                             1/1     Running   0          6h38m
conference-redis-master-0                           1/1     Running   0          6h38m
conference-redis-replicas-0                         1/1     Running   0          6h38m
```

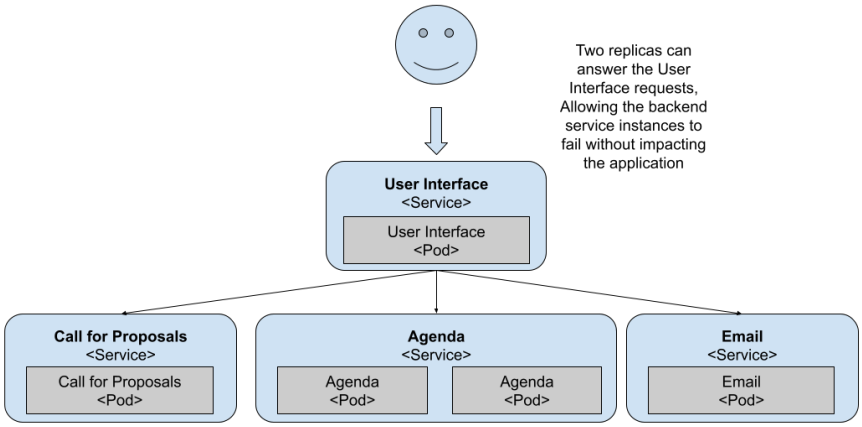

Figure 1.37 Two replicas can now deal with more traffic.

With two replicas dealing with your user requests, now the Frontend will have two instances to query. Kubernetes will take care of doing the load balancing between the two replicas, but your application will have no control over which replica is hitting. Because we are using a database to back up the data outside of the context of the service pod we can scale the replicas to a number of pods that can deal with the application demand (figure 1.38).

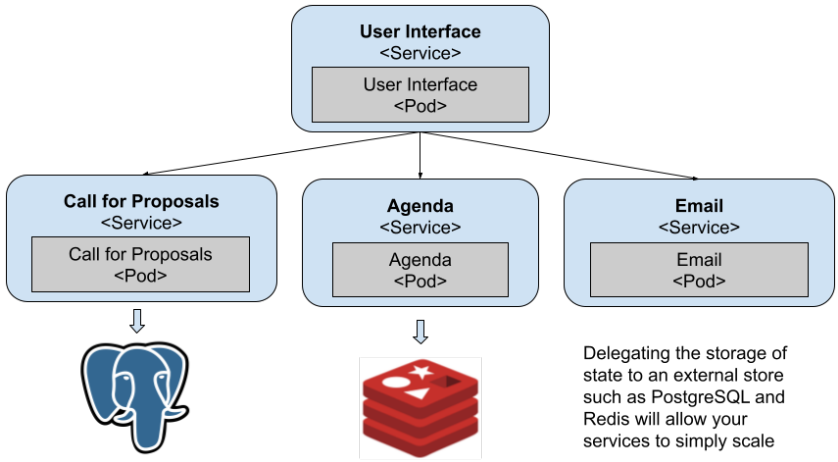

Figure 1.38 Both data sensitive services use persistent stores.

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 start scaling up the Agenda service pods (figure 1.39)?

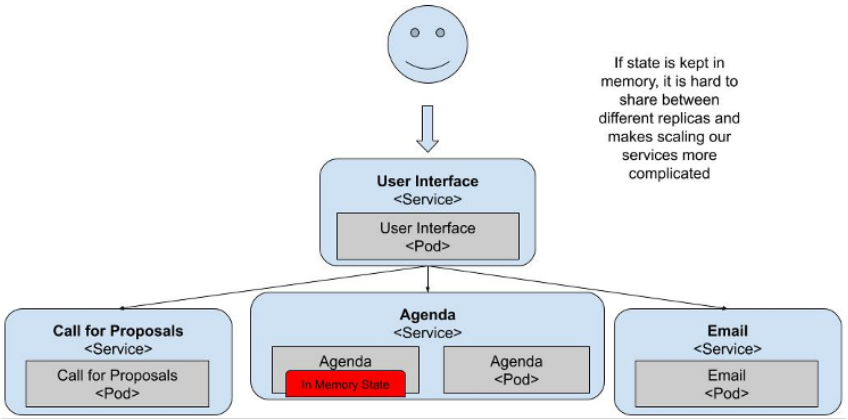

Figure 1.39 What would happen if the Agenda service keeps state in-memory?

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 balances the requests across different replicas, the frontend 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 because 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 saw before, we installed Redis and PostgreSQL as part of the application Helm chart, and this is usually not a good idea as databases and tools like message brokers will need special care by the Operations team that can choose not to run these services inside Kubernetes.

1.4.4 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. Because 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 1.40).

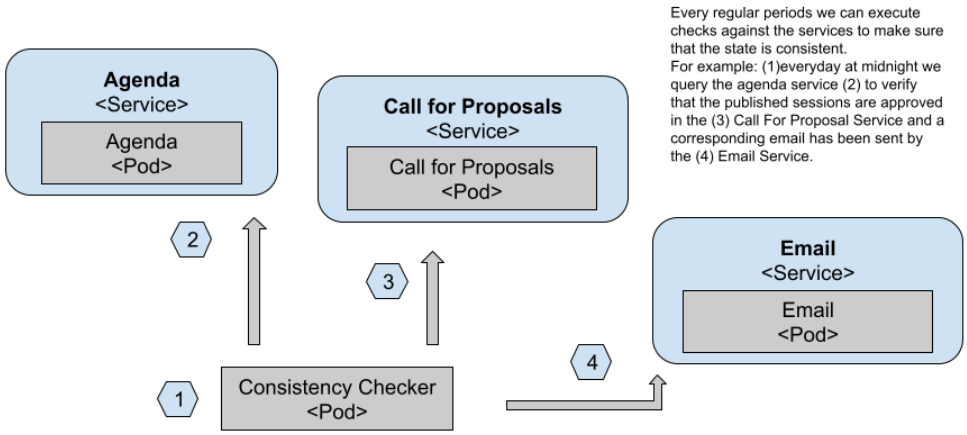

Figure 1.40 **Consistency checks can run as CronJobs.**

In figure 1.40 we can see how a CronJob (1) will be executed every X period of time, depending on how important it is for us to fix consistency issues. Then it will proceed to 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 that 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 were sent.

1.4.5 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 are working at any given time.

The OpenTelemetry (https://opentelemetry.io/) 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 because 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.

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 1.41.

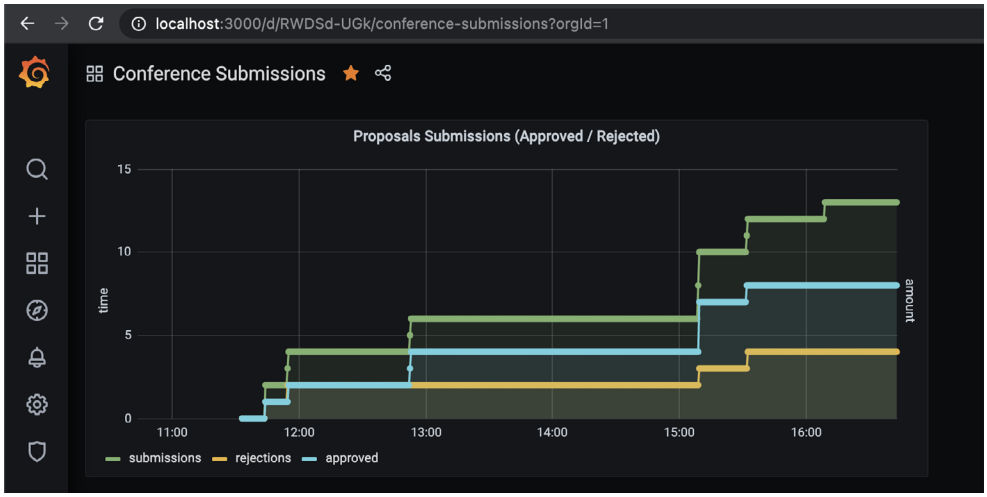

Figure 1.41 Monitoring telemetry data with Prometheus and Grafana.

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 time 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 1.42).

Once the instrumentation libraries are included in our services, we can check the traces with tools like Jaeger (figure 1.43). You can see the amount of time used by each service while processing a single user request.

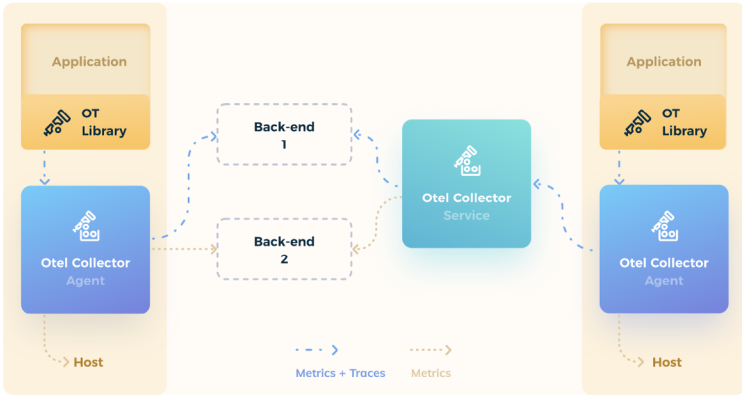

Figure 1.42 OpenTelemetry architecture and library.

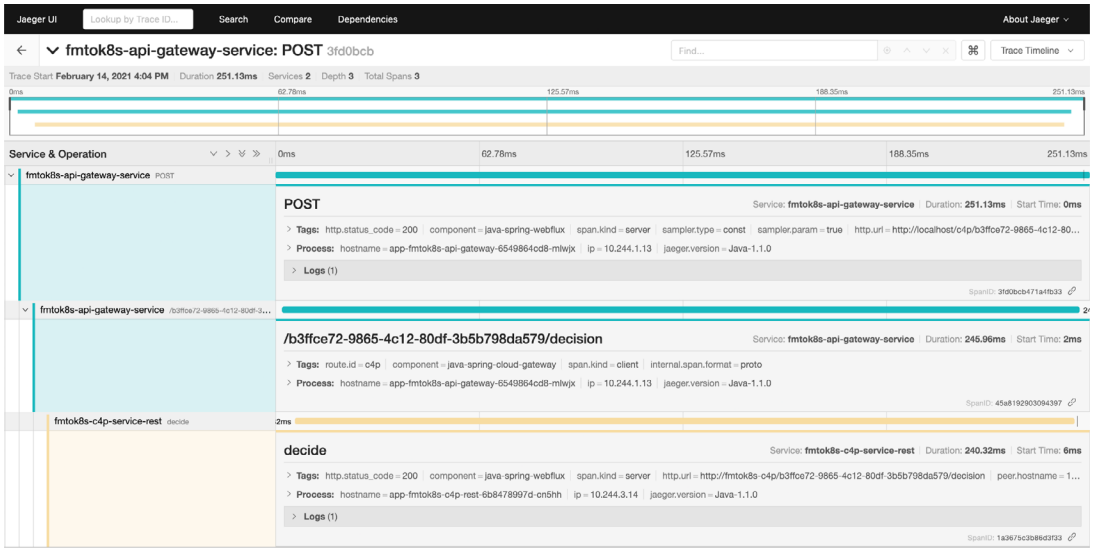

Figure 1.43 Tracing diagram from Jaeger.

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.

1.4.6 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, because authorization and the user identity need 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 API Gateway 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 API Gateway to connect with an authorization and 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 log into our applications without the need for them to create a separate account. This is usually known as social logic.

There are some popular solutions that brings both OAuth2 and Identity Management together such as Keycloak (https://www.keycloak.org/) and Zitadel (https://zitadel.com/opensource). 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 probably will have, sooner or later) including single sign-on and identity management into the walking skeleton will push you to think about the specifics of "who will be able to do what", refining your use case even more.

1.5 Running Cloud-Native applications

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.

Now that we have a Cloud-Native application up and running and we understand some of the advantages and challenges that we get when working with these applications on top of Kubernetes, we are ready to jump into part two, which covers what it takes to deliver new features to these applications.

If we want to map what we have done in part 1 to our Continuous Delivery journey, we can say that:

- We created a development environment (in our local pc/laptop).
- We deployed a Cloud-Native application that was already packaged using Helm.
- We interacted with an instance of our Cloud-Native Conference platform as end-users and understood its limitations and challenges.

In some way, starting the journey by deploying the application to a local environment might seem counterintuitive, because to deploy the application somebody must have code, build, and package the application. But if we go all the way back to section 1.2, we should be measuring our practices to deliver business value. In this case, we have managed to create a new instance of the application by creating a local KinD cluster and running two lines to install the application in the cluster. If your business is to provide these instances as a service for customers, this should be one of your key measurements (figure 1.44).

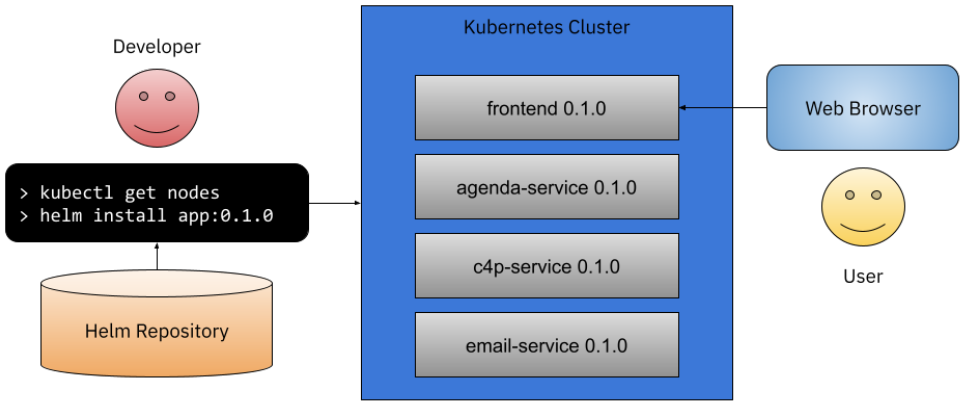

Figure 1.44 **Creating a new instance of our application.**

It is important to understand that the steps that we 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 make sure that developers are focused on building new features and improving our application. Figure 1.45 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 in part 2 of this report.

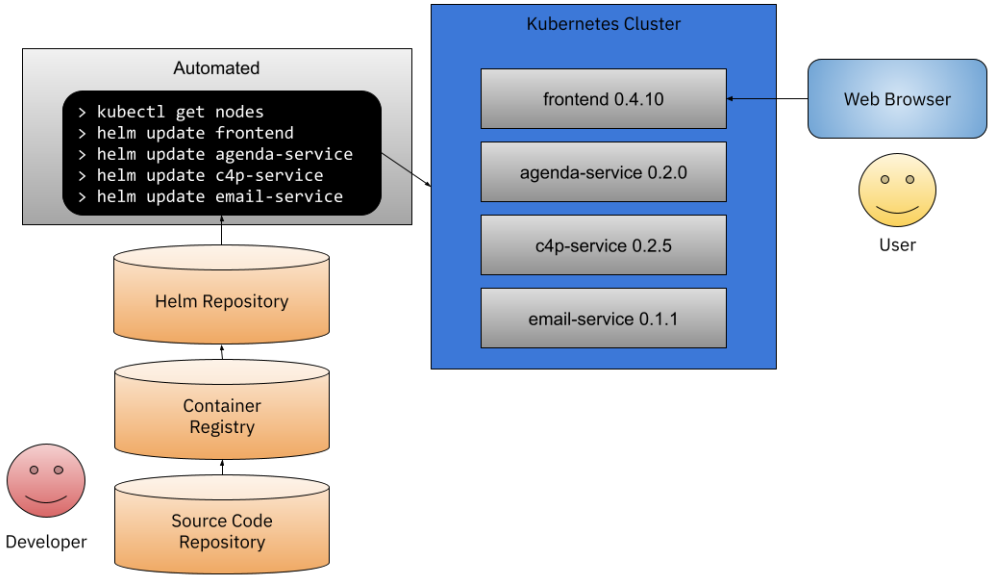

Figure 1.45 **Developers should focus on building features, so we need to automate the entire process.**

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 figure 1.45, we will not be installing a static application definition, we want to release and deploy new versions of the services as they become available.

Part 2 covers the tools that we need in order to update our application's services code and release new versions of the services that can be deployed in any number of environments. We will also look at how to create these environments and which tools can be used to manage them and keep them always in sync.

2 Delivering Cloud-Native applications

In part 1, you installed and interacted with a simple distributed Conference Platform composed of four services. This second part covers what it takes to deliver each of these components in a continuous delivery fashion by using Pipelines as delivery mechanisms. We will start this second part by describing and showing in practice how each of these services can be built, packaged, released, and published so they can run in your organization's environments.

We will look at two main concepts: *Service Pipelines* and *Environment Pipelines*. The Service Pipeline takes care of all the steps needed to build your software from source code until the artefacts are ready to run. Environment Pipelines, on the other hand, cover the aspects of dealing with the installation and upgrade of new versions of each of these services into a live environment such as staging, testing, and production.

This second part is divided into four main sections:

- What does it take to deliver a Cloud-Native application?
- Pipelines
  - What is a Service Pipeline?
    - Tekton as the Cloud-Native Pipeline Engine
    - Service Pipelines in action with Tekton
  - What is an Environment Pipeline?
    - Environment pipeline requirements
    - ArgoCD for automated deployments

- Release strategies
  - Kubernetes built-in mechanisms
  - Using Argo Rollouts implementing different release strategies such as Canary Releases and Blue/Green deployments

In part 1 we started by asking, "Are you Cloud-Native?" Here in Part 2, we'll begin with something a bit more advanced: How do you continuously deliver a Cloud-Native application?

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

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 have each service up and 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 the second half of this chapter are designed to automate all the tasks involved, from source code to a service that is up and running inside Kubernetes.

This part is focused on describing 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 teams* 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 services* that are running. 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 that 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 will need to be provisioned, configured, and maintained in-house.

- *Managing and provisioning different environments for various purposes such as development, testing, QA, and production.* If you really 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 are hitting 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 worry and waste time every time that 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 about adding or refactoring services. This automation is usually performed by what is commonly known as *Pipelines*. As shown in figure 2.1, these pipelines describe what needs to be done to build and run our services, and usually they can be executed without human intervention.

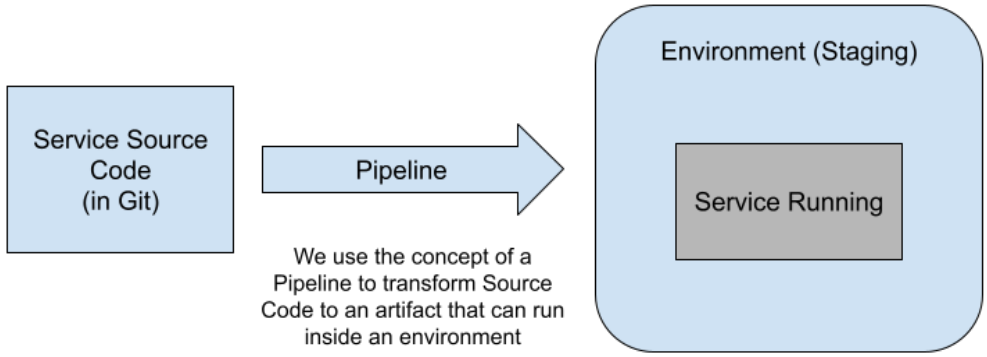

Figure 2.1 From source to a running service using a pipeline.

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 next two sections are 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 quite 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, 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 2.2, we can see the Pipeline Engine picking up our pipeline definition and creating different instances that can be parameterized differently to have 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.

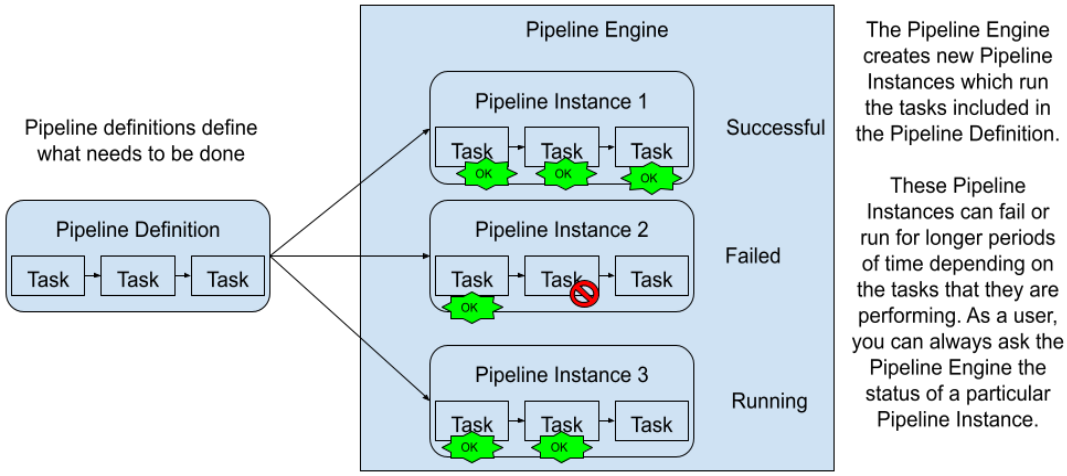

Figure 2.2 A pipeline definition can be instantiated by a Pipeline Engine multiple times.

As expected, with pipeline definitions we can create loads of different automation solutions, but how do these concepts 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.

Service and Environment Pipelines are executed on top of different resources and with different expectations. The following sections go into more detail about the steps that we will need to define for our service and environment pipelines, because these steps will be similar no matter which technology stack we are using. Let's look at the archetype of a Service Pipeline definition.

2.2 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 need to 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, because they will probably be reinventing something that has been already defined, tested, and improved by other teams.

As we will see in this section, 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" references the fact that each of our application 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 from end to end.

> **NOTE** It is tempting to think about creating a single pipeline for the entire application (collection of services), as we did with monolith applications, but 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, because it will block your ability to release services independently.

2.2.1 Conventions will save you time

Service Pipelines can be more opinionated on how they are structured and what is their reach, and 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 that 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 which is 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.

- *One service/one repository/one service 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, as well as 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. A quick note about mono-repositories, using one repository per service might become too complicated to manage for some organizations; hence, some people like having a single repository with all the services and then one service pipeline configured to build that specific service. I am not a fan of either of these approaches, but you need to keep an eye on bottlenecks. If too many repositories (one for each service) becomes a burden, you should check best practices around mono-repositories. If a single repository and who can commit what and when becomes something that is slowing down your teams, you might want to separate the code into separate repositories to avoid pushing different teams to implement complex coordination mechanisms to push (or merge) new code to the code base.
- *Consumer-driven contract testing:* Your service uses contracts to run tests against other services, so unit testing an individual service shouldn't require having other services up and running. By creating consumer-driven contracts, each service can test its own 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.

If we take these practices and conventions into account, we can define the responsibility of a Service Pipeline as follows: "Transform source code to an artifact that can be deployed in any environment".

2.2.2 Service pipeline structure

With this definition in mind, let's 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 and 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 that 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 that 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 will need to build a container image. The most common way of doing this today is using Docker. This step requires the source code repository to have, for example, a Dockerfile defining how this container image needs to be built.
- *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 it can be accessed by others. This container image will have the same version as the tag that was created in the repository and the binary that was published. This helps us to clearly see which source code will run when you run the container image.
- *Lint, verify, and 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 a Kubernetes manifest that defines how the containers are going to be deployed into a Kubernetes Cluster. If you are using a package manager such as Helm, you can version the package with the same version used for the binaries and the container image.
- *(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 you are trying to automate all the way from source to a service running, the Service Pipeline (figure 2.3) should be able to send a notification to all the interested services who might be waiting for new versions to be deployed.

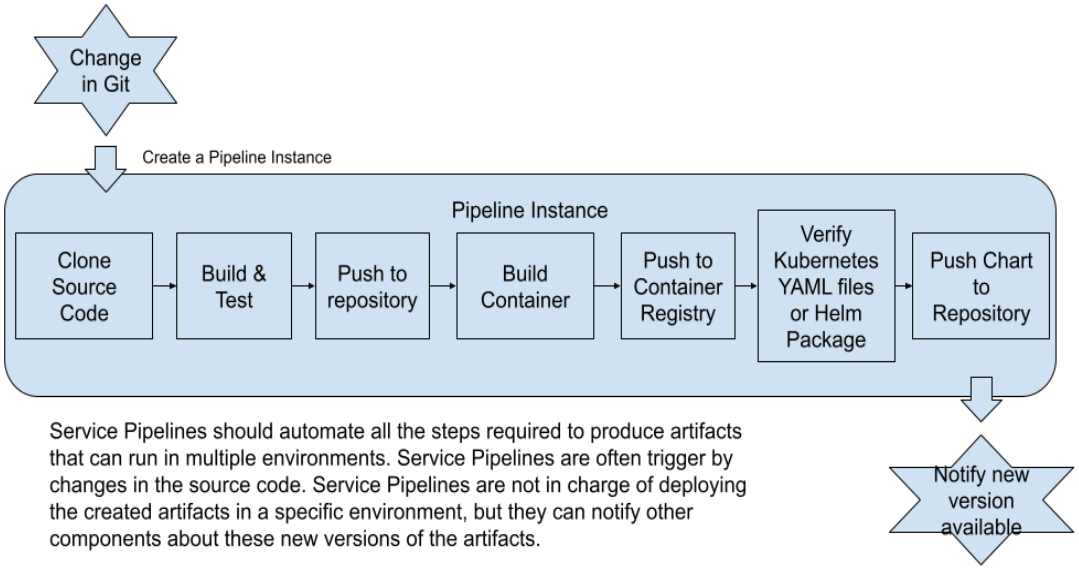

Figure 2.3 **Tasks expected for a service pipeline.**

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 itself needs to be built and packaged in a way that is not dependent 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.

2.2.3 Service Pipelines in real life

In real life, this pipeline will need to run every time that you merge changes to the main branch of your repository. This is how it should work if you follow a trunk-based development approach (figure 2.4):

- When you merge changes to your main branch, this pipeline should run, creating a new release for your software. This means that you shouldn't be merging code into your main branch if it is not releasable.

- 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. 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.

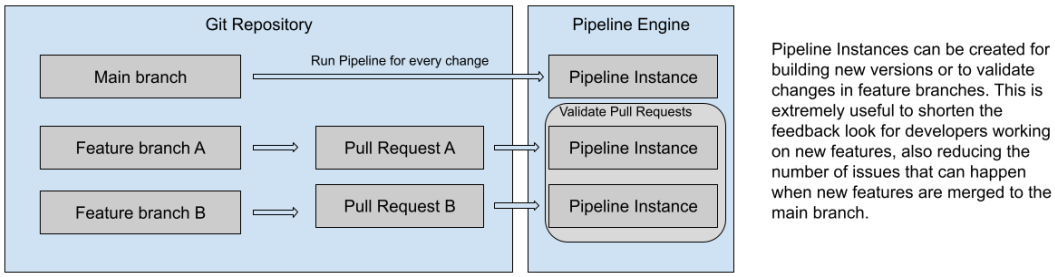

Figure 2.4 Service pipelines for main branch and feature branches.

This Service Pipeline, shown in figure 2.4, represents the most common steps that you will need to execute every time you merge something to the main branch, but 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/change request:* The pipeline will validate that the pull requests/change requests 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 and change requests can be a good option to avoid running pipelines for every change in the feature branches, because when the developer(s) is ready to get feedback from the build system, they can create a PR and that will trigger the pipeline. If developers made changes on top of the pull request, the pipeline would be retriggered.

Despite small 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 that is being produced can be deployed to an environment.

2.2.4 Service Pipelines requirements

This section covers the infrastructural requirements for service pipelines to work as well as 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 (figure 2.5):

- *Webhooks for source code changes notifications:* First, 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 an artifact repository where all artifacts are stored. This requires having an artifact repository configured and the valid credentials to be able to push new artifacts to it.
- *Docker 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. Having a container registry available with valid credentials is needed to accomplish this step.
- *Helm Chart repository and valid credentials:* Kubernetes manifests can be packaged and distributed as Helm charts, so if you are using Helm you will need to have a Helm Chart repository and valid credentials to push these packages.

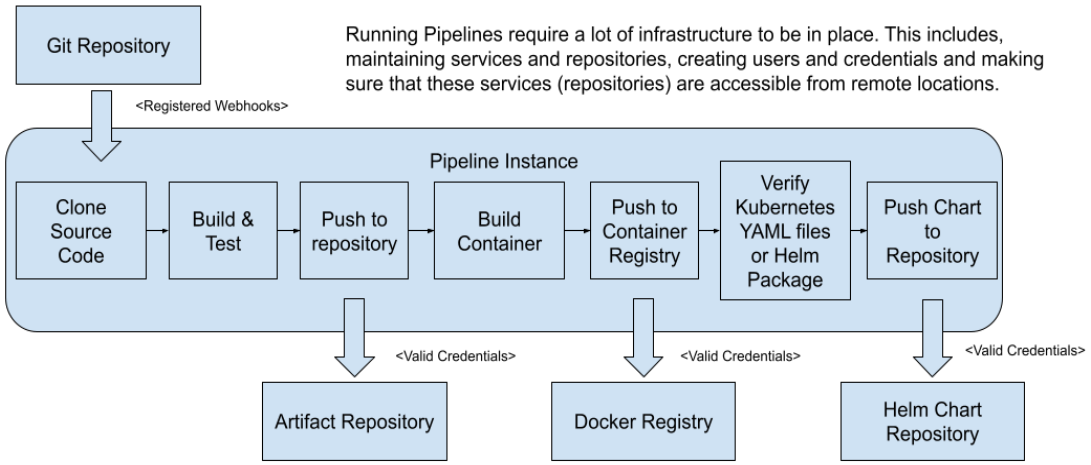

Figure 2.5 Service pipelines required infrastructure.

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 deploy the service into Kubernetes. 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 (figure 2.6).

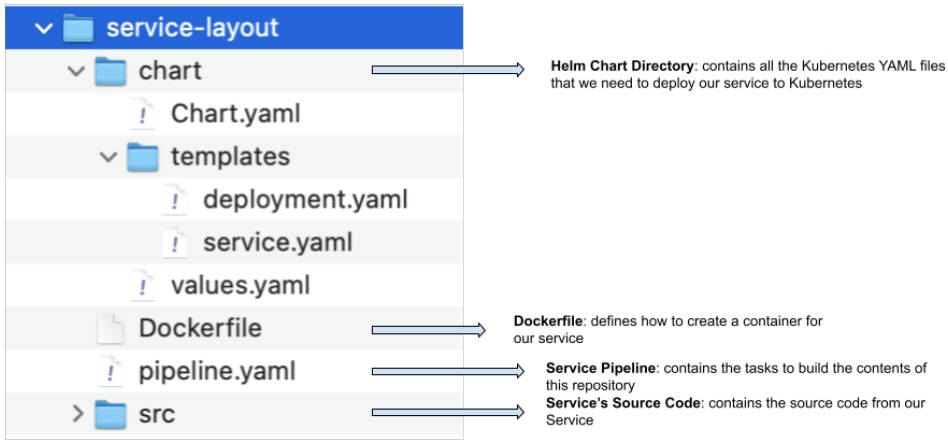

Figure 2.6 The service source code repository needs to have all the configurations for the service pipeline to work.

Figure 2.6 shows a possible directory layout of our service source code repository, which includes the source (src) directory which will contain all the files that will be compiled into binary format. The Dockerfile 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.

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.

A more advanced but very useful setup might include running the pipeline for pull requests (change requests) which can include deploying the artifact to a "Preview" environment where developers and other stakeholders can validate the changes before merging them to the main branch. Projects like Jenkins X provide this feature out-of-the-box.

In summary, Service Pipelines are in charge of building our source and related artifacts so they can 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 which we will cover after looking at a couple of different implementations for service pipelines.

2.2.5 Service Pipelines in action

There are several Pipeline Engines out there and even full managed services like GitHub Actions (https://github.com/features/actions), which will provide loads of integrations for you to build and package your application's services. In the following sections, we will look at a project called Tekton (https://tekton.dev), which was designed as a Pipeline Engine for Kubernetes. Because Tekton is a generic Pipeline Engine, you can create any kind of pipeline with it, we will be contrasting Tekton with GitHub actions.

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

2.2.6 Tekton in action

In Tekton, you have two main concepts: Tasks and Pipelines. Tekton, the Pipeline Engine is composed of a set of components that will understand how to execute tasks and pipelines 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.

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 itself that knows how to deal with tasks and pipelines resources when we create them.

You can install Tekton using Helm, thanks to a collaboration with the Continuous Delivery foundation: https://github.com/cdfoundation/tekton-helm-chart.

As with every Helm Chart, you will need to first add the Helm Chart repository and then install the chart:

```
helm repo add cdf https://cdfoundation.github.io/tekton-helm-chart/
helm repo update
helm install tekton cdf/tekton-pipeline
```

Once you install the Tekton Helm Chart, you will see that a new namespace was created. This new namespace called "tekton-pipelines" contains the pipeline controller (which is 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 a lot if you are working with multiple tasks and complex pipelines. You can follow the instructions for installations at https://github.com/tektoncd/cli.

If you now list all the Custom Resource Definitions that are installed in the cluster and belongs to tekton you will see that there is a new set of resources that you can use:

```
> kubectl get crds | grep tekton
clustertasks.tekton.dev
conditions.tekton.dev
pipelineresources.tekton.dev
pipelineruns.tekton.dev
pipelines.tekton.dev
runs.tekton.dev
taskruns.tekton.dev
tasks.tekton.dev
```

We can get all the Custom Resource Definitions (crd) using kubectl and filtering only the ones related to Tekton.

The resource types that we will be creating for our pipelines. Notice that you can get these resources by also using "kubectl get" and the resource type.

Once you have Tekton installed, you can start by creating a simple task definition in YAML. A Task in Tekton will look like a normal Kubernetes resource:

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: echo-hello-world
spec:
  steps:
    - name: echo
      image: ubuntu
      command:
        - echo
      args:
        - "Hello World"
```

The docker image called ubuntu is going to be used for this task.

The main command used by this task is echo. Notice that is an list of commands.

The command arguments (args) in this case is just a "Hello World" string. Notice that you can send a list of arguments for more complex commands.

Derived from this example, you can create a task for whatever you want, because 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/v1beta1
kind: TaskRun
metadata:
  generateName: hello-run-
spec:
  taskRef:
    name: echo-hello-world
```

Alternatively, you can use `tkn` to start a task definition:

```
salaboy> tkn task start echo-hello-world
TaskRun started: echo-hello-world-run-q7vgw
```

In order to track the TaskRun progress run:

```
tkn taskrun logs echo-hello-world-run-q7vgw -f -n default
```

And then get the logs by running the suggested command:

```
salaboy> tkn taskrun logs echo-hello-world-run-q7vgw -f -n default
[echo] Hello World
```

Whether you apply this TaskRun to the cluster (`kubectl apply -f taskrun.yaml` or using `tkn task start`), the Pipeline Engine will execute this task. On the YAML file, you can see this resource doesn't have a `metadata.name`, instead, it has a `metadata.generateName` field. When Tekton runs this task, it will generate a unique name for the resource to track that specific execution. You can keep applying the same resource and for each time you apply it to the cluster, a new execution will be scheduled. The TaskRun resources are used to keep all the information on the execution of the task definitions and the outputs for these executions.

2.2.7 Pipelines in Tekton

A Task in itself can be useful, but Tekton becomes really interesting when you create sequences of these tasks by using pipelines.

A pipeline is a collection of these tasks in a concrete sequence, let's look at a simple Service Pipeline defined in Tekton (service-pipeline.yaml):

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: service-pipeline
spec:
  resources:
    ...
  tasks:
    - name: clone-repository
      taskRef:
       name: git-clone
       bundle: gcr.io/tekton-releases/catalog/upstream/git-clone:0.4
      params:
        ...
    - name: maven-build
      runAfter: [clone-repository]
      taskRef:
       name: maven
       bundle: gcr.io/tekton-releases/catalog/upstream/maven:0.2
      params:
        …
    - name: docker-image-build-and-publish
      runAfter: [maven-build]
      taskRef:
name: kaniko
bundle: gcr.io/tekton-releases/catalog/upstream/kaniko:0.4
      params:
        …
```

You can find the full pipeline definition at https://github.com/salaboy/from-monolith-to-k8s/blob/main/tekton/resources/service-pipeline.yaml.

Once again, you will need to apply this pipeline resource to your cluster for Tekton to know about: `kubectl apply -f service-pipeline.yaml`.

As you can see in the pipeline definition, the spec.tasks field contains an array of task references. 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 own 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, because tasks need TaskRuns for the executions, you will need to create a `PipelineRun` for every time that you want to execute your pipeline.

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  generateName: service-pipeline-
```

```
spec:
  pipelineRef:
    name: service-pipeline
```

Now when you apply this file to the cluster `kubectl apply -f pipelinerun.yaml`, Tekton will execute the pipeline by running all the tasks defined in the pipeline definition. Alternatively, you can also use `tkn`:

```
salaboy> tkn pipeline start service-pipeline
```

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.

2.2.8 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

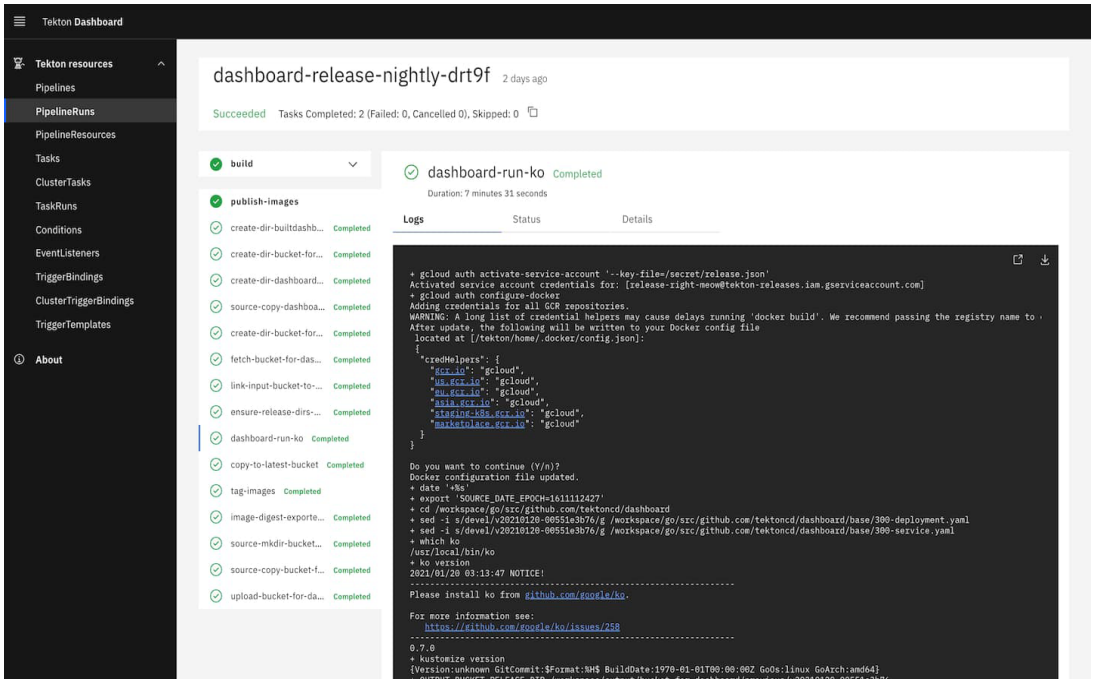

Figure 2.7 Tekton dashboard is a user interface to monitor your pipelines.

Figure 2.7 shows the community-driven Tekton dashboard that 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, but 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 and 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 that have been 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 you with detailed documentation about how to install and use these tasks in your pipelines.

Tekton Hub and the Tekton catalog allow you to reuse tasks and pipelines that had been created by a large community of users and companies (figure 2.8).

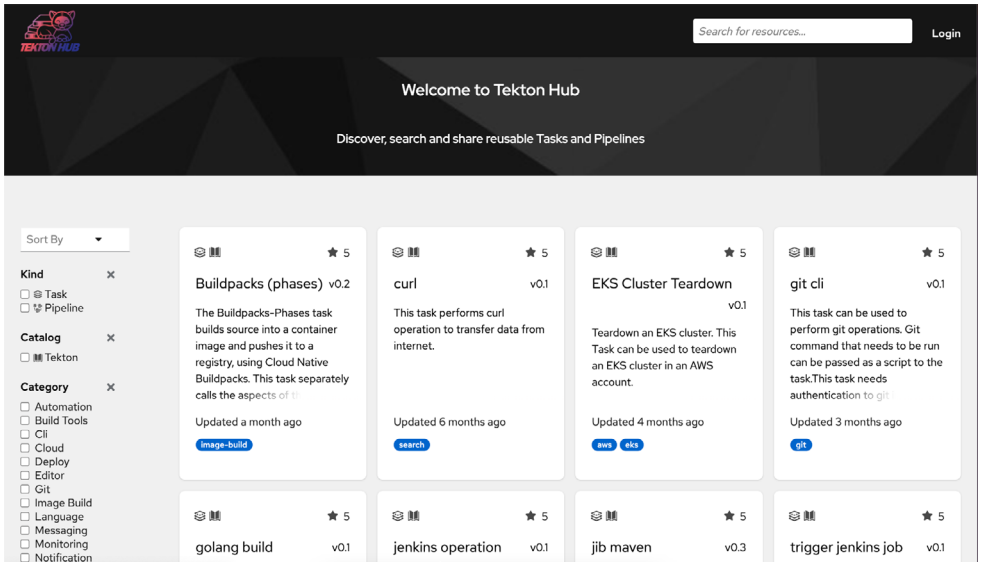

Figure 2.8 Tekton Hub is a portal to share and reuse tasks and pipeline definitions.

2.2.9 A workflow approach to automation using Argo Workflows and Argo Events

As you have seen, Tekton provides us with the basic building blocks to construct very unopinionated pipelines. In other words, we can use Tekton to build not only Service Pipelines but almost every imaginable pipeline that will leverage Kubernetes

resource-based approach, scalability, and self-healing features. But Tekton is not alone. Another project worth checking is Argo Workflows in conjunction with Argo Events, which also provide a very powerful workflow engine that can be combined with event sources for building end-to-end automations.

In Argo Workflows we can define `Workflow` resources to implement Service Pipelines on steroids. Argo Workflows allow us to define DAGs (direct acyclic graphs) Workflows enabling us to define dependencies in a declarative for each task on the workflow (figure 2.9).

A very simple Workflow example defining a DAG is shown:

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: hello-world-  # Name of this Workflow
spec:
 - name: diamond
    dag:
      tasks:
      - name: A
        template: echo
      - name: B
        dependencies: [A]
        template: echo
      - name: C
        dependencies: [A]
        template: echo
      - name: D
        dependencies: [B, C]
        template: echo
```

Name of the task.

The task can depend on an array of previous tasks.

We can reference which template will be used for this particular task, promoting reuse of these tasks definitions.

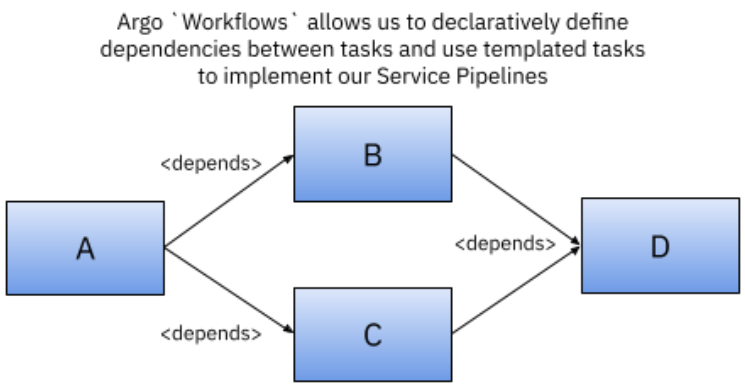

Figure 2.9 **Argo Workflow using DAG task definitions.**

As we seen in Tekton, Argo Workflows is also equipped with Templates that promote the reutilization of the tasks that we define for our workflows. In the previous example, we can see that each task inside the workflow is referencing the same template, which makes the Workflow more readable, but it also opens the door to create libraries of these tasks that can be shared across teams and curated by platform teams.

It is beside the point to show a Service Pipeline implemented with Argo Workflows, because it will look similar to the one that we define with Tekton, and it will run in a very similar way. Workflows are like Tekton Pipelines executed in sequence or in parallel, and each of them creating containers for each task execution. Workflows run to completion and report status back to the workflow resource.

Argo Workflows also provides a dashboard where we can monitor our workflow executions and logs using a graphical interface instead of the command line tools.

The main reason why you should look at Argo Workflows is because Workflows were designed alongside Argo Events and the rest of the Argo projects modules such as ArgoCD and Argo Rollouts that we will dig into later in this report.

With Argo Events, you can consume events from popular event sources such as Git Repositories, Slack, Kafka, and a large number of supported sources that you can find here (https://argoproj.github.io/argo-events/concepts/event_source/) and trigger workflows when events are received.

Argo Events introduce the concepts of event sources, event bus, sensors, and triggers. These concepts allows you to wire Event sources, route events to different sensors, and define which needs to be informed about those events using triggers.

Argo Events are quite powerful and I recommend you check their official documentation for more details, specially their architecture at https://argoproj.github.io/argo-events/concepts/architecture/, which shows how these concepts work together and can be extended to implement a myriad of integration scenarios. One detail that I would like to mention here is their use of NATS as their event bus, which will be used to connect sources with sensors. NATS is a highly efficient way to connect distributed systems.

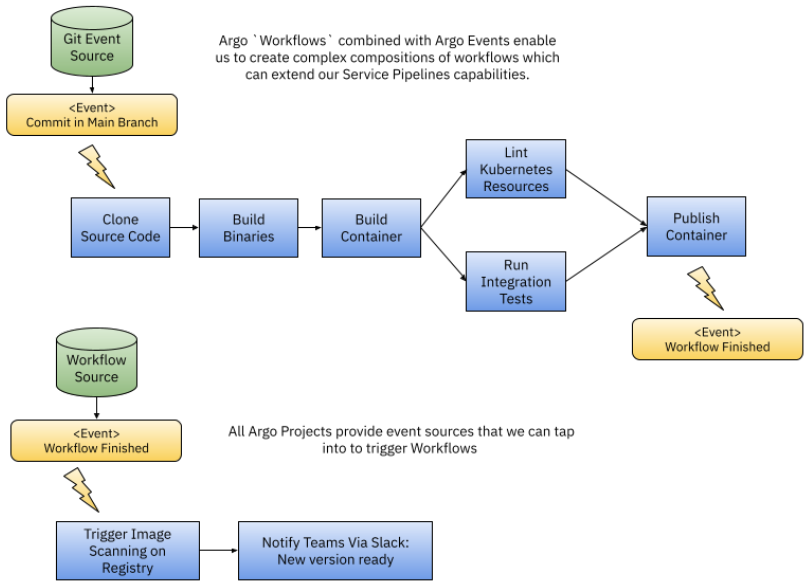

Figure 2.10 **Argo Workflow and Argo Events to expand our Service Pipeline capabilities.**

In figure 2.10 we are triggering a workflow from a Git Event source, imagine GitHub, or GitLab and emitting an event when the workflow is finished. This event can trigger another workflow, for example, to notify other teams about our new artifact version and maybe run some security checks after the artifacts were published. The nice thing about all Argo projects is that when you adopt one, you can smoothly start adding others because these projects were designed to work together.

While you can do something similar with Tekton Triggers, it will take you more time to put the pieces together, because you will need to use other projects to route and consume events from different sources.

Argo Workflows and Argo Events feels more like an end-to-end toolkit to implement Cloud Native automations, and if they are used alongside ArgoCD and Argo Rollouts we can cover the entire delivery cycle without reinventing the wheel or gluing tools that were created by different teams in isolation.

Tekton, in comparison, feels more low-level and very useful if you are planning to build higher-level abstractions on top and combining it with other tools in the CNCF ecosystem.

2.3 Environment Pipelines

Environment Pipelines are in charge of configuring and maintaining 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 a production environment which is where the real users interact with our applications. These (staging, QA, and production) are just examples, but there shouldn't be any hard limit on how many environments we can have.

In recent years, there has been a big surge in the industry to enable developers to request new environments in a self-service manner to make sure that they have what they need to find issues earlier in the development process. If these environments are closer to how the production environment (where the real users interact with our applications) operate, developers have a richer context to understand how their applications behave, to avoid what is commonly known as "but it works on my laptop" symptom.

At the end of the day, these environments are computing resources that someone needs to provision, configure, and pay for maybe in a cloud provider or on-premise, which means that optimizing and automating the way of creating and configuring these resources is vital for the entire development to production cycle.

Once the development is done, and Service Pipelines have produced new artifacts, Environment pipelines are in charge of promoting these artifacts to our different environments until they reach to our end users. Before jumping into how Environment Pipelines work, we need to cover a bit of background first.

2.3.1 How does this used to work and what has changed lately?

Traditionally, creating new environments was hard and costly. For these two reasons, creating new environments on demand wasn't a thing and the differences between

the environment where a developer used to create an application and where the application ran for end users was completely different. These differences, not only in computing power, caused a huge stress on operations teams responsible for running these applications, because they needed to fine tune the applications configurations depending on the environment capabilities.

Most of the time, deploying a new application or a new version of an application required you to shut down the server, run some scripts, copy some binaries, and then restart the server again with the new version running. After the server started again, the application could fail to start, hence more configuration tunning might be needed. Most of these configurations were done manually in the server itself, making it really difficult to remember and keep track of what was changed and why.

As part of automating these processes, tools like Jenkins (a Pipeline Engine) and/or scripts were used to simplify the process of deploying new binaries. So instead of manually stopping servers and copying binaries, an operator could run a Jenkins job defining which versions of the artifacts they wanted to deploy, and Jenkins would 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, for example, having fixed environments that were completely different from where the software was being developed and tested. To reduce even further the difference between different environments, we needed to first have a specification of how the environment is created and configured down to the version of the operating system and the software that was installed into the machines or virtual machines that conform the environment. Virtual machines helped a lot with this task, because we could easily create two or more virtual machines that were configured exactly in the same way.

When using virtual machines, we can even give our developers virtual machines for them to work on. But now we have a new problem, because 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 manually start these VMs in each server; hence, we will need a hypervisor to monitor and run VMs is a cluster of physical computers.

Using tools Jenkins and virtual machines (with hypervisors) was a huge improvement, because we implemented some automation, operators didn't need to manually access to servers or VMs to change configurations, and our environments were created using a configuration that was predefined in a fixed virtual machine configuration (figure 2.11).

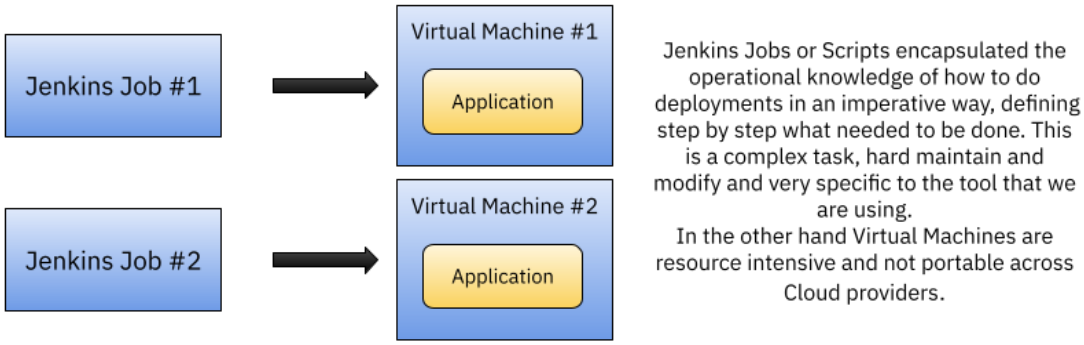

Figure 2.11 **Jenkins and virtual machines.**

While this approach is still a common approach 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 can be a great disadvantage, because 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 that you start a virtual machine, you start a complete instance of an operating system. Running the operating system processes is not adding any business value, and the larger the cluster the bigger the operating system overhead is. Running VMs in developers environment might not be possible depending on the VMs requirements.
- Most of the environments 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 different way to create virtual machines, pushing us into a vendor lock-in situation, where if we created VMs for Amazon Web Services, we cannot run these VMs into the Google Cloud Platform or Microsoft Azure.

How are teams approaching this with modern tooling? That is an easy question, because we now have Kubernetes and Containers that aimed to solve the overhead caused by VMs and the cloud provider portability by relying on containers and the widely adopted Kubernetes APIs. Kubernetes also provide us with the base building blocks to make sure that 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 on its own doesn't solve the process of configuring the clusters themselves, applying 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?

2.3.2 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 have in mind 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, because 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:* 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 build resilience in our environments and the entire delivery process, because 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 that we make. By relying on Git, we can easily rollback changes if these changes don't work as expected. GitOps not only covers the storying 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, because configuration management tools have existed for a long time. With the rise of cloud providers tools for managing Infrastructure as Code becoming popular, tools such as Chef, Ansible, Terraform, and Pulumi are loved by operation teams, because 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 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 express in the repository is what we have in our live environments.

By using GitOps we can reconfigure any Kubernetes Cluster to have the same configuration that is stored in our Git repository by running an Environment Pipeline (figure 2.12).

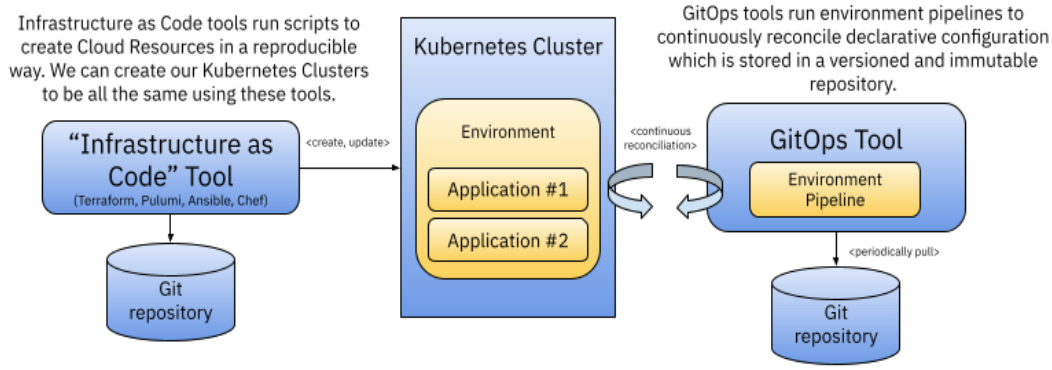

Figure 2.12 Infrastructure as Code, GitOps, and Environment Pipelines working together.

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 rollback both our infrastructural changes or our application changes as needed. It is also important to understand that because we are working with the Kubernetes APIs, our environments definitions are now expressed in a declarative way, supporting changes in the context where these configurations are applied and letting Kubernetes to deal on how to achieve the desired state expressed by these configurations.

Figure 2.13 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.

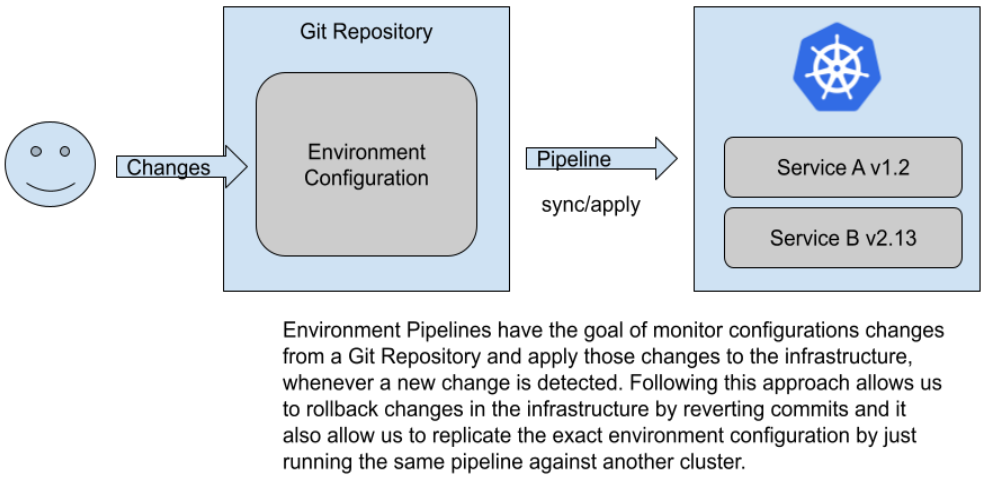

Figure 2.13 Defining the state of the cluster using the configuration in Git (GitOps).

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 that 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.

2.3.3 Steps involved with an Environment Pipeline

An environment pipeline will 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 pull the repository and check 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 process.
- *Clone the source code from the repository which contains the desired state for our environment:* This step will clone the configurations that had changed to be able to apply

them to the cluster. 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.

- *Apply the desired state to a live environment:* 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 2.14).

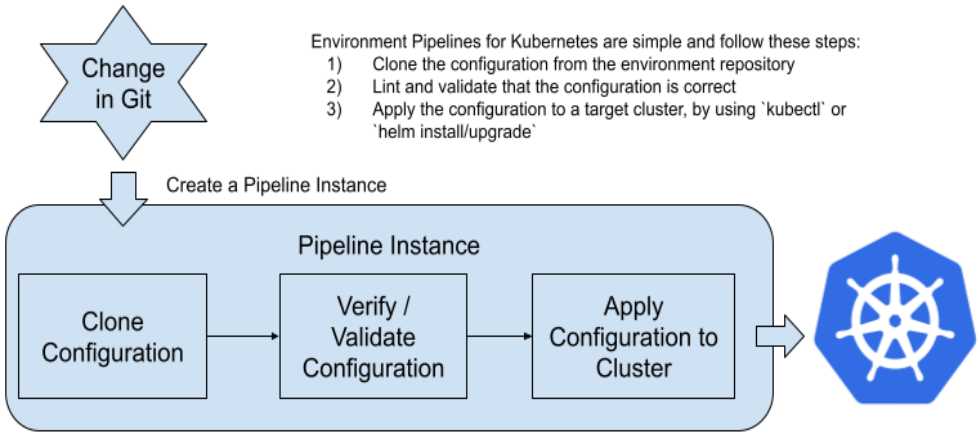

Figure 2.14 **Environment pipeline for a Kubernetes environment.**

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 services 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 references the fact that each environment will have a pipeline associated with it. Because having multiple environments is usually required (development, staging, production) for delivering applications, each will have a pipeline in charge of deploying and upgrading the components that are 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 and the pipeline will take care of reflecting the changes into the target cluster.

2.3.4 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 figure 2.15, the contents of the Environment Repository are just the definition of which services need to be present in the environment, the pipeline then can apply these Kubernetes manifests to the target namespace.

The following figure shows three 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.

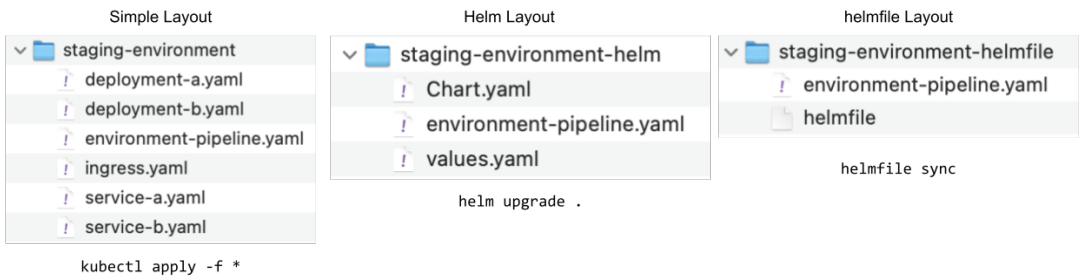

Figure 2.15 Three different approaches for defining environments' configurations.

The first option (Simple layout) is just to store all the Kubernetes YAML files in 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 2.16, it is common to have a separate Git repository for each environment, because 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, because 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.

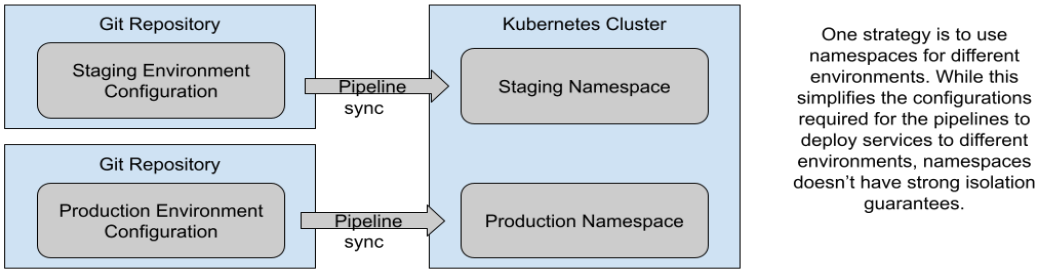

Figure 2.16 One environment per Kubernetes namespace.

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 stricter 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 2.17 shows how you can use the namespace approach for development environments that will be created by different teams and then have separate 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.

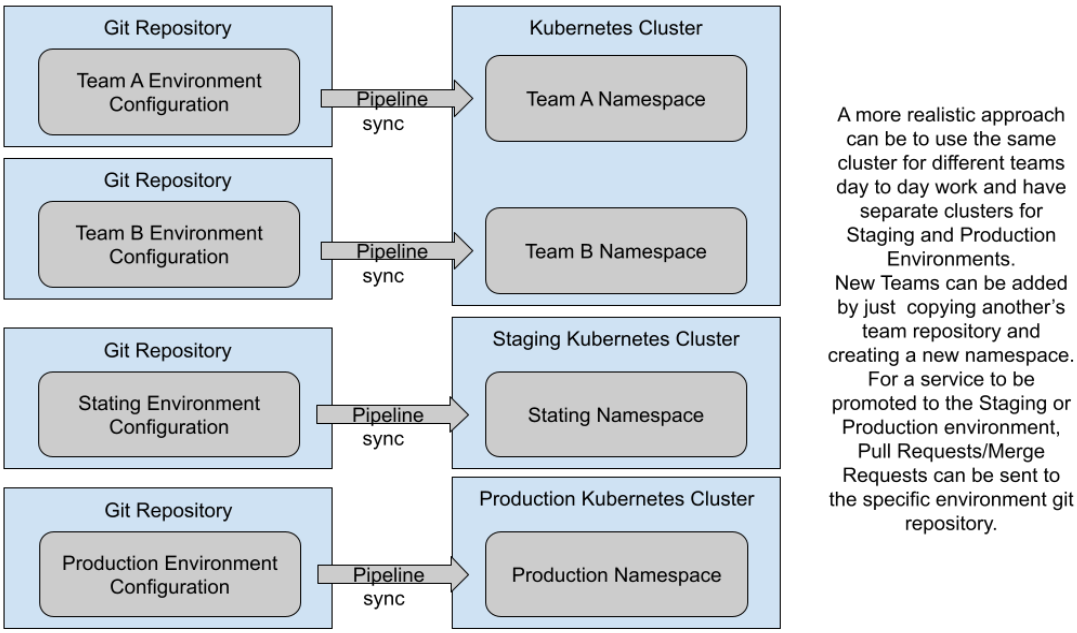

Figure 2.17 One environment per Kubernetes Cluster.

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.

2.4 Environment Pipelines in Action

You can definitely go ahead and implement an Environment Pipeline as described in the previous section using Tekton. This has been done in projects like Jenkins X (https://jenkins-x.io), and I have done it too in this repository: https://github.com/salaboy/from-monolith-to-k8s/blob/main/tekton/resources/environment-pipeline.yaml. 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 are using, 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.

2.4.1 Argo CD

ArgoCD provides a very opinionated but flexible GitOps implementation. When using ArgoCD, we will delegate all the steps required to continuously deploy software into our environments. 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 truths.

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 2.18).

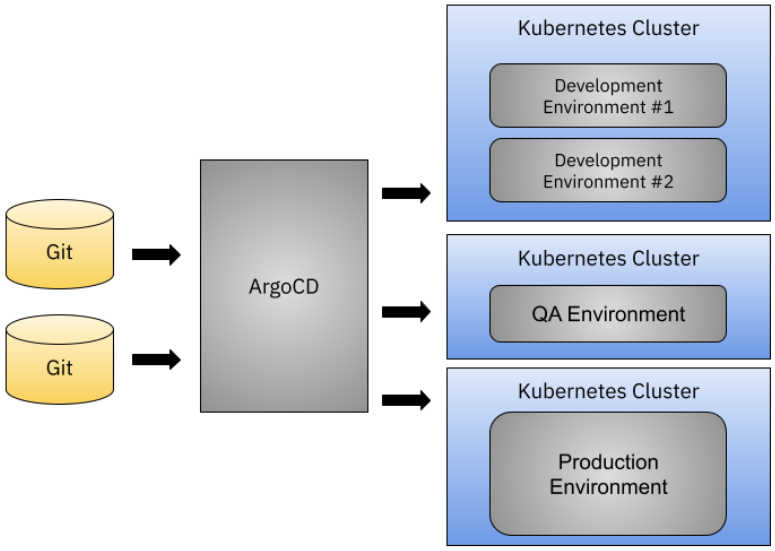

Figure 2.18 ArgoCD will sync environments configurations from Git to live clusters.

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 configure our staging environment using a GitOps approach. For that, we need a Git repository that serves as our source of truth.

For installing ArgoCD, I recommend you check their getting started guide that you can find here: https://argo-cd.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 is sometimes very handy. In the following sections, we will focus on the User Interface, but you can access pretty much the same functionality 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 aim 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. Because there are no restrictions on what you can include in a Helm Chart, we will use a Helm Chart to configure our Conference Platform application into this environment.

2.4.2 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 2.19):

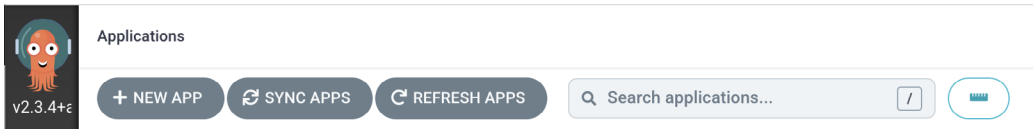

Figure 2.19 ArgoCD user interface: New application creation.

Go ahead and hit that button to see the Application creation form. Beside adding a name and selecting a project where our ArgoCD application will live (we will select the "default" project) and check the "Auto-Create Namespace" option (figure 2.20).

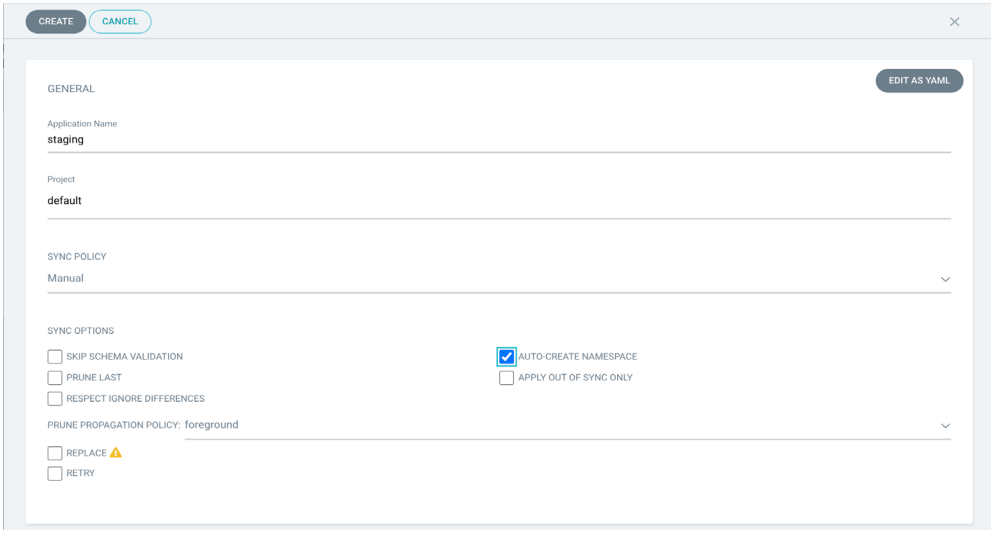

Figure 2.20 **New application parameters, manual sync, and auto-create namespace.**

By associating our environment to a new namespace in our cluster, we can use the Kubernetes RBAC mechanism to only allow administrators to modify the Kubernetes resources located in that namespace. Remember that by using ArgoCD, we want to make sure 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 that are defined in a Git repository. So where is that Git repository? That's exactly what we need to configure next in figure 2.21.

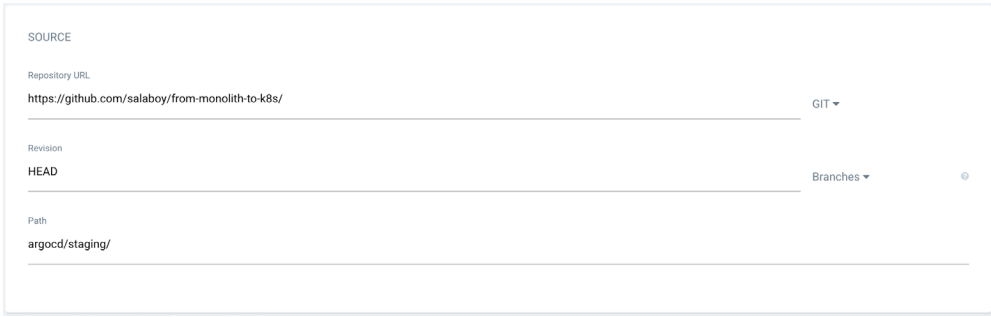

Figure 2.21 **ArgoCD application's configuration repository, revision, and path.**

As mentioned before, we will use 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 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 over time.

The next step is to define where this environment configuration is going to 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 use the same Kubernetes Cluster where we installed ArgoCD, and because the namespace will be automatically created, make sure to enter "staging" as the namespace so ArgoCD creates it for you (figure 2.22).

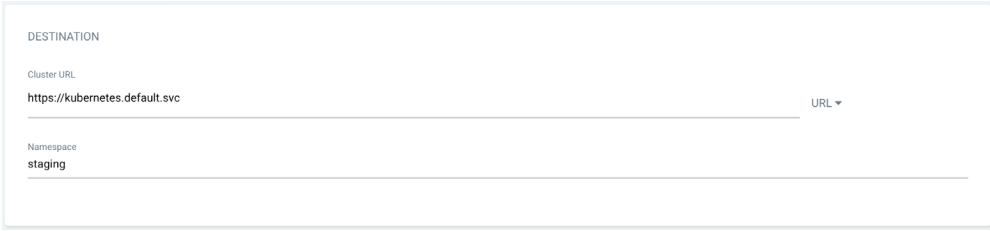

Figure 2.22 Configuration destination, for this example, is the cluster where ArgoCD is installed.

Finally, because it makes sense to reuse the same configuration for similar environments, ArgoCD enables us to configure different parameters that will be 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 2.23).

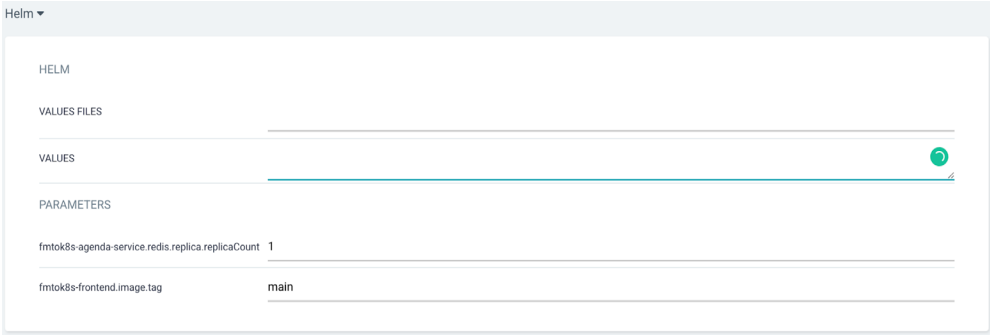

Figure 2.23 Helm configuration parameters for the staging environment.

As you can see in the previous image, ArgoCD also identified that there is a non-empty values.yaml file inside the repository path that we provided, and it is automatically parsing the parameters. We can add more parameters to the VALUES text box if we want to override any other chart (or sub-charts) configurations.

For example, if you are configuring the application to run in a cloud provider, you can add to the values section the following parameters:

```
fmtok8s-frontend:
 service:
   type: LoadBalancer
```

This configuration will automatically expose the "frontend" service to an external IP that you can access from outside your cluster.

After we provided 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 selected Manual Sync it will not automatically apply the configuration to the cluster (figure 2.24).

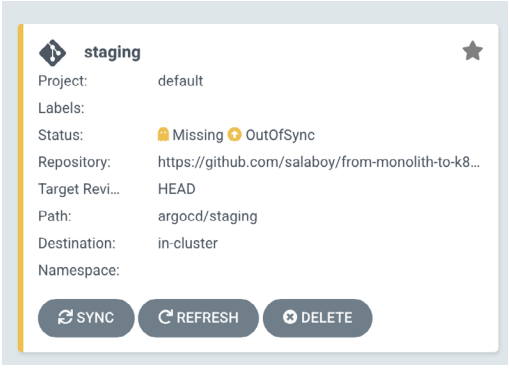

Figure 2.24 Application created but not synced.

If you click into the application, you will drill down to the application full view, which shows you the state of all the resources associated with the application (figure 2.25).

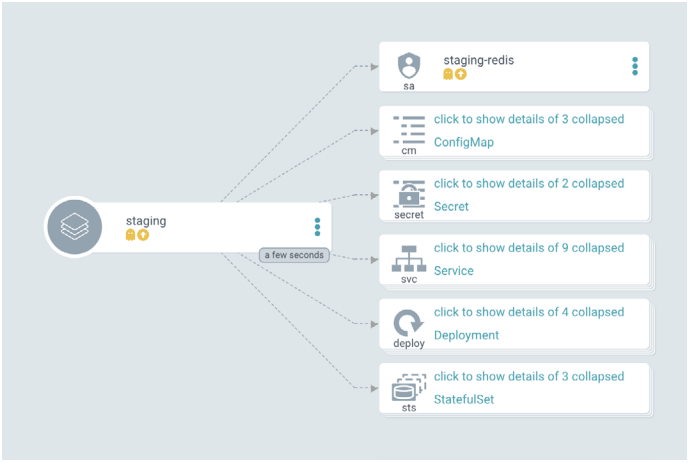

Figure 2.25 Application resources before sync.

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 2.26).

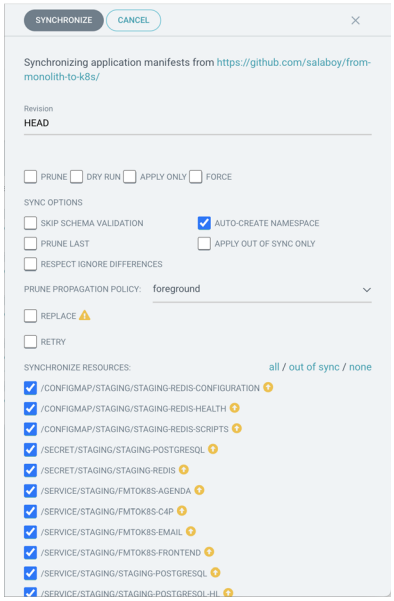

Figure 2.26 Application sync parameters.

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 being created and monitored by ArgoCD (figure 2.27).

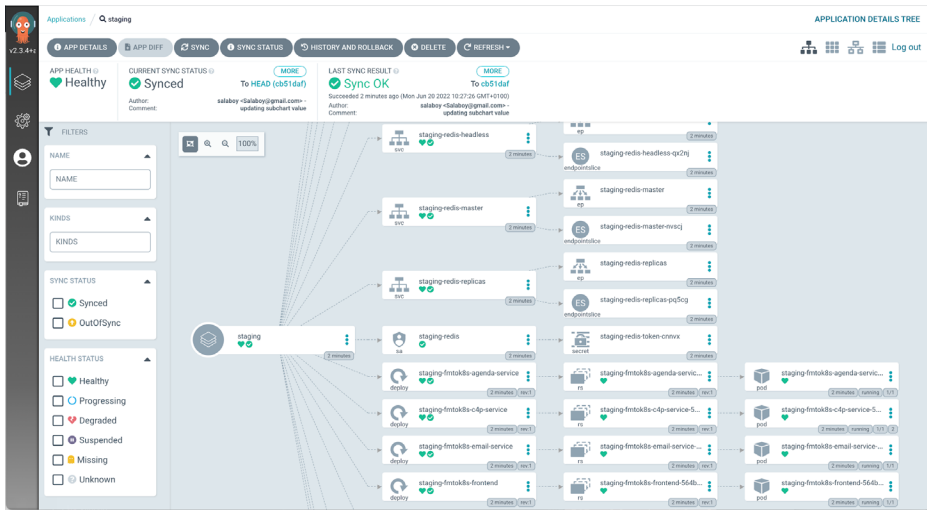

Figure 2.27 Our staging environment is Healthy, and all the services are up and running.

Depending on whether 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 installed ArgoCD into our Kubernetes Cluster. Using the provided ArgoCD UI, we created a new ArgoCD application for our staging environment.
- We created our staging environment configuration in a Git repository hosted in GitHub, which uses a Helm Chart definition to configure our Conference Platform services and their dependencies (Redis and PostgreSQL).
- We synced the configuration to a namespace (staging) in the same cluster where we installed ArgoCD.
- Most importantly, we 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.

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 (figure 2.28). The creation of these artifacts is the responsibility of the service pipeline, as we saw in previous sections.

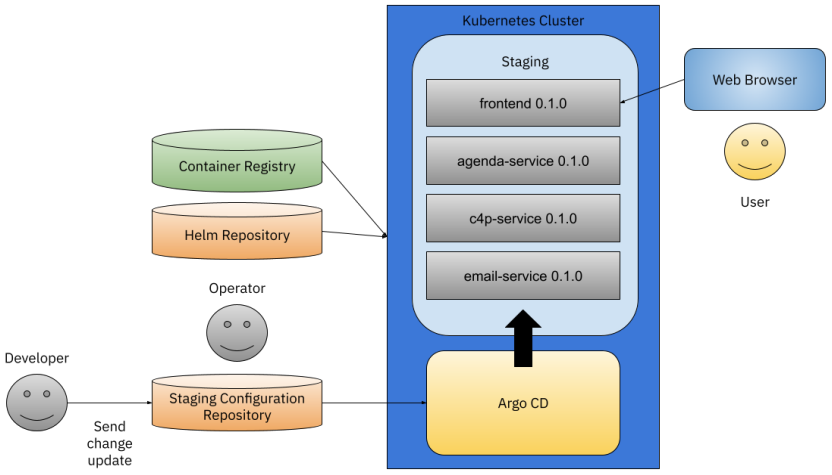

Figure 2.28 Components to set up the staging environment with ArgoCD.

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 that we used to configure our ArgoCD application. The changes in the environment configuration repository are usually going to be 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, because 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, because 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, because 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 try 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 in 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 2.29) for our pull requests to test changes before they get merged and artifacts are released. Jenkins X implements preview environments (https://jenkins-x.io/docs/build-test-preview/preview/). Unfortunately, Jenkins X doesn't use ArgoCD, but I can see this as a potential feature to build to ArgoCD out of the box.

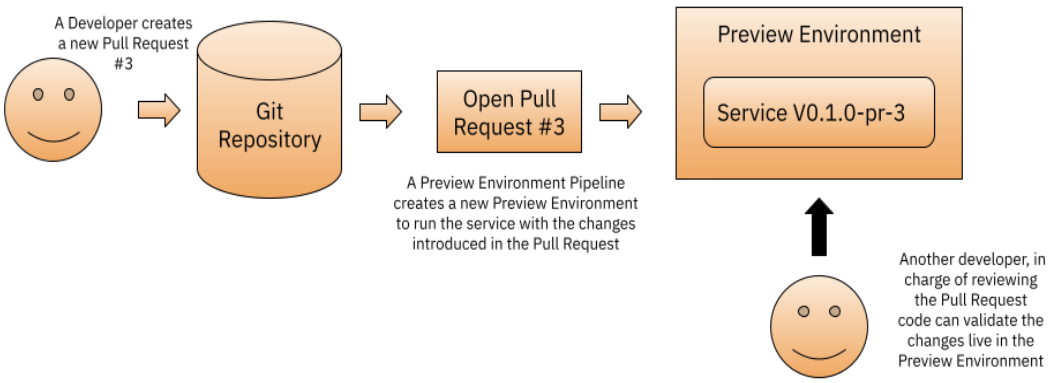

Figure 2.29 **Preview environments for faster iterations.**

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 it also allows unifying different templating engines for YAML. At the same time, it also allows ArgoCD to decorate the resulting resources with ArgoCD custom annotations.

Finally, 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.

2.5 Service + Environment Pipelines summary

Finally, let's look at how Service Pipelines and Environment Pipelines connect. The connection between these two pipelines happens via pull requests to repositories, because the pipelines will be triggered when changes are submitted and merged (figure 2.30).

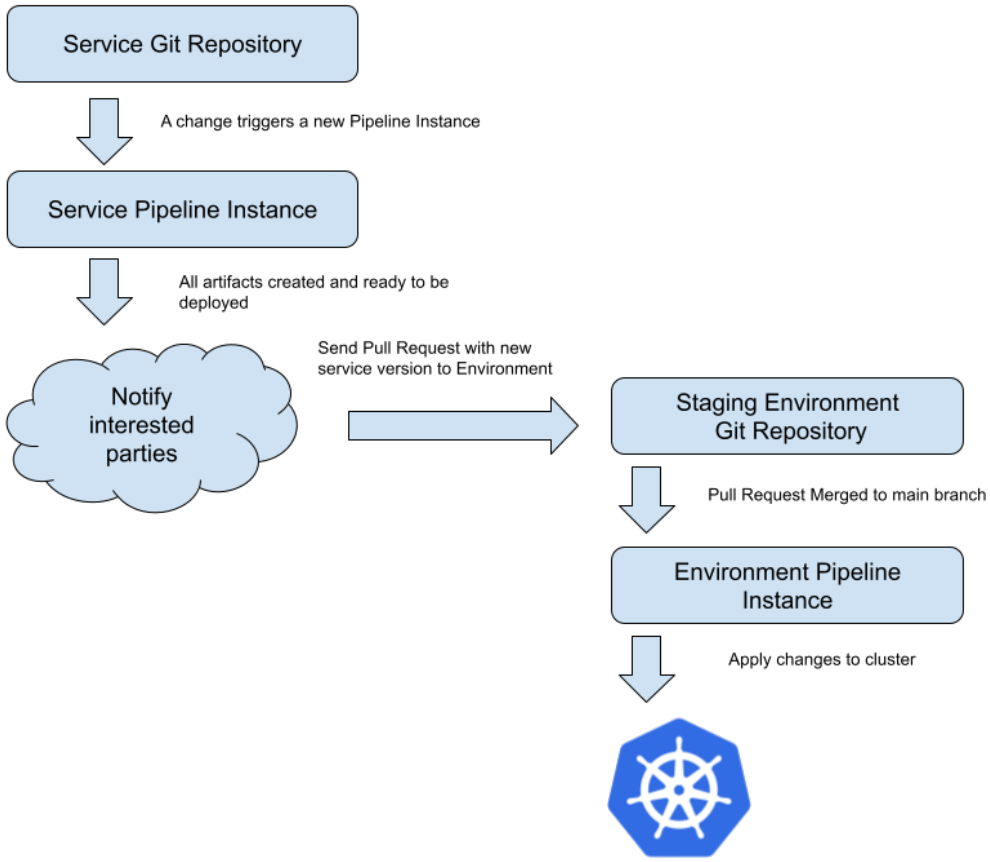

Figure 2.30 **A Service Pipeline can trigger an Environment Pipeline via a pull request.**

The Service Pipeline, after all the artifacts were released and published, can send an automatic pull request to the environment repository where the service needs to be updated or deployed triggering the Environment Pipeline.

This simple but effective mechanism allows automated or manual pull requests to be sent every time that we want to upgrade or deploy something into our environments. For certain environments, such as staging or development environments, you can automate the merging of the pull request containing new services versions enabling changes on the service repositories to be propagated automatically to these low-risk environments.

Now that our Service and Environment Pipelines are up and running, we can go all the way from a source code change into one of our service repositories to deploying the new artifacts generated by the Service Pipeline to one of our environments.

The more automated this process is the faster we can deliver new features, but we are far from being ready. Even if we can continuously deploy new artifacts to our defined environments most of the time, we need more flexibility and that is exactly where having different release strategies in our toolbox becomes really important.

The main reason to look into different release strategies is to recognize that sometimes having just a single version of our services running is not enough. Maybe we want to test different approaches before deciding how to go about building a new feature, or maybe we want to expose a feature to users in a certain region or maybe it makes a lot of sense for different versions of the same service to deal with different requests.

In the next couple of sections (2.6 and 2.7), we will review what we have available out of the box in Kubernetes and how tools like Argo Rollouts can help us to deal with more advanced scenarios.

2.6 Release strategies in Kubernetes

Kubernetes comes with built-in mechanisms ready to deploy and upgrade your services. Both deployment and StatefulSets resources orchestrate ReplicaSets to run a built-in rolling update mechanism when the resource's configurations change. Both deployments and StatefulSets keep track of the changes between one version and the next, allowing rollbacks to previously recorded versions.

Both deployments and StatefulSets are like cookie cutters; they contain the definition of how the container(s) for our service(s) needs to be configured and, based on the replicas specified, will create that amount of containers. To route traffic to these containers, we will need to create a service.

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 (from 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 2.31).

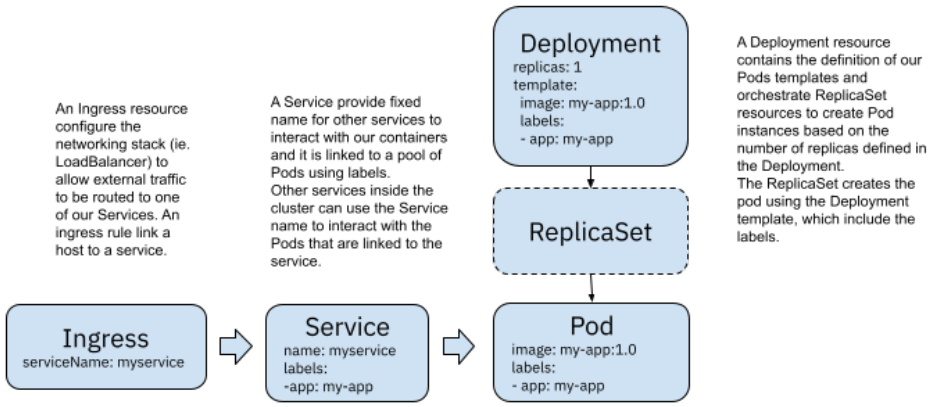

Figure 2.31 Kubernetes built-in resources for routing traffic.

Now, imagine what happens when you want to update the version of one of these external-facing services. We can agree that it is pretty common that these services are user

interfaces. And we can also agree that it will be pretty good if we can upgrade the service without having downtime. The good news is that Kubernetes was designed with zero-downtime upgrades in mind, and for that reason, both deployments and StatefulSets come equipped with a rolling update mechanism.

If you have multiple replicas of your application pods (figure 2.32), the Kubernetes Service resource acts as a load balancer. This allows other services inside your cluster to use the Service name without caring about which replica they are interacting with (or which IP address each replica has).

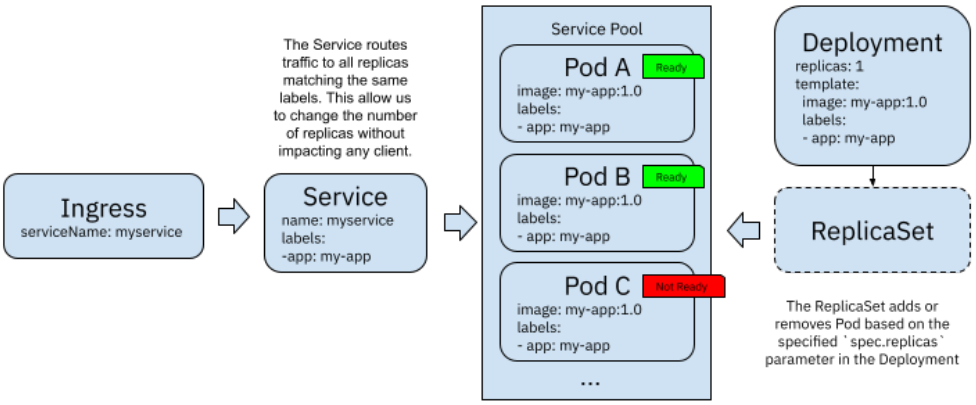

Figure 2.32 Kubernetes service acting as a load balancer.

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 2.32, 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 the "my-app:1.1" container image, we need to perform some very detailed orchestration of these containers. Suppose we want to make sure that 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 make sure that this new instance is up and running and ready to receive traffic before we remove the old version. If we have multiple replicas, we probably don't want to start all the new replicas simultaneously, because 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.`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/).

2.6.1 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 figure 2.33 shows how this mechanism works.

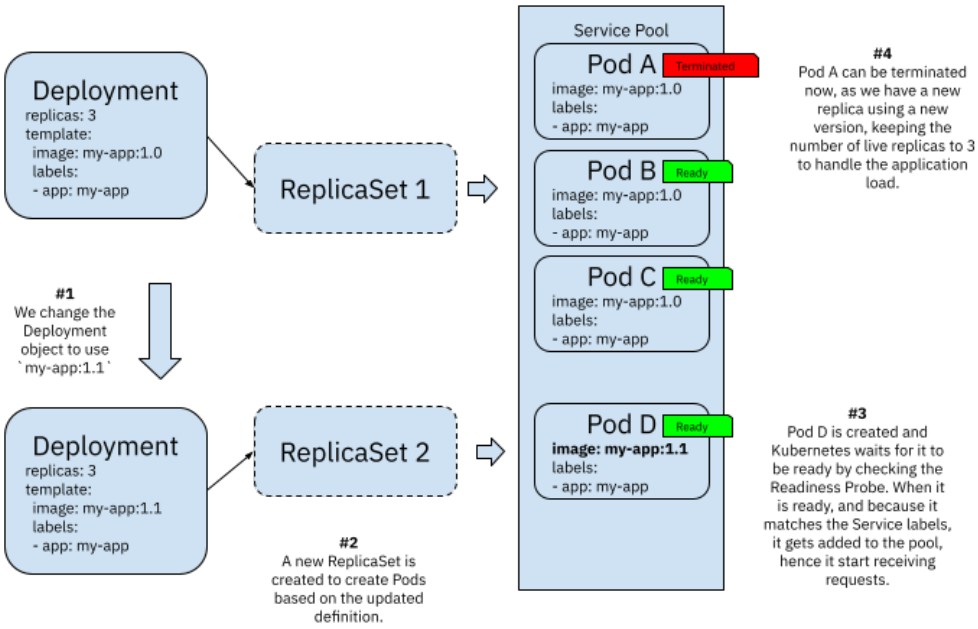

Figure 2.33 **Kubernetes deployments rolling updates.**

Whenever you update a Deployment resource, the rolling update mechanism kicks off by default. 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 then terminate a replica with the old configuration.

By doing this, the Deployment object guarantees three replicas are 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 similarly.

An extra benefit of using 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 rolled 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, because 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 about how this work here (https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/).

Check out the following commands to review the deployment revisions history and doing rollbacks to previous versions at (https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#checking-rollout-history-of-a-deployment):

```
> kubectl rollout history deployment/frontend
> kubectl rollout undo deployment/frontend --to-revision=2
```

If you haven't played around rolling updates with Kubernetes, I strongly recommend you create a simple example and try these mechanisms out. There are loads of examples online and interactive tutorials where you can see this in action.

2.6.2 Canary releases

Rolling updates kick in automatically, and they are performed as soon as possible if we are using deployments; what happens when we want to have more control over when and how we roll out new versions of our services?

Canary releases (https://martinfowler.com/bliki/CanaryRelease.html) are 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 do. Kubernetes will not check that the latest versions perform the same or better than the previous. Hence, we can be introducing issues to our applications. More control on how these updates are doing is needed.

If we start with a deployment configured to use a Docker image called "my-app:1.0", have two replicas, and we 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 (figure 2.34). In this scenario, 100% of the traffic will be routed to both replicas using the same Docker image (my-app:1.0).

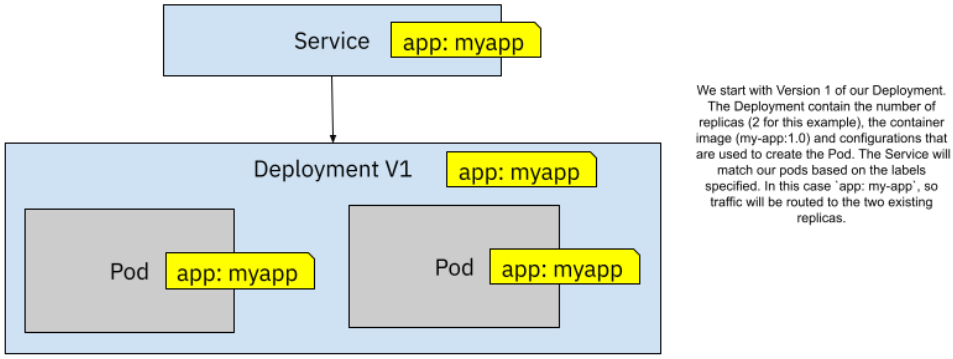

Figure 2.34 **Kubernetes service routes traffic to two replicas by matching labels.**

Now 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 2.35).

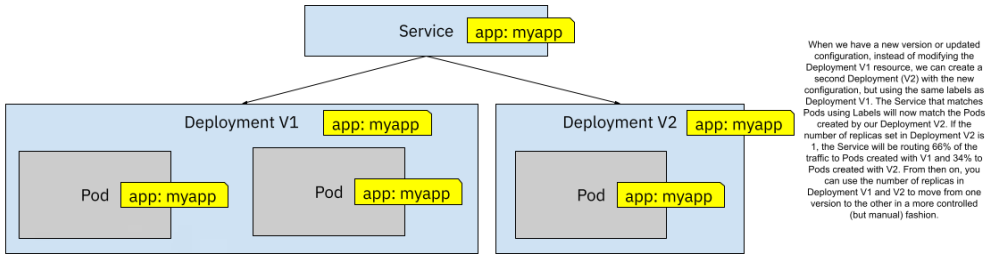

Figure 2.35 **One service and two Deployments sharing the same labels.**

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 2.35 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 slowly move towards V2. Notice that you don't have a fine-grained control over which requests go to which version—the service forwards traffic in a round-robin fashion to all the matched pods.

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 to make about rolling updates and Canary releases is that they depend on our services supporting traffic to be 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 the other. Teams making the changes need to be aware of this restriction when using rolling updates and Canary releases, because traffic will be forwarded to both versions simultaneously. For cases when two different versions cannot 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.

2.6.3 Blue/Green deployments

Whenever you face a situation where you just cannot 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 2.36.

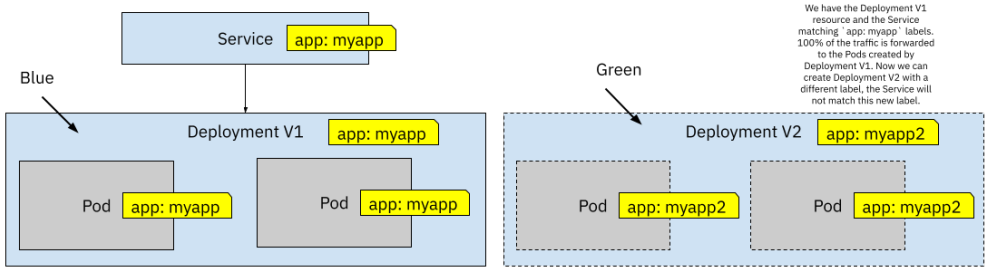

Figure 2.36 Two Deployments using different labels.

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 get ready for the new version of our service when we have a new version. This new deployment needs to have different labels for the pods that it will create, so the service doesn't match these pods just 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 2.37).

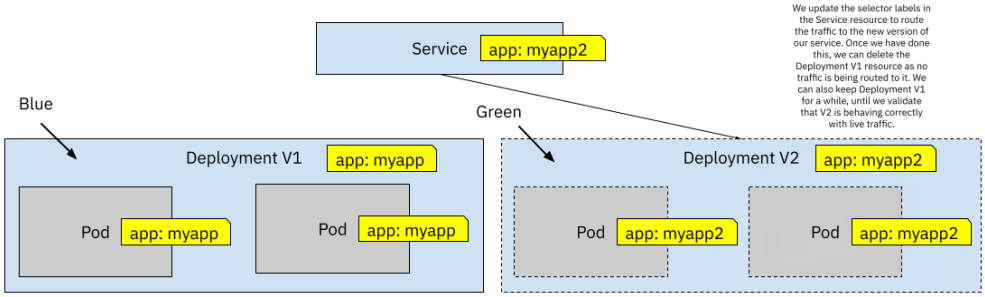

Figure 2.37 When the new version is ready, we switch the label from the service selector to match version V2 labels.

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 at the same time to switch traffic from one to the other. When we do Blue/Green deployments, it is recommended to have the same amount of replicas running for the new version. We need to make sure that when traffic is redirected to the new version, this version is ready to handle the same load as the old version.

The moment we switch the label selector in the service resource, 100% of the traffic is routed to the new version, 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 state from the Pods, migrate and make this state available for V2 Pods. In other words, if your application is holding state in-memory, you should write that state to persistent storage so V2 Pods can access it and resume whatever work was being 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 point in time, but what about scenarios when we want to actively test two or more versions with our users at the same time, to then decide which one performs better. Let's look at A/B testing next.

2.6.4 A/B testing

It is a quite common requirement to have two versions of your services running at the same time 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 into your app and gather feedback from users to decide which one works best.

To implement this kind of scenario in Kubernetes, you need to have 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 different 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 2.38).

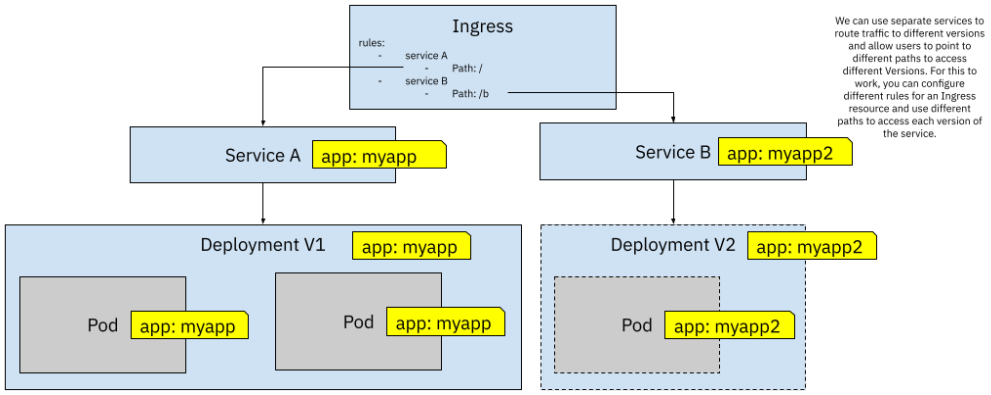

Figure 2.38 Using two Ingress rules for pointing to A and B versions.

If you have your application hosted under the www.example.com domain, the Ingress resource defined in figure 2.38 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 that I need to define and maintain just to achieve something that feels basic and needed for everyday operations. Let's quickly summarize the limitations and challenges that we found so far, so we can look at Argo Rollouts and how can it help us to implement these strategies in a more streamlined way.

2.6.5 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 saw 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 task and very error-prone. Even if you use a GitOps approach as shown with ArgoCD, 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 in charge of implementing the release strategies needs to understand deeply how Kubernetes behaves to achieve the desired configuration.
- No automated mechanisms are provided out-of-the-box to coordinate and implement the resources required by each release strategy.
- Error-prone, because 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 autoscaler (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, and that becomes a problem quite quickly if you are dealing with a large number of 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 in order 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 section, we will look at Argo Rollouts, a set of tools and mechanisms built on top of Kubernetes to simplify all the manual work described in the previous sections.

2.7 Reducing releases risk to improve delivery speed

We want to ensure that our operations teams have the right tools to implement the introduced strategies for running our applications. This section covers Argo Rollouts (https://argoproj.github.io/rollouts), a set of tools and mechanisms that facilitate a progressive delivery (https://redmonk.com/jgovernor/2018/08/06/towards-progressive-delivery/) approach. In the following sections, we will go over how we can implement canary releases and blue-green deployments using Argo Rollouts.

2.7.1 Introduction to Argo Rollouts

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 from manually interacting 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, because ArgoCD will do it for you.

Argo Rollouts as defined in 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 of manually modifying and keeping 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 that 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 locally.

You can follow a tutorial on how to install Argo Rollouts on a local Kubernetes KinD Cluster at 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.

2.7.2 Argo Rollouts and Canary rollouts

First, we'll begin by creating our first Rollout resource. With Argo Rollouts we will not define deployments because we will delegate 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 needs to be created).

For these examples, we will use only the Email Service from the Conference Platform application, and we will not use Helm, because 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 create files to test how Argo Rollouts behave. You can look at an Argo Rollout example using Helm at 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 at https://github.com/salaboy/from-monolith-to-k8s/blob/main/argorollouts/canary-release/rollout.yaml.

This Rollout resource manage 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-grained 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 push 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.argoproj.io
NAME                    DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
email-service-canary    3         3         3            3           11s
```

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 ReplicaSets, we can check using `kubectl get rs` that our Rollout has created a new ReplicaSet:

```
> kubectl get rs
NAME                               DESIRED   CURRENT   READY   AGE
email-service-canary-7f45f4d5c6    3         3         3       5m17s
```

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 in the main page (figure 2.39).

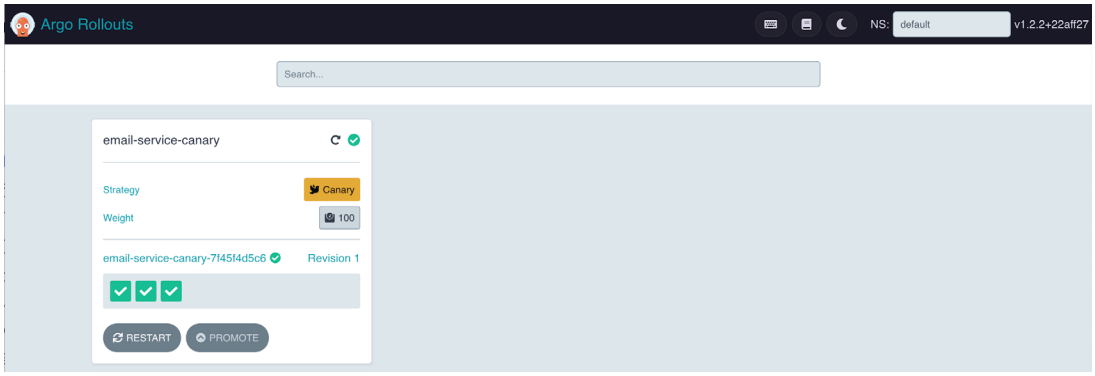

Figure 2.39 Argo Rollouts dashboard.

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 2.40).

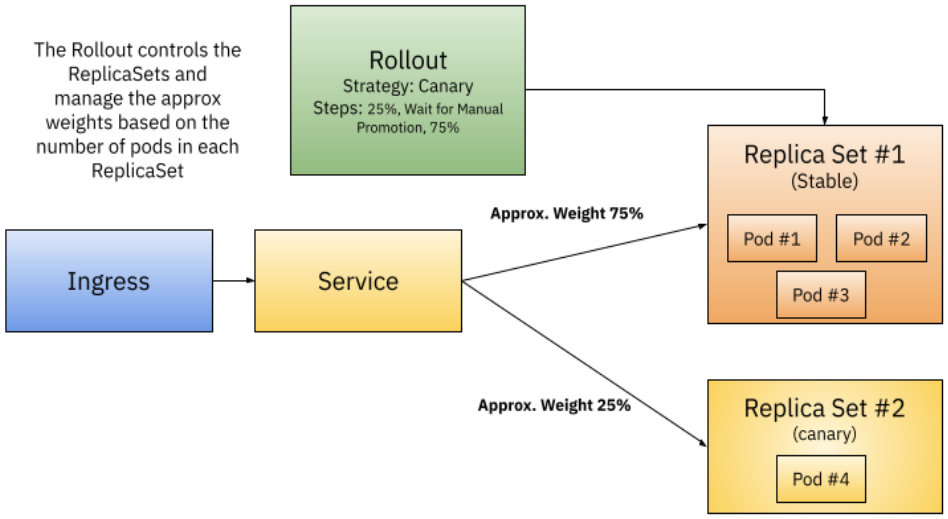

Figure 2.40 Argo Rollouts Canary release Kubernetes resources.

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",
    "podNamepsace": "default",
    "podNodeName": "dev-control-plane",
    "source": "https://github.com/salaboy/fmtok8s-email-service/releases/tag/
     v0.1.0",
    "version": "v0.1.0"
}
```

The request shows the output of the info endpoint of our email service application. Because we 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 changing 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-native (stable)
Replicas:
  Desired:       3
  Current:       3
  Updated:       3
  Ready:         3
  Available:     3
```

```
NAME                                          KIND        STATUS      AGE
     INFO
⟳ email-service-canary                        Rollout     ✔ Healthy   22h
└──# revision:1
   └──⧉ email-service-canary-7f45f4d5c6       ReplicaSet  ✔ Healthy   22h
     stable
     ├──□ email-service-canary-7f45f4d5c6-52j9b  Pod       ✔ Running   22h
     ready:1/1
     ├──□ email-service-canary-7f45f4d5c6-f8f6g  Pod       ✔ Running   22h
     ready:1/1
     └──□ email-service-canary-7f45f4d5c6-fhzzt  Pod       ✔ Running   22h
     ready:1/1
```

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        STATUS      AGE  INFO
⟳ email-service-canary                        Rollout     ⏸ Paused    22h
├──# revision:2
│  └──⧉ email-service-canary-7784fb987d       ReplicaSet  ✔ Healthy   18s  canary
│     └──□ email-service-canary-7784fb987d-q7ztt Pod      ✔ Running   18s  ready:1/1
```

You can see that revision 2 is labeled as the "canary" and the status of the rollout is "II 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 2.41).

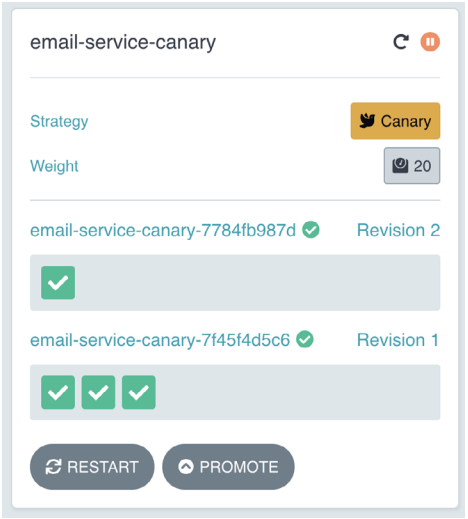

Figure 2.41 **A Canary release has been created with approximately 20% of the traffic routed to it.**

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 we continue 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",
    "podNamepsace": "default",
    "podNodeName": "dev-control-plane",
    "source": "https://github.com/salaboy/fmtok8s-email-service/releases/tag/
     v0.1.0",
    "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",
```

```
    "podNamepsace": "default",
    "podNodeName": "dev-control-plane",
    "source": "https://github.com/salaboy/fmtok8s-email-service/releases/tag/
     v0.2.0",
    "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 underlaying Replica Set objects and their replicas. You can check the replicasets by running `kubectl get rs`:

```
> kubectl get rs
NAME                                DESIRED   CURRENT   READY   AGE
email-service-canary-7784fb987d     1         1         1       33m
email-service-canary-7f45f4d5c6     3         3         3       23h
```

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 (because we are using `svc/email-service`), but `kubectl port-forward` resolves to a pod instance and connects to a single pod, allowing you only hit the Canary or a stable pod. For this reason, we 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 results, 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 2.7.3.

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        STATUS        AGE  INFO
⟳ email-service-canary                              Rollout     ✔ Healthy     22h
├──# revision:2
│  └──⧉ email-service-canary-7784fb987d             ReplicaSet  ✔ Healthy     13m  stable
│     ├──□ email-service-canary-7784fb987d-q7ztt    Pod         ✔ Running     13m  ready:1/1
│     ├──□ email-service-canary-7784fb987d-zmd7v    Pod         ✔ Running     81s  ready:1/1
│     └──□ email-service-canary-7784fb987d-hwwbk    Pod         ✔ Running     70s  ready:1/1
└──# revision:1
   └──⧉ email-service-canary-7f45f4d5c6             ReplicaSet  • ScaledDown  22h 22h
```

You can see this rollout progression live in the dashboard as well (figure 2.42).

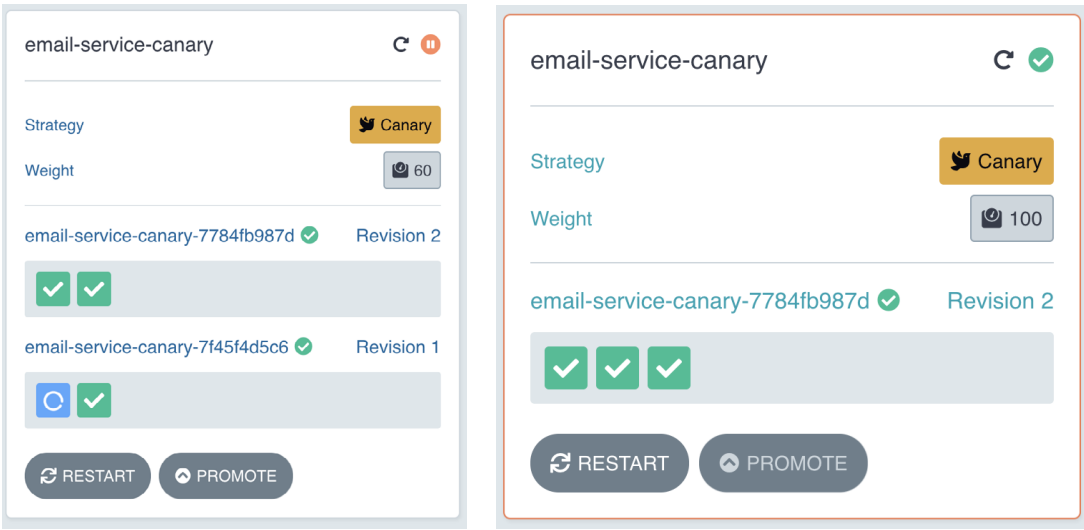

Figure 2.42 **The canary revision is promoted to be the stable version.**

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   CURRENT   READY   AGE
email-service-canary-7784fb987d     3         3         3       35m
email-service-canary-7f45f4d5c6     0         0         0       23h
```

We have successfully created, tested and promoted a canary release with Argo Rollouts!

Compared to what we saw in section 2.6.2 using two Deployment resources to implement the same pattern, with Argo Rollouts you have full control on 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.

2.7.3 Argo Rollouts and Blue/Green deployments

In section 2.6.3 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 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 look at what our Rollout with a `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 at 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-native (stable, active)
Replicas:
  Desired:       2
  Current:       2
  Updated:       2
  Ready:         2
  Available:     2

NAME                                              KIND        STATUS      AGE  INFO
⟳ email-service-bluegreen                         Rollout     ✔ Healthy   11m
└──# revision:1
   └──⧉ email-service-bluegreen-54b5fd4d7c        ReplicaSet  ✔ Healthy   10m  stable,active
      ├──□ email-service-bluegreen-54b5fd4d7c-gvvwt  Pod       ✔ Running   10m  ready:1/1
      └──□ email-service-bluegreen-54b5fd4d7c-r9dxs  Pod       ✔ Running   10m  ready:1/1
```

And in the dashboard, you should see something like figure 2.43.

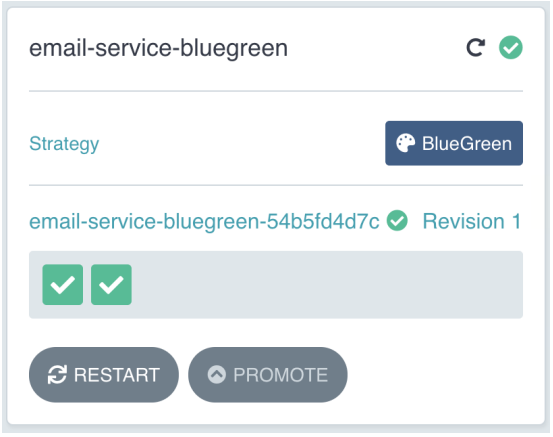

Figure 2.43 **Blue/Green deployment in the Argo Rollout dashboard.**

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",
    "podNamepsace": "default",
    "podNodeName": "dev-control-plane",
    "source": "https://github.com/salaboy/fmtok8s-email-service/releases/tag/
     v0.1.0",
    "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:            <none>
Annotations:       argo-rollouts.argoproj.io/managed-by-rollouts: email-service-bluegreen
Selector:          app=email-service,rollouts-pod-template-hash=64d9b549cf
```

```
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:            <none>
```

This selector is matching with the revision 2 Replica Set that is 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=64d9b549cf
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 a selector and labels, the Rollout with the `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 two pods each:

```
NAME                                          KIND        STATUS      AGE  INFO
⟳ email-service-bluegreen                     Rollout     ❚❚ Paused   22h
├──# revision:2
│  └──⧉ email-service-bluegreen-64d9b549cf    ReplicaSet  ✔ Healthy   22h  preview
│     ├──□ email-service-bluegreen-64d9b549cf-glkjv  Pod  ✔ Running   22h  ready:1/1,restarts:4
│     └──□ email-service-bluegreen-64d9b549cf-sv6v2  Pod  ✔ Running   22h  ready:1/1,restarts:4
└──# revision:1
   └──⧉ email-service-bluegreen-54b5fd4d7c    ReplicaSet  ✔ Healthy   22h  stable,active
      ├──□ email-service-bluegreen-54b5fd4d7c-gvvwt  Pod  ✔ Running   22h  ready:1/1,restarts:4
      └──□ email-service-bluegreen-54b5fd4d7c-r9dxs  Pod  ✔ Running   22h  ready:1/1,restarts:4
```

In the Argo Rollouts dashboard, you should see the same information (figure 2.44).

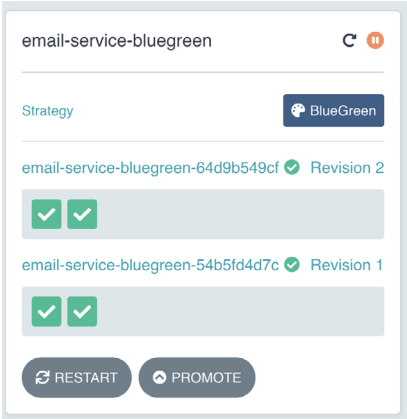

Figure 2.44 **Argo Rollout dashboard Blue and Green revisions are up.**

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",
    "podNamepsace": "default",
    "podNodeName": "dev-control-plane",
    "source": "https://github.com/salaboy/fmtok8s-email-service/releases/tag/
     v0.2.0",
    "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 2.45).

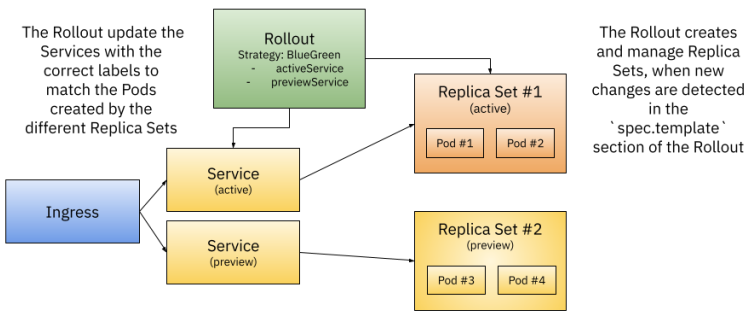

Figure 2.45 **Blue/Green deployment using Kubernetes resources.**

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-second 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 2.46).

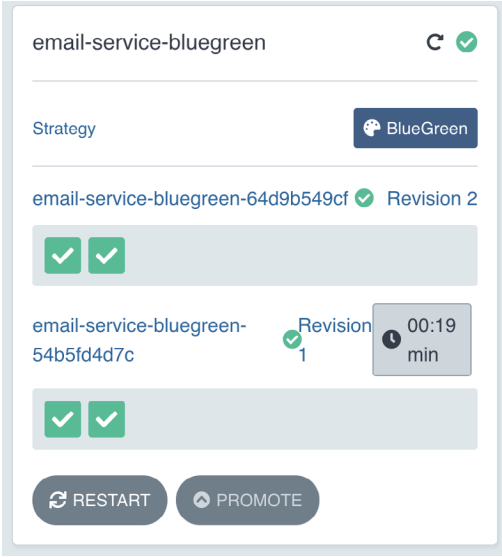

Figure 2.46 **Green service promotion using the Argo Rollouts dashboard (delay of 30 seconds).**

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.

2.7.4 Argo Rollouts analysis for progressive delivery

So far, we have managed to have more control for our different release strategies, but where Argo Rollouts really shine is by providing the AnalysisTemplate CRD that lets

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 to DataDog, New Relic among others, providing maximum flexibility to define these automated tests against the new revisions of our services (figure 2.47).

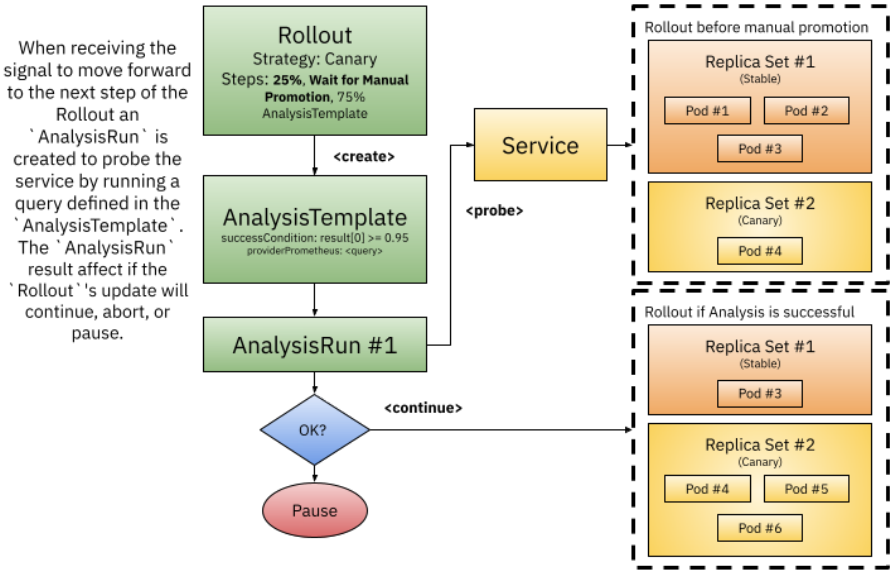

Figure 2.47 Argo Rollouts and Analysis working together to make sure that our new revisions are sound before shifting more traffic to them.

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 vector.
    # So it is common to access the index 0 of the returned array to obtain
     the value
    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 reference 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 setWeight: 40%
        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 Blue/Green strategy, we can trigger analysis runs pre- and post-promotion (figure 2.48).

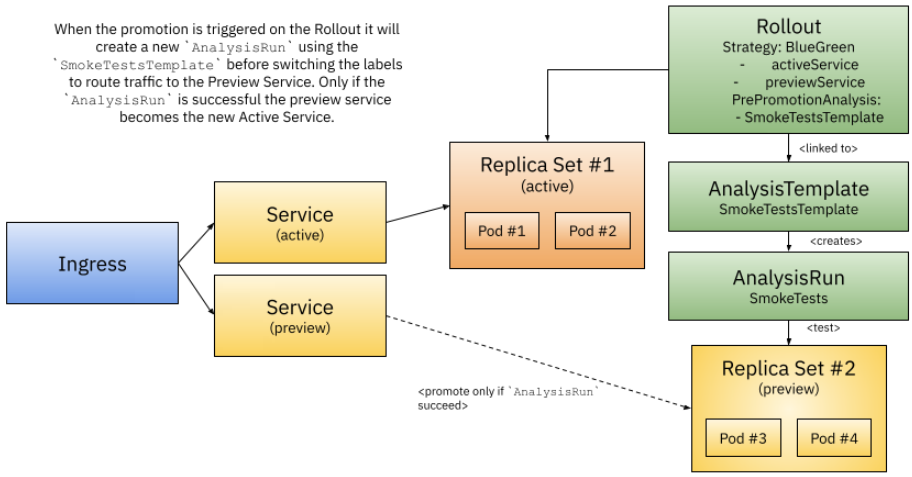

Figure 2.48 Argo Rollouts with blueGreen deployments, and PrePromotionAnalysis in action.

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 fails 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 check the Analysis section of the official documentation because 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/.

2.7.5 Argo Rollouts and traffic management

Finally, it is worth mentioning that so far Rollouts have used 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 on which traffic management tool we have available in our Kubernetes Cluster. Argo Rollouts today supports Istio, AWS ALB Ingress Controller, Ambassador Edge Stack, Nginx Ingress Controller, Service Mesh Interface (SMI), and 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 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-vsvc
      steps:
        - setCanaryScale:
            weight: 25
        - setMirrorRoute:
            name: mirror-route
            percentage: 35
            match:
              - method:
                  exact: GET
                path:
                  prefix: /
        - pause:
            duration: 10m
        - setMirrorRoute:
            nam": "mirror-ro"te" # removes mirror based traffic route
```

As you can see, this simple example already requires knowledge around Istio Virtual Services and a more advanced configuration that is out of scope for this section. I strongly recommend checking the *Istio in Action* book by Christian Posta and Rinor Maloku (https://www.manning.com/books/istio-in-action) if you are interested in learning about Istio (figure 2.49).

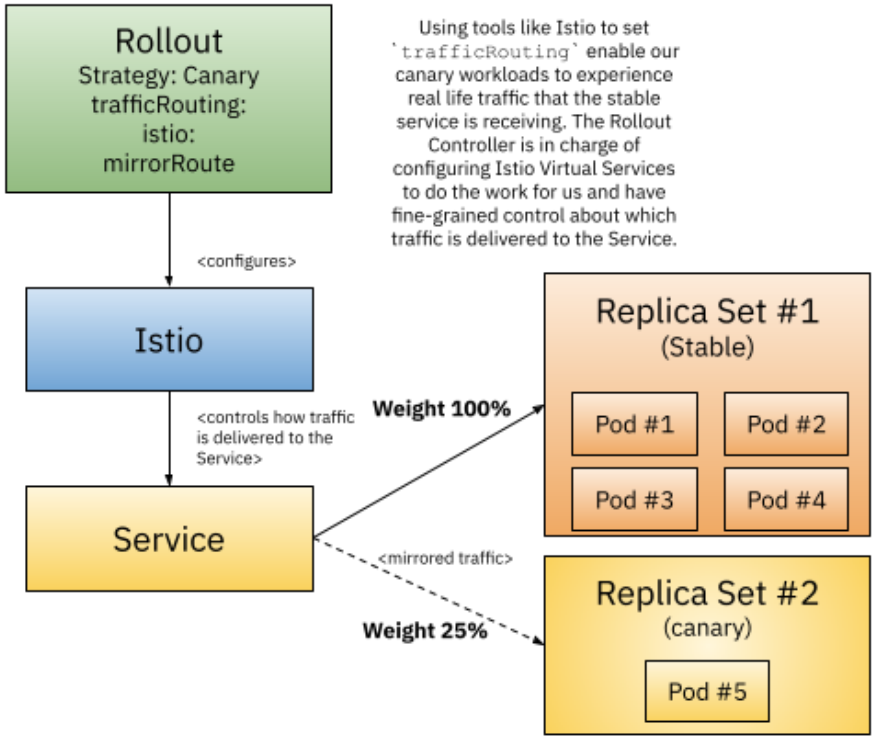

Figure 2.49 Traffic Mirroring to a Canary release using Istio.

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 following the examples there, because the rollouts needs to be configured differently depending on the Service Mesh that you have available.

In the last two sections (2.6 and 2.7) we have seen both what can be achieved with basic Kubernetes building blocks and how Argo Rollouts simplify the lives of teams releasing new versions of their applications to Kubernetes. 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. There are many software vendors who can assist you and provide recommendations too.

2.8 Summary

We started this report by looking at what Cloud-Native means in the context of Kubernetes and the reason behind continuously delivering new value to our users by leveraging the tools that have been described here. We used a Walking Skeleton to demonstrate the main challenges behind creating, building, packaging, and deploying these distributed applications.

Kubernetes is a great platform, but without having the right tools in place, it can be daunting and challenging for teams to be productive. I hope by the end of this report, you find yourself with enough practical experience to set up these tools for your teams and research the CNCF Landscape to look for more specific projects that can help you in your journey to the Cloud.

You can always refer to the step-by-step tutorials that you can find in this repository at https://github.com/salaboy/from-monolith-to-k8s/. I will do my best to keep these tutorials updated, and I am sure that I will keep adding more tools related to platform building and Continuous Delivery. As with all the work that I do, these repositories are under the Apache License V2, and you are more than welcome to contribute, provide feedback, and spread the word about them as much as you want.

In this report, we covered tools like Helm, KinD, Argo CD, Argo Rollouts, Tekton, and Argo Workflows, but in real life you will need to make your own decisions based on the problems that you are trying to solve and the tools that you are already using. One very important takeaway from this report is to make sure that whatever tools you choose, you understand the problems that these tools were designed to solve and how they match your technology stack decisions. The Kubernetes ecosystem is growing at a really fast pace, hence getting involved with open-source projects can provide you (and your teams) with an extra edge to understand how tools are being designed and where the entire ecosystem is going.

See you all out there in these open-source communities, GitHub issues, and pull requests! Remember that if you want to get in touch, you can always drop me a direct message in Twitter @Salaboy or a comment in my blog at https://salaboy.com.