COMPLIMENTS OF codefresh

Software Deployment with Kubernetes

by Mauricio Salatino

MANNING REPORT

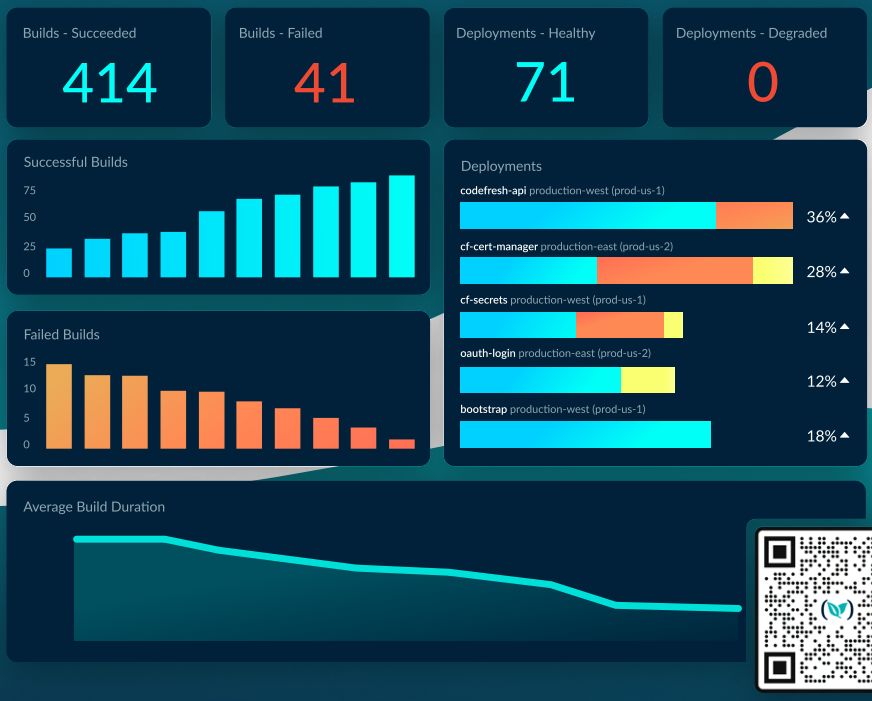
codefresh

The World's Most Modern CI/CD Platform for Cloud-Native Apps

What do we do?

Codefresh is a next-generation enterprise CI/CD platform for cloud-native applications. We help you automate your microservice builds and deployments with Kubernetes native workflows. When you are ready for GitOps, we have you covered with advanced deployments such as Canary and Blue/Green. Thousands of DevOps teams depend on Codefresh to build and deploy their software in a safe and scalable manner. Codefresh integrates with best-of-breed tools to support your software delivery for even the most complex scenarios.

- POWERFUL BUILDS
- INSTANT DEVELOPER FEEDBACK
- RESILIENT DEPLOYMENTS



Sign-Up

TRUSTED BY

GUARDANT · SKILLSHARE · GoodRx · monday.com · HOVER

Learn more at Codefresh.io

Software Deployment with Kubernetes

Mauricio Salatino

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

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

Cover designer: Marija Tudor

ISBN: 9781633436756
Printed in the United States of America

contents

foreword

Excellent software delivery practices are the key innovation to making effective software engineering teams, and making effective software engineering teams is the most important thing to get right for any business competing on software. If you've been automating software delivery the same way for a long time, this book is for you. If you're new to software delivery, this book is for you. Mauricio has put together a short, straightforward guide to adopting modern software delivery practices, along with examples and a boilerplate, to help you get started.

Follow this guide to improve your software delivery and ignore its advice at your peril.

—Dan Garfield
Co-creator of The GitOps Working Group
Argo Maintainer
Chief Open Source Officer and Co-Founder of Codefresh

1 Environment pipelines: Deploying cloud-native applications

This chapter covers

- Deploying produced artifacts from service pipelines
- Using environment pipelines and GitOps to manage environments
- Combining Argo CD with Helm to deliver software more efficiently

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

This chapter is divided into three main sections:

- Environment pipelines
  - What is an environment pipeline?
  - Environment pipelines in action using Argo CD
- Service + environment pipelines working together

1.1 Environment pipelines

We can build as many services as we want and produce new versions, but if these versions cannot flow freely across different environments to be tested to be used by our customers, our organization will struggle to have a smooth end-to-end software delivery practice. Environment pipelines are in charge of configuring and keeping our environments up-to-date.

It is quite common for companies to have different environments for different purposes, for example, a staging environment where developers can deploy their latest versions of the services or a quality assurance (QA) environment where manual testing happens, and one or more production environments which are where the real users interact with our applications. These (staging, QA, and production) are just examples (figure 1.1). There shouldn't be any hard limit on how many environments we can have.

Figure 1.1 Released service moving throughout different environments.

Each environment (development, staging, quality assurance, and production) will have one environment pipeline. These pipelines will be responsible for keeping the environment configuration in sync with the hardware running the live version of the environment. These environment pipelines use as the source of truth a repository that contains the environment configurations, including which services and which version of each service needs to be deployed (figure 1.2).

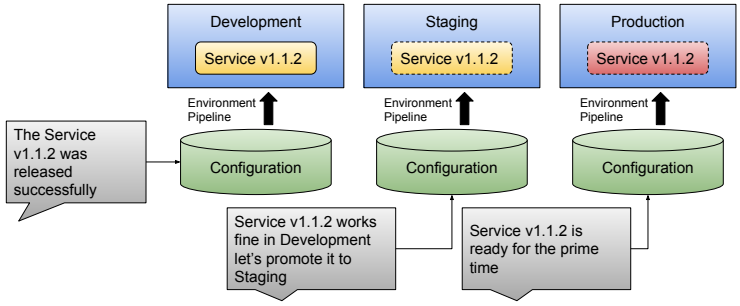


Figure 1.2 Promoting services to different environments means updating environment configurations.

If you are using this approach, each environment will have its configuration repository. Promoting a newly released version means changing the environment configuration repository to add a new service or updating the configuration to point to the newly released version.

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

But where do environment pipelines come from? And why have you not heard of them before? Before jumping into the details about how an environment pipeline would look, we need to get a bit of background on why this matters in the first place.

1.1.1 How did this work in the past, and what has changed lately?

Traditionally, creating new environments was hard and costly. Creating new environments on demand wasn't a thing for these two reasons. First, the differences between the environment that a developer used to create an application and where the application ran for end users were completely different. These differences, not only in computing power, caused huge stress on operations teams responsible for running these applications. Depending on the environment's capabilities, they needed to fine-tune the application's configurations (that they didn't design). Second, tools for automating the provisioning and configuration of complex setups have become mainstream. With the help of containers and Kubernetes, there has been a standardization around how these tools are designed and how they work across cloud providers. These tools have reached a point where developers are allowed to codify infrastructure using their programming language of choice or by relying on the Kubernetes API to create these definitions.

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

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

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

While automating with tools like Jenkins was a big improvement compared with manually deploying new versions, there were still some problems, for example, having to fix environments that were completely different from where the software was being developed and tested. To reduce the difference between different environments even further, we need to have a specification of how the environment is created and configured

down to the operating system's version and the software installed into the machines or virtual machines. Virtual machines helped greatly with this task, because we can easily create two or more virtual machines that are similarly configured.

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

Using tools Jenkins and virtual machines (with hypervisors) was a huge improvement as we implemented some automation (figure 1.3), because operators didn't need to access servers or VMs to change configurations manually, and our environments were created using a configuration predefined in a fixed virtual machine configuration. Tools like Ansible (https://www.ansible.com/) and Puppet (https://www.puppet.com/) are built on top of these concepts.

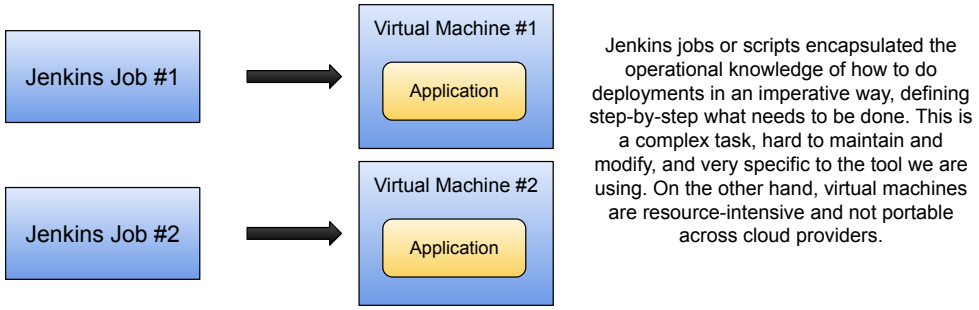


Figure 1.3 Jenkins and virtual machines.

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

- *Jenkins jobs and scripts are imperative by nature:* This means that they specify step-by-step what needs to be done. This has a great disadvantage: if something changes, let's say a server is no longer there or requires more data to authenticate against a service, the logic of the pipeline will fail, and it will need to be manually updated.
- *Virtual machines are heavy:* Every time you start a virtual machine, you start a complete instance of an operating system. Running the operating system processes does not add any business value; the larger the cluster, the bigger the operating system overhead. Depending on the VM's requirements, running VMs in developers' environments might not be possible.
- *Environments configurations are hidden and not versioned:* Most of the environment configurations and how the deployments are done are encoded inside tools like Jenkins, where complex pipelines tend to grow out of control, making the changes very risky and migration to newer tools and stacks very difficult.

- *Each cloud provider has a non-standard way of creating virtual machines.* Pushing us into a vendor lock-in situation. If we created VMs for Amazon Web Services, we could not run these VMs into the Google Cloud Platform or Microsoft Azure.

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

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

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

1.1.2 What is GitOps, and how does it relate to environment pipelines?

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

The term GitOps, defined by the CNCF's GitOps Working 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. OpenGitOpsdefines four core principles that we need to consider when we talk about GitOps:

1. Declarative: A system (https://github.com/open-gitops/documents/blob/v1.0.0/GLOSSARY.md#software-system)managed by GitOps must have its desired state expressed declaratively (https://github.com/open-gitops/documents/blob/v1.0.0/GLOSSARY.md#declarative-description). We have this pretty much covered if we use Kubernetes manifest, because we define what needs to be deployed and how that needs to be configured using declarative resources that Kubernetes will reconcile.
2. Versioned and immutable: The desired state is stored (https://github.com/open-gitops/documents/blob/v1.0.0/GLOSSARY.md#state-store) in a way that enforces immutability and 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 storing 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 (https://github.com/open-gitops/documents/blob/v1.0.0/GLOSSARY.md#continuous) observe the actual system state and attempt to apply (https://github.com/open-gitops/documents/blob/v1.0.0/GLOSSARY.md#reconciliation) 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 monitoring our environments from configuration drifts. If the reconciliation fails, GitOps tools will notify us about the problems and keep trying to apply the changes until the desired state is achieved.

By storing the configuration of our environments and applications in a Git repository, we can track and version the changes we make. By relying on Git, we can easily roll back changes if these changes don't work as expected. GitOps covers the configuration storage and how these configurations are applied to the computing resources where the applications 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. Instead, GitOps represents a refinement of these tried-and-tested approaches that can be applied to any software operation, not just Kubernetes. With the rise of cloud providers' tools for managing Infrastructure as Code became popular, tools like 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 that we can automate the creation of these clusters.

With GitOps, we manage configuration and rely on the Kubernetes APIs as the standard way to deploy 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 interact manually with the Kubernetes clusters to avoid configuration drifts and security problems. 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 ensure that the desired state expressed in the repository is what we have in our live environments.

We can reconfigure any Kubernetes Cluster to have the same configuration stored in our Git repository by running an environment pipeline (figure 1.4).

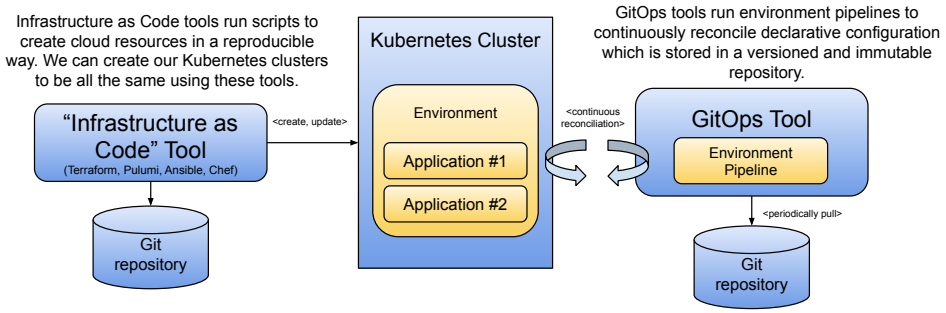
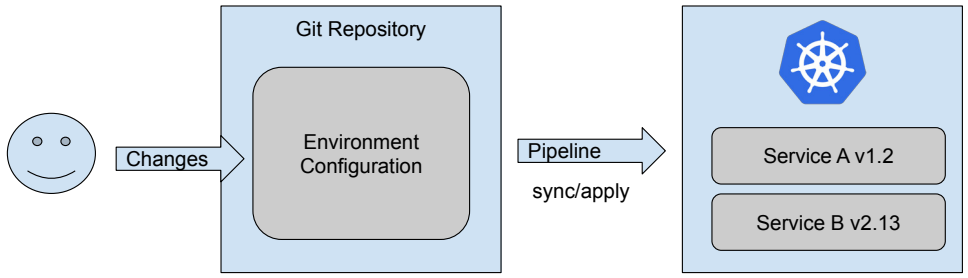


Figure 1.4 Infrastructure as Code, GitOps, and environment pipelines working together.

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

Figure 1.5 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 ensure that this configuration is in sync with the target environment.



Environment pipelines monitor configuration changes from a Git repository and apply those changes to the infrastructure whenever a new change is detected. Following this approach allows us to roll back changes in the infrastructure by reverting commits. It also allows us to replicate the exact environment configuration by running the same pipeline against another cluster.

Figure 1.5 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 so 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, an operator 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, an operator that monitors a Git repository and reacts to changes is all you need, but in practice, a set of steps are needed to ensure we have full control of what is deployed to our environments. Hence, thinking about GitOps as a pipeline helps us to understand that for some scenarios, we will need to add extra steps to these pipelines that are triggered every time an environment configuration is changed.

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

1.1.3 Steps involved in an environment pipeline

Environment pipelines usually include the following steps (figure 1.6):

- *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 sync. Remember, the GitOps principles state "pulled automatically", which means we can use webhooks, but we should not rely entirely on them for getting config change updates.
- *Clone the source code from the repository, which contains the desired state for our environment:* This step fetches the configuration from a remote Git repository that contains the environment configurations. Tools like Git fetch only the delta between the remote repository and what we have locally.
- *Apply the desired state to a live environment:* This usually includes doing a `kubectl apply -f` or a `helm install` command to install new versions of the artifacts. Notice that with both `kubectl` and `helm`, Kubernetes is smart enough to recognize where the changes are and only apply the differences. Once the pipeline has all the configurations locally accessible, it will use a set of credentials to apply these changes to a Kubernetes cluster. Notice that we can fine-tune the access rights that the pipelines have to the cluster to make sure that they are not exploited from a security point of view. This also allows you to remove access from individual team members to the clusters where the services are deployed.

- *Verify that the changes are applied and that the state is matching what is described inside the Git repository (deal with configuration drift):* Once the changes are applied to the live cluster, checking that the new versions of services are up and running is needed to identify if we need to revert to a previous version. If we need to revert changes, it is quite simple, because all the history is stored in Git. Applying the previous version is just looking at the previous commit in the repository.

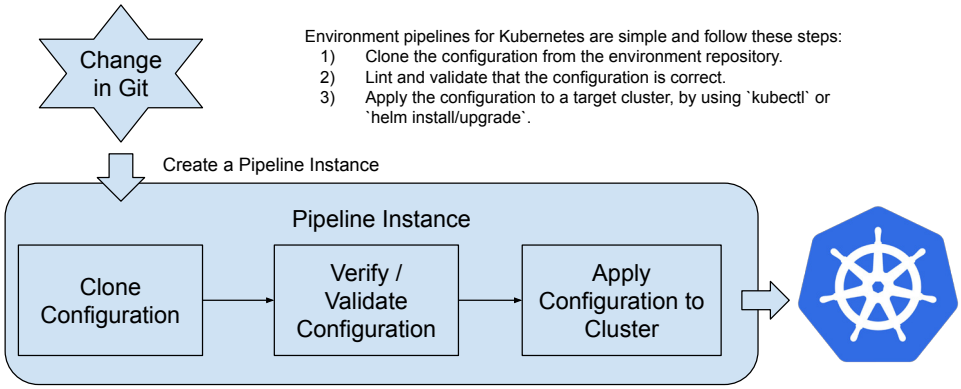


Figure 1.6 **Environment pipeline for a Kubernetes environment.**

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

The term environment pipeline refers to the fact that each environment will have a pipeline associated with it. Because multiple environments are usually required (development, staging, production) for delivering applications, each will have a pipeline in charge of deploying and upgrading the components running in them. By using this approach, promoting services between different environments is achieved by sending pull requests/change requests to the environment's repository. The pipeline will reflect the changes in the target cluster.

1.1.4 Environment pipeline requirements and different approaches

What are the contents of these environments' 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 1.7, the contents of the environment repository are just the

definition of which services need to be present in the environment. The pipeline can then 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.

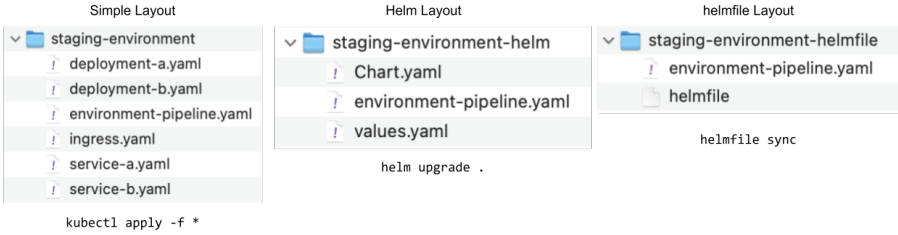


Figure 1.7 Three different approaches for defining environments' configurations.

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

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

The third option is to use a project called "helmfile" (https://github.com/roboll/helmfile), which was designed for this very specific purpose, to define environment configurations. A helmfile allows you to declaratively define what Helm releases need to be present in our cluster. This Helm releases will be created when we run `helmfile sync`, having defined a helmfile containing the helm releases that we want to have in the cluster.

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

It is common to have several environments (staging, QA, production), and even allow 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 1.8, 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.

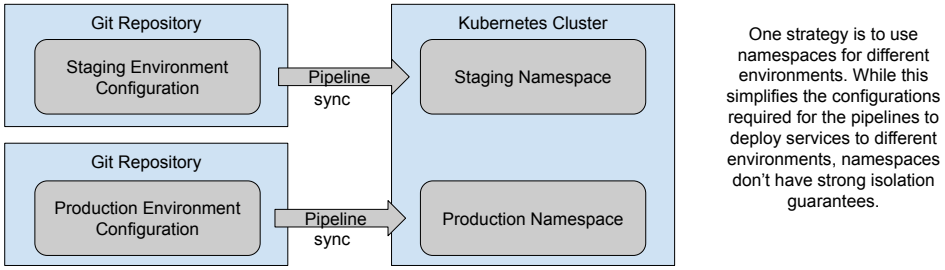


Figure 1.8 One Environment per Kubernetes Namespace.

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 1.9 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.

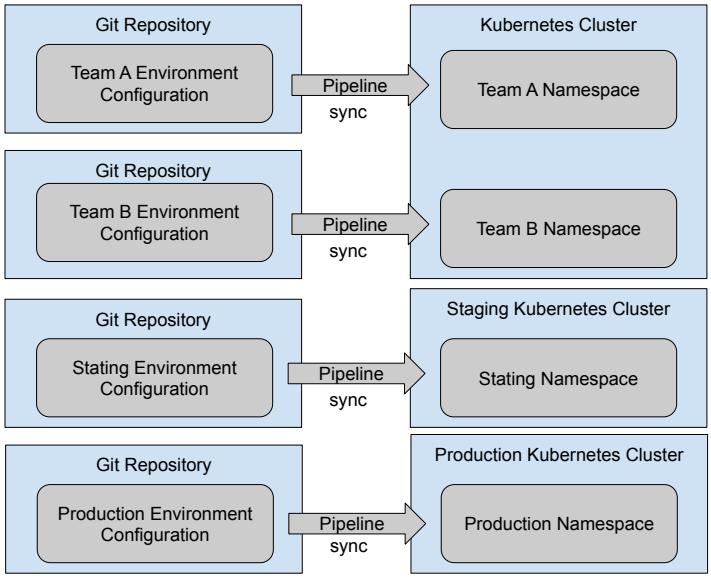


Figure 1.9 One environment per Kubernetes cluster.

Okay, but how can we implement these pipelines? Should we implement these pipelines using Tekton? In the next section, we will look at Argo CD (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.

1.2 Environment pipelines in action

You can definitely go ahead and implement an environment pipeline as described in the previous section using Tekton or Dagger. This has been done in projects like Jenkins X (https://jenkins-x.io), but nowadays, the steps for an environment pipeline are encoded in specialized tools for continuous deployment like Argo CD (https://argo-cd.readthedocs.io/en/stable/).

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

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

1.2.1 Argo CD

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

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

Argo CD is not limited to a single cluster, meaning that we can have our environment living in separate clusters, even in different cloud providers (figure 1.10).

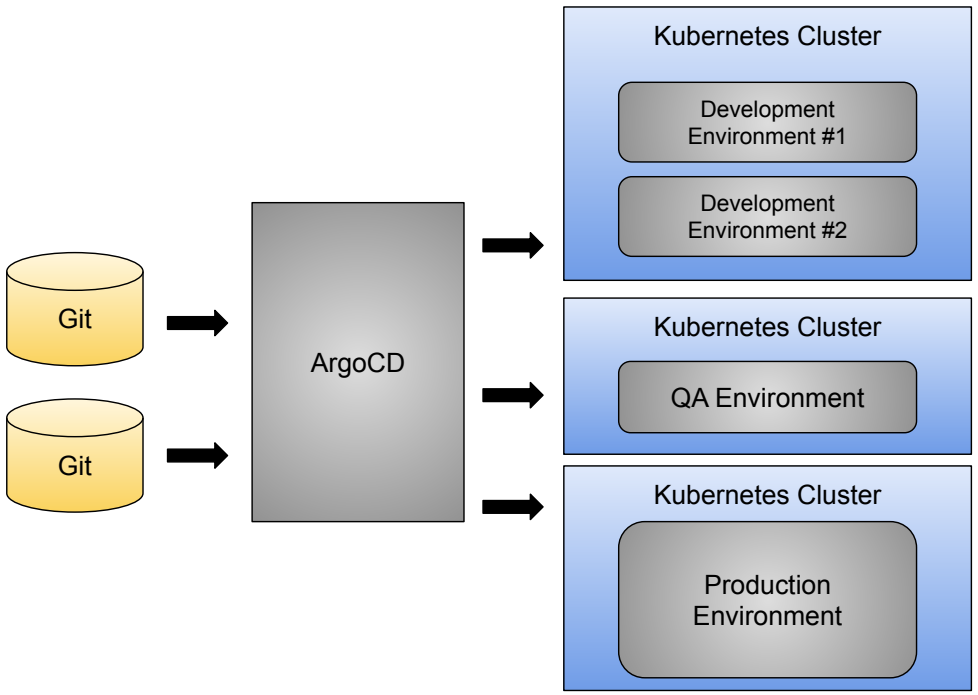


Figure 1.10 Argo CD will sync environments configurations from Git to live clusters.

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. Argo CD can monitor repositories or directories inside repositories for changes to sync our environments' configurations.

For this example, we will install Argo CD 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.

You can follow a step-by-step tutorial located at https://github.com/salaboy/from-monolith-to-k8s/tree/main/argocd.

For installing Argo CD, 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 Argo CD to work, so after finishing this guide, we should have all we need to get our staging environment going.

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

Argo CD 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 installation and interaction 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 Argo CD provides an out-of-the-box Helm integration.

> **NOTE** Argo CD uses a different nomenclature than the one that we have been using here. In Argo CD, you configure applications instead of environments. In the following figures, you will see that we will be configuring an Argo CD application to represent our staging environment. Because there are no restrictions on what you can include in a Helm Chart, we will be using a Helm Chart to configure our Conference application into this environment.

1.2.2 Creating an Argo CD application

If you access the Argo CD user interface, you will see right in the top left corner of the screen the "+ New App" button (figure 1.11).

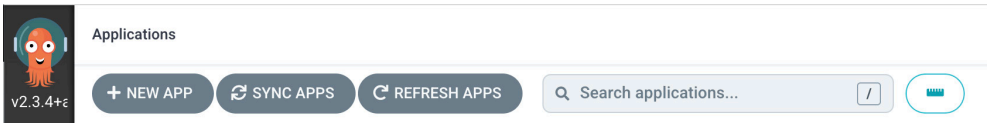


Figure 1.11 Argo CD user interface - new application creation.

Go ahead and hit that button to see the Application creation form. Besides adding a name and selecting a Project where our Argo CD application will live (we will select the default project) and then check the `Auto-Create Namespace` option (figure 1.12).

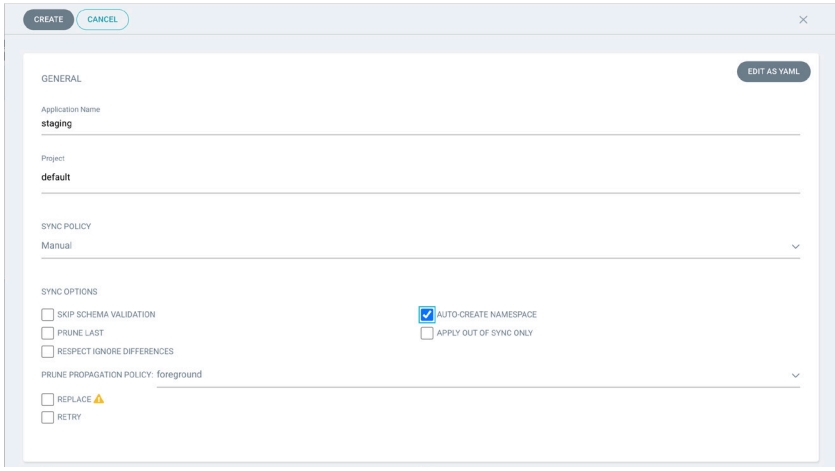


Figure 1.12 **New application parameters, manual sync, and auto-create namespace.**

By associating our environment to a new namespace in our cluster, we can use the Kubernetes RBAC mechanism only to allow administrators to modify the Kubernetes resources located in that namespace. Remember that by using Argo CD, we want to ensure that developers don't accidentally change the application configuration or manually apply configuration changes to the cluster. Argo CD will take care of syncing the resources defined in a Git repository. So where is that Git repository? That's exactly what we need to configure next in figure 1.13.

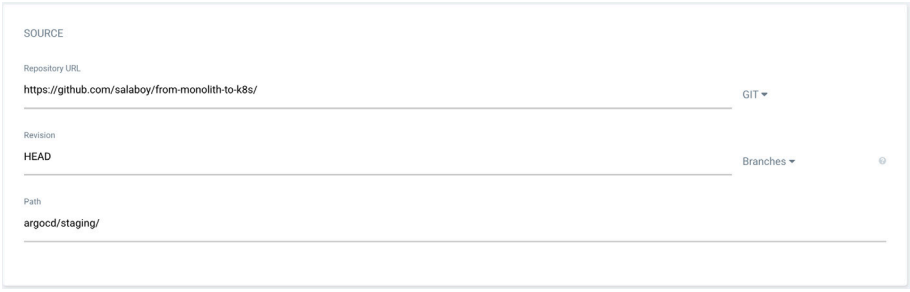


Figure 1.13 **Argo CD application's configuration repository, revision, and path.**

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 (and then use your fork URL) so you can make any changes that you want to the environment configuration.

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

The next step is to define where this environment configuration will be applied by Argo CD. As mentioned before, we can use Argo CD to install and sync environments in different clusters, but for this example we will use the same Kubernetes cluster where we installed Argo CD, because the namespace will be automatically created, make sure to enter `staging` as the namespace so Argo CD creates it for you (figure 1.14).

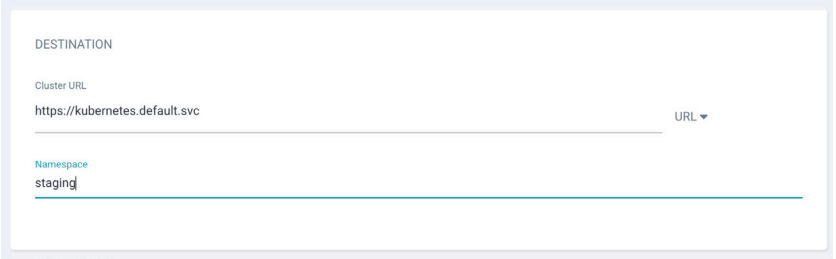


Figure 1.14 Configuration destination, for this example, is the cluster where Argo CD is installed.

Finally, because it makes sense to reuse the same configuration for similar environments, Argo CD enables us to configure different parameters specific to this installation. Since we are using Helm and the Argo CD 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, Argo CD allows us to set up Environment Variables that we can use as parameters for our configuration scripts (figure 1.15).

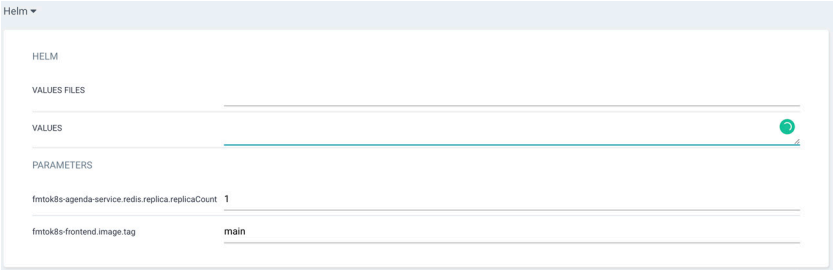


Figure 1.15 Helm configuration parameters for the staging environment.

As you can see in the previous image, Argo CD also identified that there are non-empty `values.yaml` files inside the repository path that we have provided, and it is automatically parsing the parameters. We can add more parameters to the "`VALUES`" text box to override any other chart (or sub-charts) configurations.

After we provide all this configuration, we are ready to hit the "`Create`" button at the top of the form.

Argo CD will create the application, but because we have selected Manual Sync it will not automatically apply the configuration to the cluster (figure 1.16).

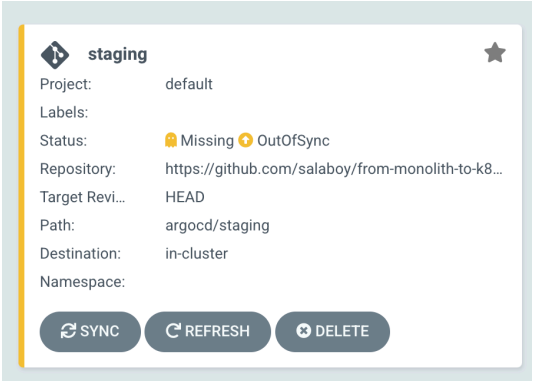


Figure 1.16 Application created but not synced.

If you click into the application, you will drill down to the application's full view, which shows you the state of all the resources associated with the application (figure 1.17).

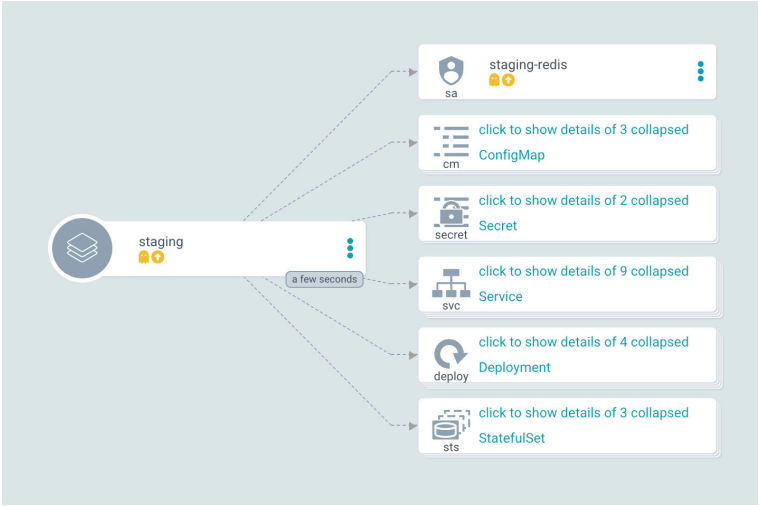


Figure 1.17 Application resources before sync.

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 1.18).

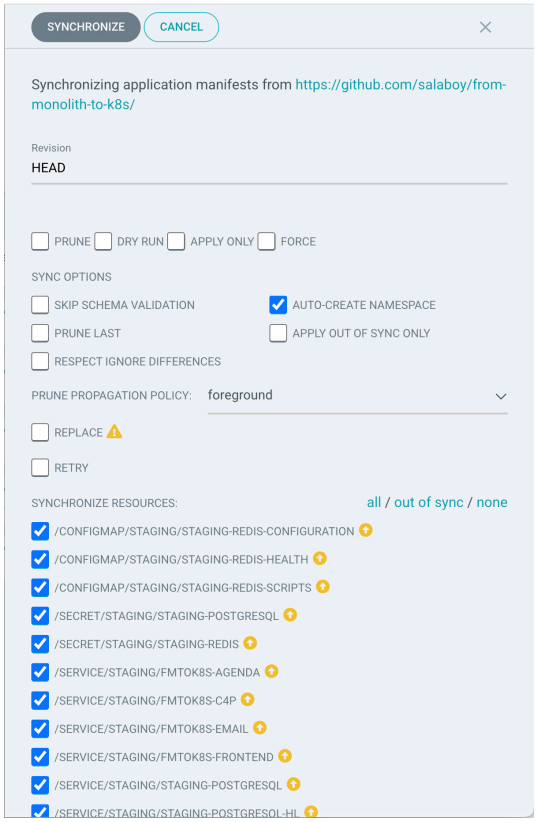


Figure 1.18 Application sync parameters.

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 created and monitored by Argo CD (figure 1.19).

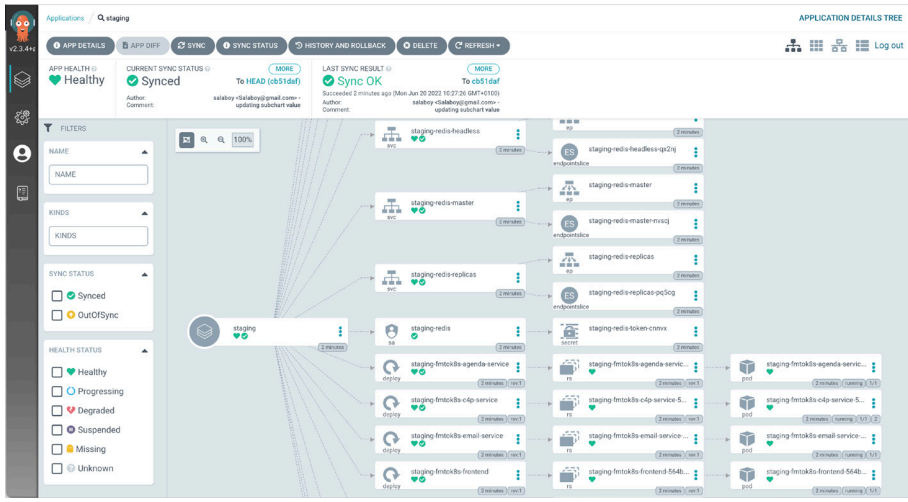


Figure 1.19 **Our staging environment is Healthy, and all the services are up and running.**

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

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

I strongly recommend you follow you the step-by-step tutorial to get hands-on with tools like Argo CD.

1.2.3 Dealing with changes, the GitOps way

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

artifacts using the release number. The creation of these artifacts is the responsibility of the service pipeline, as we saw in previous sections (figure 1.20).

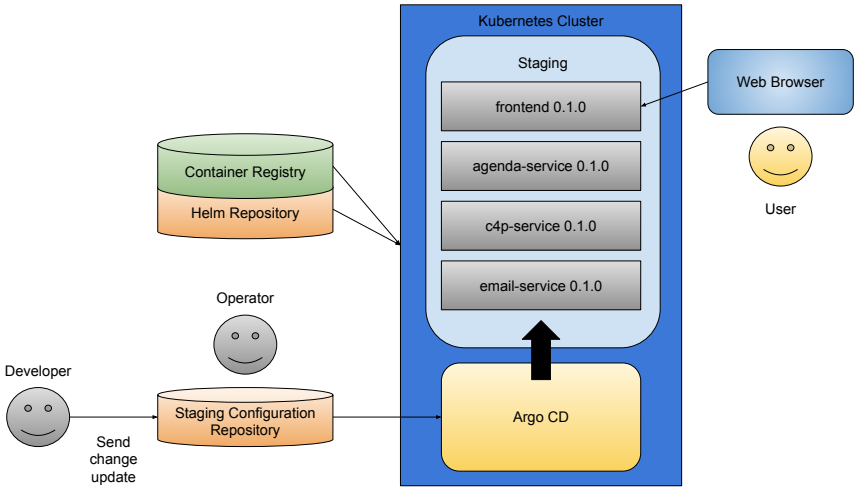


Figure 1.20 Components to set up the staging environment with Argo CD.

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

- *Bumping up or reverting a service version:* For our example, this is as simple as changing the version of the chart of one or more services. Rolling back one of the services to the previous is as simple as reverting the version number in the environment chart or even reverting the commit that increased the version in the first place. Notice that reverting commits is always recommended, 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 Argo CD installation. Suppose the services' charts are available and the configuration parameters are valid. In that case, the next time that we sync our Argo CD application, the new services 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 be trying to finetune the application parameters to accommodate performance or scalability requirements, monitoring configurations, or the log level for a set of services. These kinds of changes are also versioned and should be treated as new features and bug fixes.

If we compare this with manually using Helm to install the application into the cluster, we will quickly notice the differences. First, a developer might have the environment configuration on a laptop, making the environment very difficult to replicate from a different location. Changes to the environment configuration that are not tracked 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. Configuration drifts are much more difficult to track down and troubleshoot.

Following this automated approach with Argo CD can open the door to more advanced scenarios. For example, we can create preview environments (figure 1.21) for our pull requests to test changes before they get merged and artifacts are released.

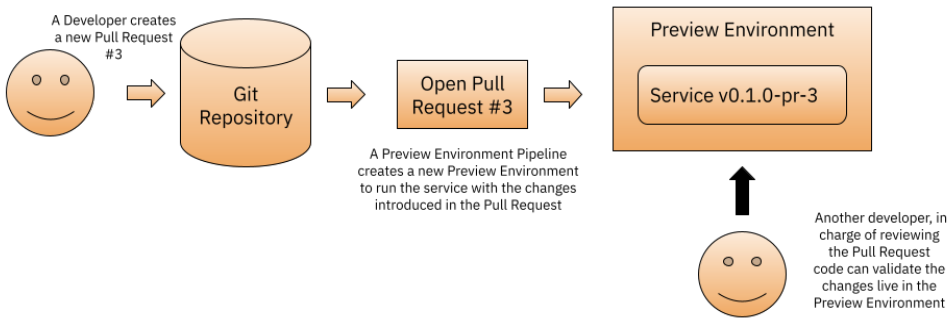


Figure 1.21 **Preview environments for faster iterations.**

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, so an automated clean-up mechanism is straightforward to implement.

NOTE Another important detail to mention when using Argo CD 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, but Argo CD will not use this Helm feature. Argo CD takes the approach of using "helm template" to render the Kubernetes resources YAML, and then it does apply the output using "kubectl apply". This approach relies on the fact that everything is versioned in Git and also allows the unification of different templating engines for YAML. In addition to some security benefits, this is key to enabling diffing in Argo CD, which allows us to specify which resources should be managed by Argo CD and which elements may be managed by different controllers.

Finally, to tie things together, let's see how the service and environment pipelines interact to provide end-to-end automation from code changes to deploying new versions into multiple environments.

1.3 Service + environment pipelines

Let's look at how service pipelines and environment pipeline connect. The connection between these two pipelines happens via pull/change requests to Git repositories, because the pipelines will be triggered when changes are submitted and merged (figure 1.22).

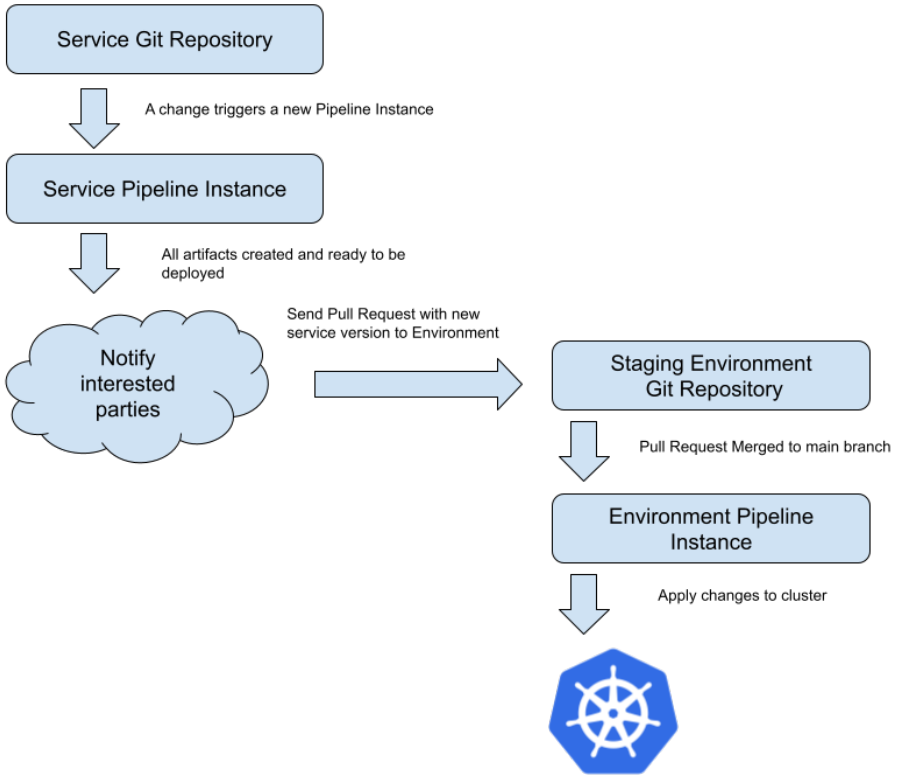


Figure 1.22 A service pipeline can trigger an environment pipeline via a pull request.

Developers, when they finish a new feature, create a pull/change request to the repository's main branch. This pull/change request can be reviewed and built by a specialized service pipeline. When this new feature is merged into the repository's main branch, a new instance of the service pipeline is triggered. This instance creates a new release and all the artifacts needed to deploy the service's new version into a Kubernetes cluster. This includes a binary with the compiled source code, a container image, and Kubernetes manifests that can be packaged using tools like Helm.

As the last step of the service pipeline, you can include a notification step that can notify the interested environments that there is a new version of a service that they are running available. This notification is usually an automated pull/change request into the environment's repository. Alternatively, you can monitor (or be subscribed to notifications) for your artifact repositories, and when a new version is detected a pull/change request is created to the configured environments.

The pull/change requests created to environment repositories can be automatically tested by a specialized Environment Pipeline, in the same way as we did with service pipelines, and for low-risk environments, these pull/change requests can be automatically merged without any human intervention.

By implementing this flow, we can enable developers to focus on fixing bugs and creating new features that will be automatically released and promoted to low-risk environments.

Once the new versions are tested in environments like staging, and we know that these new versions or configurations are not causing any problems, a pull/change request can be created for the repository that contains the production environment configuration (figure 1.23).

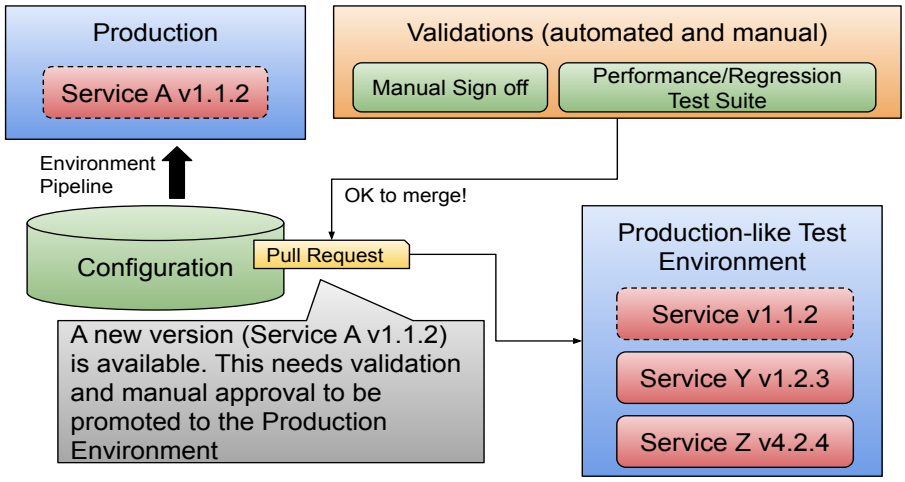


Figure 1.23 **Promoting changes to the production environment.**

The more sensitive the environments are, the more checks and validations they will require. In this case, as shown in figure 1.23, to promote a new service version to the production environment, a new test environment will be created to validate and test the changes introduced in the pull/change request submitted. Once those validations are done, a manual sign-off is required to merge the pull request and trigger the environment pipeline synchronization.

At the end of the day, environment pipelines are the mechanism you use to encode your organization's requirements to release and promote software to different environments. We have seen in this chapter what a tool like Argo CD can do for us. Next, we need to evaluate whether a single Argo CD installation would be enough and who will manage it and keep it secure. Do you need to extend Argo CD with custom hook points? Do you need to integrate it with other tools? These are all valid questions we will explore in chapter 3, but now it is time to tackle one more important topic: application infrastructure.

Summary

- Environment pipelines are responsible for deploying software artifacts to live environments. Environment pipelines avoid teams interacting directly with the cluster where the applications run, reducing errors and misconfigurations.
- Using tools like Argo CD, you can define the content of each environment into a Git repository that is used as the source of truth for what the environment configuration should look like. Argo CD will keep track of the state of the cluster where the environment is running and make sure that there is no drift in the configuration that is applied in the cluster.
- Teams can upgrade or downgrade the versions of the services running in an environment by submitting pull/change requests to the repository where the environment configuration is stored. A team or an automated process can validate these changes, and when approved and merged, these changes will be reflected in the live environment. If things go wrong, changes can be rolled back by reverting commits to the Git repository.

2 Multi-cloud (app) infrastructure

This chapter covers

- Defining and managing infrastructure
- Managing your application infrastructure components
- Using Kubernetes to deal with infrastructure with Crossplane

In the previous chapters, we installed a walking skeleton, we understood how to build each separate component using service pipelines, and we learned how to deploy them into different environments using environment pipelines. We are now faced with a big challenge: dealing with our application infrastructure, meaning running and maintaining not only our application services but the components that these services need to run. These applications are expected to require other components to work correctly, such as databases, message brokers, identity management solutions, email servers, etc. While several tools out there are focused on automating the installation or provisioning of these components for on-premise setups and in different cloud providers, in this chapter, we will focus on one that does it in a Kubernetes way. This chapter is divided into three main sections:

- The challenges of dealing with infrastructure.
- How to deal with infrastructure leveraging Kubernetes constructs.
- How to provision infrastructure for our walking skeleton using Crossplane.

Let's get started. Why is it so difficult to manage our application infrastructure?

2.1 The challenges of managing Infrastructure in Kubernetes

When you design applications like the walking skeleton, you face specific challenges that are not core to achieving your business goals. Installing, configuring, and maintaining application infrastructure components that support our application's services is a big task that needs to be planned carefully by the right teams with the right expertise.

These components can be classified as application infrastructure, which usually involves third-party components not being developed in-house, such as databases, message brokers, identity management solutions, etc. A big reason behind the success of modern cloud providers is that they are great at providing and maintaining these components and allowing your development teams to focus on building the core features of applications, which bring value to the business.

It is important to distinguish between application infrastructure and hardware infrastructure, which is also needed. In general, I assume that for public cloud offerings, the provider solves all hardware-related topics. For on-prem scenarios, you likely have a specialized team taking care of the hardware (removing, adding, and maintaining hardware as needed).

It is common to rely on cloud provider services to provision application infrastructure. There are a lot of advantages to doing so, such as pay-as-you-use services, easy provisioning at scale, and automated maintenance. But at that point, you heavily rely on provider-specific ways of doing things and their tools. The moment you create a database or a message broker in a cloud provider, you are jumping outside the realms of Kubernetes. Now you depend on their tools and their automation mechanisms, and you are creating a strong dependency of your business with the specific cloud provider.

Let's look at the challenges associated with provisioning and maintaining application infrastructure, so your teams can plan and choose the right tool for the job:

- *Configuring components to scale:* Each component will require different expertise to be configured (database administrators for databases and message broker experts) and a deep understanding of how our application's services will use it, as well as the hardware available. These configurations need to be versioned and monitored closely, so new environments can be created quickly to reproduce problems or test new versions of our application.
- *Maintaining components in the long run:* Components such as databases and message brokers are constantly released and patched to improve performance and security. This pushes the operations teams to ensure they can upgrade to newer versions and keep all the data safe without bringing down the entire application. This requires a lot of coordination and impact analysis between the teams involved with these components and services.
- *Cloud provider services affect our multi-cloud strategy:* If we rely on cloud-specific application infrastructure and tools, we need to find a way to enable developers to create and provision their components for developing and testing their

services. We need a way to abstract how infrastructure is provisioned to enable applications to define what infrastructure they need without relying directly on cloud-specific tools.

Interestingly, we had these challenges even before having distributed applications, and configuration and provisioning architectural components have always been hard and usually far away from developers. Cloud providers are doing a fantastic job by bringing these topics closer to developers so they can be more autonomous and iterate faster. Unfortunately, when working with Kubernetes, we have more options that we need to consider carefully to make sure that we understand the tradeoffs. The next section covers how we can manage our application infrastructure inside Kubernetes; while this is usually not recommended, it can be practical and cheaper for some scenarios.

2.1.1 Managing your application infrastructure

Application infrastructure has become an exciting arena. With the rise of containers, every developer can bootstrap a database or message broker with a couple of commands, which is usually enough for development purposes. In the Kubernetes world, this translates to Helm Charts, which uses containers to configure and provision databases (relational and NoSQL), message brokers, identity management solutions, etc. As we saw, you installed the walking skeleton application containing four services and two databases with a single command.

For our walking skeleton, we are provisioning a Redis NoSQL database for the Agenda Service and a PostgreSQL database for the Call for Proposals (C4P) Service using Helm Charts. The number of Helm charts available today is amazing, and it is quite easy to think that installing a Helm Chart is the way to go.

As discussed previously, if we want to scale our services that keep state, we must provision specialized components, such as databases (figure 2.1). Application developers will define which kind of database will suit them best depending on the data they need to store and how that data will be structured.

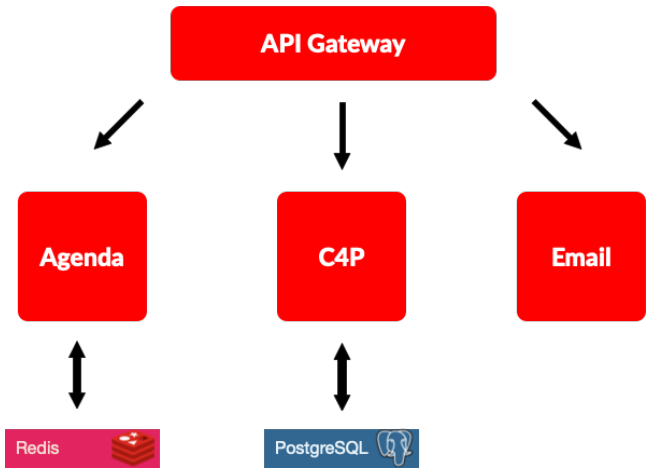


Figure 2.1 Keeping service state using application infrastructure.

The process of setting up these components inside your Kubernetes Cluster involves the following steps (figure 2.2):

- Finding or creating the right Helm Chart for the component you want to bootstrap. For this case, PostgreSQL (https://github.com/bitnami/charts/tree/master/bitnami/postgresql) and Redis (https://github.com/bitnami/charts/tree/master/bitnami/redis) can be found in the Bitnami Helm Chart repository. If you cannot find a Helm Chart, but you have a Docker container for the component that you want to provision, you can create your chart after you define the basic Kubernetes constructs needed for the deployment.
- Research the Chart configurations and parameters you will need to set up to accommodate your requirements. Each chart exposes a set of parameters that can be tuned for different use cases; check the chart website to understand what is available. Here you might want to include your operations teams and DBAs to check how the databases need to be configured. This will also require Kubernetes expertise to make sure that the components can work in HA (high availability) mode inside Kubernetes.
- Install the chart into your Kubernetes Cluster using `helm install`. By running `helm install` you are downloading a set of Kubernetes manifest (YAML files) that describe how your applications/services need to be deployed. Helm will then proceed to apply these YAML files to your cluster. Alternatively, you can define a dependency on your application's services as we did for the walking skeleton.
- Configure your service to connect to the newly provisioned components. This is done by giving the service the right URL and credentials to connect to the newly created resource. For a database, it will be the database URL serving requests and possibly a username and password. An interesting detail to notice here is that your application will need some kind of driver to connect to the target database.
- Maintain these components in the long run, doing backups and ensuring the fail-over mechanisms work as expected.

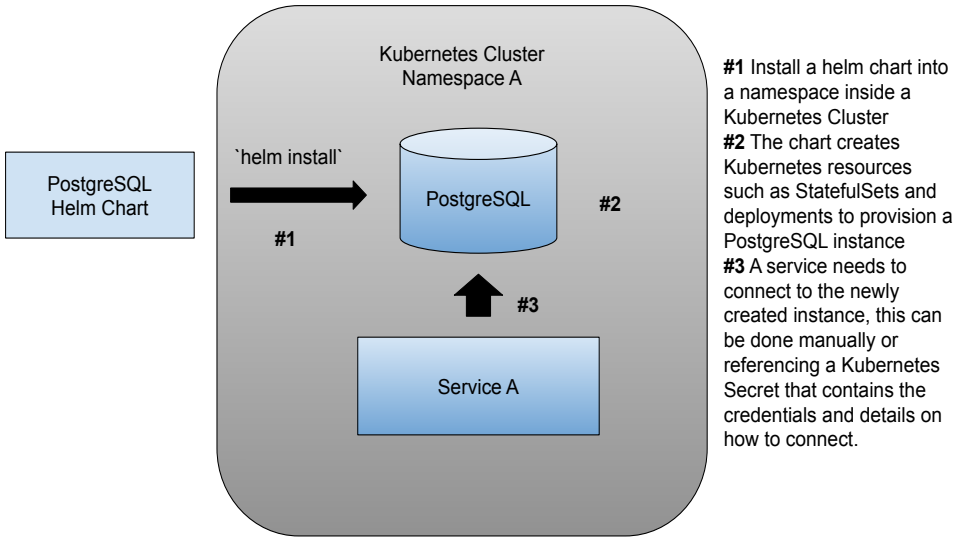


Figure 2.2 Provisioning a new PostgreSQL instance using the PostgreSQL Helm Chart.

If you are working with Helm Charts, there are a couple of caveats and tricks that you need to be aware of:

- If the chart doesn't allow you to configure a parameter that you are interested in changing, you can always use `helm template`, then modify the output to add or change the parameters that you need to finally install the components using `kubectl apply -f`. Alternatively, you can submit a pull request to the chart repository. It is a common practice not to expose all possible parameters and wait for community members to suggest more parameters to be exposed by the chart. Don't be shy and contact the maintainers if that is the case. Whatever modification you do, the chart content must be maintained and documented. By using `helm template`, you lose the Helm release management features, which allow you to upgrade a chart when a new chart version is available.
- Most charts come with a default configuration designed to scale, meaning that the default deployment will target high-availability scenarios. This results in charts that, when installed, consume a lot of resources that might not be available if you use Kubernetes KinD or Minikube on your laptop. Once again, chart documentation usually includes special configurations for development and resource-constrained environments.
- If you are installing a database inside your Kubernetes Cluster, each database container (pod) must have access to storage from the underlying Kubernetes node. For databases, you might need a special kind of storage to enable the database to scale elastically, which might require advanced configurations outside Kubernetes.

For our walking skeleton, for example, we set up the Redis chart to use the `architecture` parameter to `standalone`, (as you can see in the environment pipeline configurations, but also in the Agenda Service Helm Chart values.yaml file) to make it easier to run on environments where you might have limited resources, such as your laptop/workstation. This affects Redis's availability to tolerate failure, because it will only run a single replica in contrast with the default setup where a master and two slaves are created.

2.1.2 Connecting our services to the newly provisioned Infrastructure

Installing the charts will not make our application services automatically connect to the Redis and PostgreSQL instances. We need to provide these services the configurations needed to connect, but also be conscious about the time needed by these components, such as databases, to start (figure 2.3).

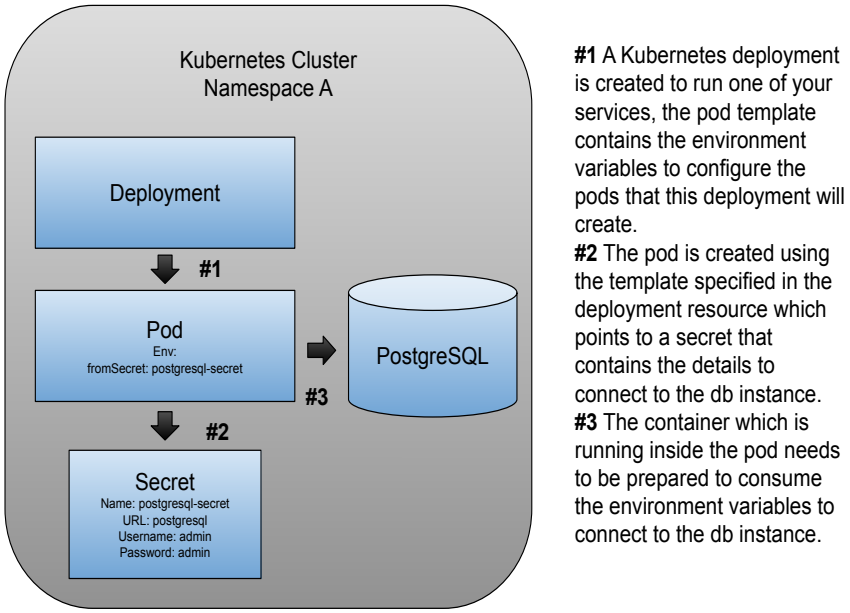


Figure 2.3 Connecting a service to a provisioned resource using secrets.

A common practice is to use Kubernetes Secrets to store the credentials for these application infrastructure components. The Helm Chart for Redis and PostgreSQL that we are using for our walking skeleton both create a new Kubernetes Secret containing the details required to connect. These Helm Charts also create a Kubernetes Service to be used as the location (URL) where the instance will be running.

To connect the Call for Proposals (C4P) Service to the PostgreSQL instance, you need to make sure that the Kubernetes Deployment for the C4P service (`conference-fmtok8s-c4p`) has the right environment variables:

```
- name: SPRING_DATASOURCE_DRIVERCLASSNAME
  value: org.postgresql.Driver
- name: SPRING_DATASOURCE_PLATFORM
  value: org.hibernate.dialect.PostgreSQLDialect
- name: SPRING_DATASOURCE_URL
  value: jdbc:postgresql://${DB_ENDPOINT}:${DB_PORT}/postgres
- name: SPRING_DATASOURCE_USERNAME
  value: postgres
- name: SPRING_DATASOURCE_PASSWORD
  valueFrom:
    secretKeyRef:
      key: postgres-password
      name: postgresql
- name: DB_ENDPOINT
  value: postgresql
- name: DB_PORT
  value: "5432"
```

In bold are highlighted how we can consume the dynamically generated password when we install the chart and the DB endpoint URL, which in this case is the PostgreSQL Kubernetes Service, also created by the chart. The DB endpoint would be different if you used a different chart release name. A similar configuration applies to the Agenda Service and Redis:

```
- name: SPRING_REDIS_HOST
  value: redis-master
- name: SPRING_REDIS_PASSWORD
  valueFrom:
    secretKeyRef:
      key: redis-password
      name: redis
```

As before, we extract the password from a secret called `redis`. The `REDIS_HOST` is obtained from the name of the Kubernetes Service that is created by the chart; this depends on the `helm release` name that you have used.

Being able to configure different instances for the application infrastructure components opens the door for us to delegate the provisioning and maintenance to other teams and even cloud providers.

2.1.3 I've heard about Kubernetes Operators. Should I use them?

Now you have four application services and two databases inside your Kubernetes cluster. Believe it or not, now you are in charge of six components to maintain and scale depending on the application's needs. The team that built the services will know exactly how to maintain and upgrade each service, but they are not experts in maintaining and scaling databases.

You might need help with these databases depending on how demanding the services are. Imagine if you have too many requests on the Agenda Service, so you decide to scale up the number of replicas of the Agenda deployment to 200. At that point, Redis must have enough resources to deal with 200 pods connecting to the Redis cluster. The advantage of using Redis for this scenario, where we might get a lot of reads while the conference is ongoing, is that the Redis Cluster allows us to read data from the replicas so the load can be distributed (figure 2.4).

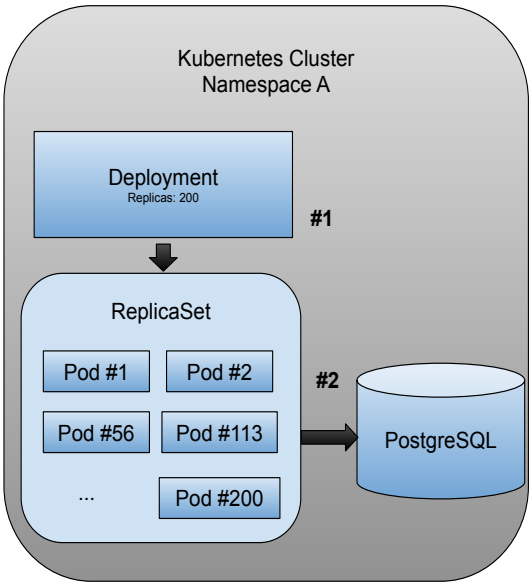


Figure 2.4 Application infrastructure needs to be configured according how our services will be scaled.

If you are installing application infrastructure with Helm, notice that Helm will not check for the health of these components, it is just doing the installation. It is quite common nowadays to find another alternative to install components in a Kubernetes cluster called Operators. Usually associated with application infrastructure, you can find more active components that will install and monitor the installed components. One example of these Operators is the Zalando PostgreSQL Operator, which you can find here: https://github.com/zalando/postgres-operator. While these operators are also focused on allowing you to provision new instances of PostgreSQL databases, they also implement other features focused on maintenance, for example:

- Rolling updates on Postgres cluster changes, including quick minor version updates
- Live volume resize without pod restarts (AWS EBS, PVC)

- Database connection pooling with PGBouncer
- Support fast in-place major version upgrades

In general, Kubernetes Operators try to encapsulate the operational tasks associated with a specific component, in this case, PostgreSQL. While using Operators might add more features on top of installing a given component, you still need to maintain the component and the operator itself now. Each Operator comes with a very opinionated flow that your teams will need to research and learn to manage these components. Take this into consideration when researching and deciding which Operator to use.

Regarding the application infrastructure that you and your teams decide to use, if you plan to run these components inside your cluster, plan accordingly to have the right in-house expertise to manage, maintain, and scale these extra components.

In the following section, we will look at how we can tackle these challenges by looking at an open-source project that aims to simplify the provisioning of cloud and on-prem resources for application infrastructure components by using a declarative approach.

2.2 Declarative infrastructure using Crossplane

Using Helm to install application infrastructure components inside Kubernetes is far from ideal for large applications and user-facing environments, because the complexity of maintaining these components and their requirements, such as advanced storage configurations, might become too complex to handle for your teams.

Cloud providers do a fantastic job at allowing us to provision infrastructure, but they all rely on cloud provider-specific tools, which are outside of the realms of Kubernetes.

In this section, we will look at an alternative tool; a CNCF project called Crossplane (https://crossplane.io), which uses the Kubernetes APIs and extension points to enable users to provision real infrastructure in a declarative way, using the Kubernetes APIs. Crossplane relies on the Kubernetes APIs to support multiple cloud providers and it integrates nicely with all the existing Kubernetes tooling.

By understanding how Crossplane works and how it can be extended, you can build a multi-cloud approach to build and deploy your cloud-native applications into different providers without worrying about getting locked in on a single vendor. Because Crossplane uses the same declarative approach as Kubernetes, you will be able to create your high-level abstractions about the applications that you are trying to deploy and maintain.

To use Crossplane, you first need to install its control plane in a Kubernetes Cluster. You can do this by following their official documentation (https://docs.crossplane.io/) or the step-by-step tutorial introduced in section 2.3.

The core Crossplane components alone will not do much for you. Depending on your cloud provider(s) of choice, you will install and configure one more Crossplane providers. Let's take a look at what Crossplane providers have to offer us.

2.2.1 Crossplane providers

Crossplane extends Kubernetes by installing a set of components called *providers* (https://docs.crossplane.io/v1.11/concepts/providers/) in charge of understanding and interacting with cloud provider-specific services to provision these components for us (figure 2.5).

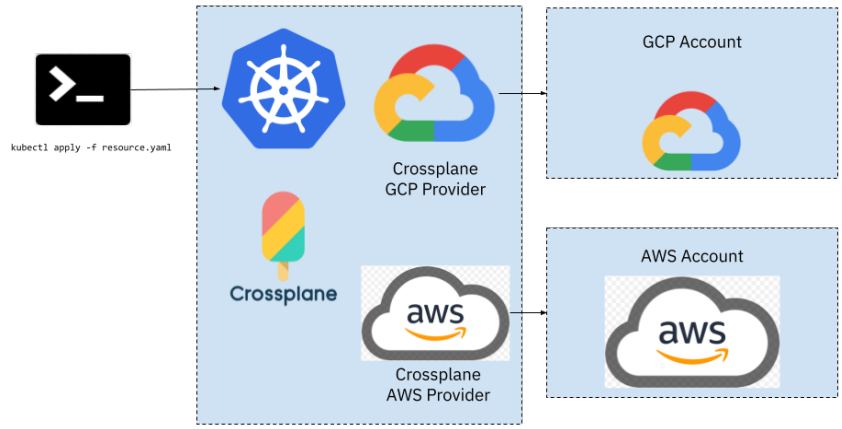


Figure 2.5 **Crossplane installed with GCP and AWS providers.**

By installing Crossplane providers, you are extending the Kubernetes APIs functionality to provision external resources such as databases, message brokers, buckets, and other cloud resources that will live outside your Kubernetes cluster but inside the cloud provider realm. There are several Crossplane providers covering the major cloud providers such as GCP, AWS, and Azure. You can find these Crossplane providers in the Crossplane GitHub organization: https://github.com/crossplane/.

Once a Crossplane provider is installed, you can create provider-specific resources in a declarative way, which means that you can create a Kubernetes Resource, apply it with `kubectl apply -f`, package these definitions in Helm Charts or use environment pipelines storing these resources in a Git repository.

For example, creating a bucket in Google Cloud using the Crossplane GCP provider looks like this:

```
cat <<EOF | kubectl create -f -
apiVersion: storage.gcp.upbound.io/v1beta1
kind: Bucket
metadata:
  generateName: crossplane-bucket-
  labels:
    docs.crossplane.io/example: provider-gcp
spec:
  forProvider:
    location: US
  providerConfigRef:
    name: default
EOF
```

Both apiVersion and kind are defined by the Crossplane GCP provider, you can find all the supported type of resources in the Crossplane provider documentation.

By creating a bucket resource in our Kubernetes cluster where Crossplane is installed, you are creating a request for Crossplane to provision and monitor this resource on your behalf.

For each resource type, you have a set of parameters to configure the resource. In this case, we want the bucket to be in the US. Different resources will expose different configuration parameters.

Provisioning cloud-specific resources relying on the Kubernetes APIs is a big step forward, but Crossplane doesn't stop there. If you look at what it takes to provision a database in any major cloud provider, you will realize that provisioning the component is just one of the tasks involved in getting the component ready to be used. You will need network and security configurations and user credentials to connect to these provisioned components.

2.2.2 Crossplane compositions

Crossplane aims to serve two different personas: *platform teams* and *application* teams. While *platform* teams are cloud providers experts that understand how to provision cloud provider-specific components, *application* teams know the application requirements and understand what is required from the application infrastructure perspective. The interesting thing about this approach is that when using Crossplane, platform teams can define these complex configurations for a specific cloud provider and expose simplified interfaces for application teams.

In real-life scenarios, it is rare to just create a single component, for example, if we want to provision a database instance application teams will also require the correct network and security configurations to be able to access the newly created instance. Being able to compose and wire up together several components is a very convenient feature and for achieving these abstractions and simplified interfaces Crossplane introduced two concepts *composite resourced definitions (XRDs)* and c*omposite resources (XR)* (figure 2.6).

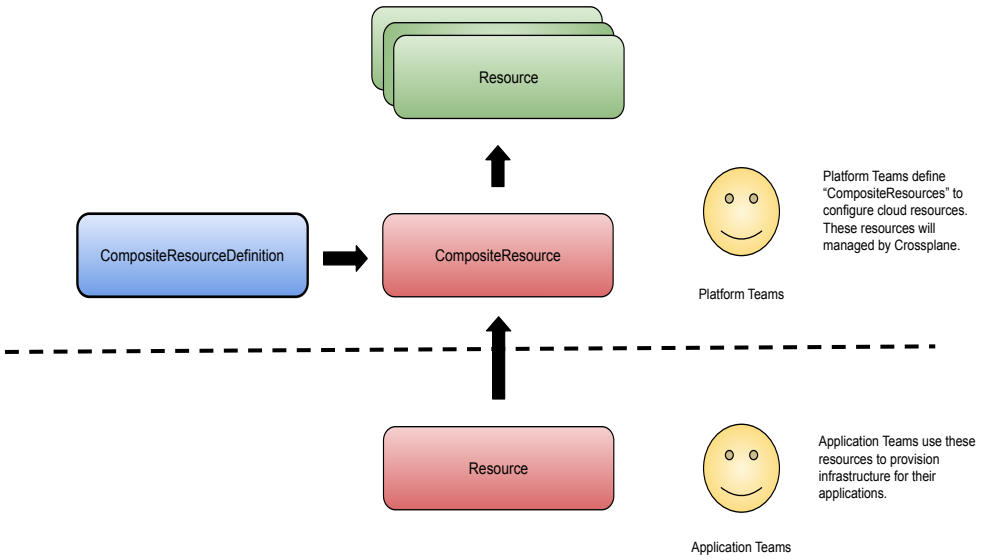


Figure 2.6 Resource composition abstraction by Crossplane composite resources.

The previous diagram shows how you can use Crossplane *Composite Resources (XRD)* to define abstractions for different cloud providers. The platform team might be very knowledgeable in Google Cloud or Azure, so they will be in charge of defining which resources they want to wire up together for a specific application. The application team has a simple resource interface to request the resource they are interested in. But as usual, abstractions are complicated and good to show who is responsible for what, so let's look at a concrete example to understand the power of Crossplane compositions (figure 2.7).

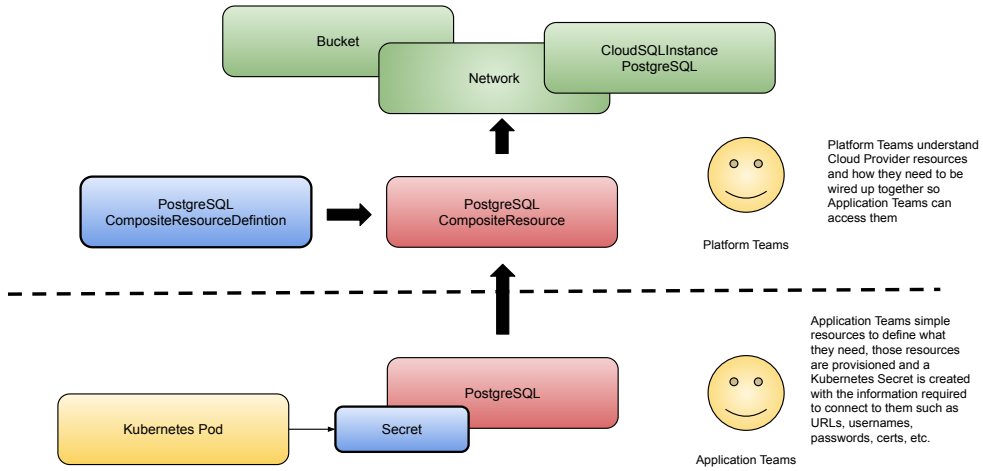


Figure 2.7 Provisioning a PostgreSQL instance in Google Cloud with Crossplane compositions.

Figure 2.7 shows how the application team can create a simple PostgreSQL resource to a provision in Google Cloud, a CloudSQLInstance, plus a network configuration and a bucket. The application team is interested in something other than what resources are created or even in which cloud provider they were created. They are only interested in having a PostgreSQL instance to connect their applications to.

This takes us to the `Secret` box in the figure, representing a Kubernetes secret that Crossplane will create for our application/services pods to connect to the provisioned resources. Crossplane creates this Kubernetes secret with all the details our applications require to connect to the newly created resources (or just with the one relevant to the application). This secret typically contains URLs, usernames, passwords, certificates, or anything required for your applications to connect. Platform teams define what will be included in the secret when defining the CompositeResources. In the following sections, when we add real infrastructure to our Conference application, we will explore how these `CompositeResourceDefinitions` look and how they can be applied to create all the components that our applications need.

2.2.3 Crossplane components and requirements

To work with Crossplane providers and `CompositeResourceDefinitions` we need to understand how Crossplane components will work together to provision and manage these components inside different cloud providers.

This section covers what Crossplane needs to work and how Crossplane components will manage our CompositeResources.

First, it is important to understand that you must install Crossplane in a Kubernetes cluster. This can be the cluster where your applications run or a separate cluster where Crossplane will run. This cluster will have some Crossplane components that will understand our `CompositeResourceDefinitions` and have enough permissions on the cloud platform to provision resources on our behalf (figure 2.8).

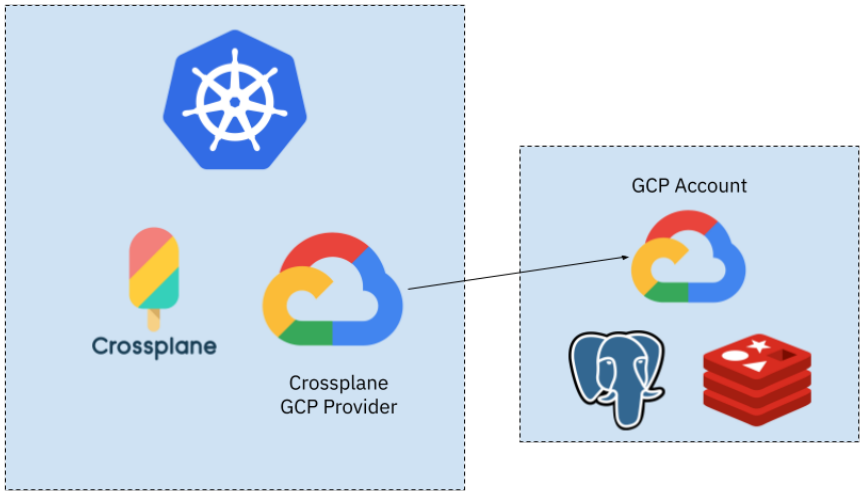


Figure 2.8 **Crossplane in Google Cloud Platform.**

The previous figure shows Crossplane installed inside a Kubernetes cluster, with the Crossplane GCP provider installed and configured to use a Google Cloud Platform account with enough rights to provision PostgreSQL and Redis instances. This means having, in some cases, admin access to create resources on the cloud provider.

For figure 2.8 to work in GCP, you need the following configurations on the cloud provider:

- To create a Redis instance in GCP:
  - Your GCP project needs to have the "redis.googleapis.com" APIs enabled.
  - You also need to have admin rights on the redis resources "roles/redis.admin".
- For creating a PostgreSQL instance in GCP:
  - Your GCP project needs to have the "sqladmin.googleapis.com" APIs enabled.
  - You also need to have admin rights on the SQL resources "roles/cloudsql.admin".

Each Crossplane provider available requires a specific security configuration to work and an account inside the cloud provider where we want to create resources (figure 2.9).

Once a Crossplane provider is installed and configured, in this case, the GCP provider, we can start creating resources managed by this provider. You can find the resources offered by each provider on the following documentation site: https://doc.crds.dev/github.com/crossplane/provider-gcp.

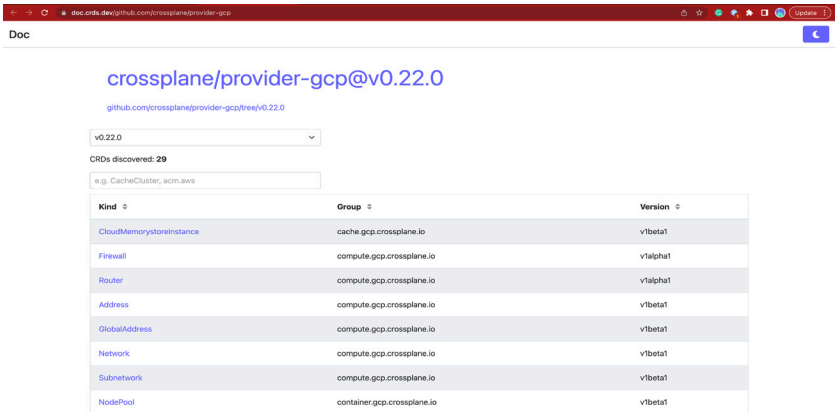


Figure 2.9 Crossplane GCP supported resources.

As you can see in the previous figure, the GCP provider version 0.22.0 supports 29 different CRDs (custom resource definitions) for creating resources in the Google Cloud Platform. Crossplane defines each of these resources as managed resources. Each of these managed resources will need to be enabled for the Crossplane provider to have the right access to the list, and then to create and modify these resources.

In section 2.3, we will look at how to provision cloud or local resources for our applications by using different Crossplane providers and Crossplane compositions. Before jumping into the technical aspects, let's look at Crossplane core behaviors that you should look for when working with tools in the Kubernetes space.

2.2.4 Crossplane behaviors

In contrast to installing Helm components in our Kubernetes clusters, we use Crossplane to interact with the cloud provider-specific APIs to provision resources inside the cloud infrastructure. This should simplify the maintenance tasks and costs related to these resources. Another important difference is that the Crossplane provider (GCP provider in this case) will monitor the created managed resources for us. These managed resources offer some advantages compared with just installed resources using Helm. Managed resources have very well-defined behaviors. Here is a summary of what to expect from a Crossplane managed resource:

- *Visible as any other Kubernetes resource*: Crossplane managed resources are just Kubernetes resources, which means that we can use any Kubernetes tool to monitor and query the state of these resources.
- *Continuous reconciliation*: When a managed resource is created, the provider will continuously monitor the resource to make sure that it exists and is working and report back the status to the Kubernetes resource. The parameters defined inside the managed resource are considered the desired state (source of truth) and providers will work to apply these configurations to the cloud provider resources. Once again, we can use standard Kubernetes tools to monitor changes in state and trigger remediation flows.
- *Immutable properties*: Providers are in charge of reporting back if a user manually changes properties in the cloud provider. The idea here is to avoid configuration drifts from what was defined to what is running in the cloud provider. If so, the state is reported back to the managed resource. Crossplane will not delete the cloud provider resource but will notify back so actions can be taken. Other tools like Terraform (https://www.terraform.io) will automatically delete the remote resources to recreate them.
- *Late initialization*: Some properties in the Managed resources can be optional, meaning each provider will select the default values for these properties. When this happens, Crossplane creates the resource with the default values and then sets the selected values into the managed resource. This simplifies the configuration needed to create resources and reuse the sensible defaults defined by cloud providers, usually in their user interfaces.
- *Deletion*: When deleting a managed resource, the cloud provider immediately triggers the action. However, the managed resource is kept until the resource is fully removed from the cloud provider. Errors that might happen during deletion on the cloud provider will be added to the managed resource status field.
- *Importing existing resources*: Crossplane doesn't necessarily need to create the resources to be able to manage them. You can create managed resources that start monitoring components created before Crossplane was installed. You can achieve this using a specific Crossplane annotation on the managed resource: crossplane.io/external-name.

To summarize the interactions between Crossplane, the Crossplane GCP Provider, and our Managed Resources, let's look at figure 2.10.

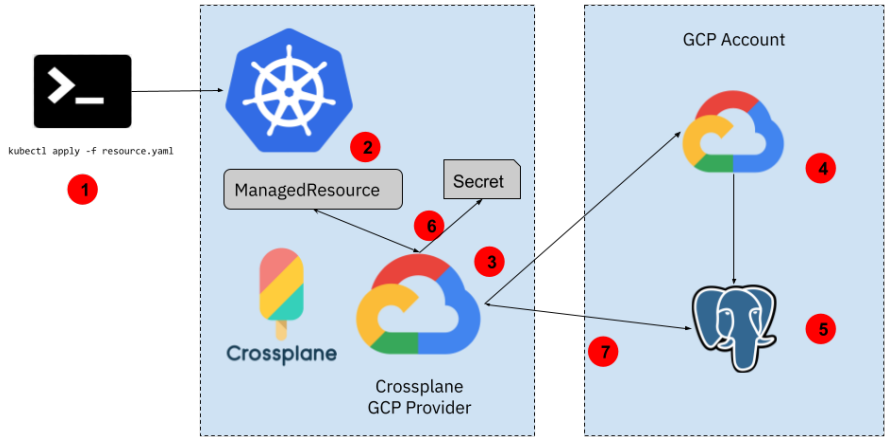


Figure 2.10 Lifecycle of managed resources with Crossplane.

The following points indicate the sequence observed in figure 2.10:

1. First, we need to create a resource. We can use any tool to create Kubernetes resources; `kubectl` here is just an example.
2. If the resource we created is a Crossplane managed resource, let's imagine a CloudSQLInstance resource. The specific Crossplane provider will pick it up and manage it.
3. The first step to execute when managing a resource will be checking whether it exists in the infrastructure (that is, in the configured GCP account). If it doesn't exist, the provider will request that the resource be created in the infrastructure. The appropriate SQL database will be provisioned depending on the properties set on the resource, such as which kind of SQL database is required. Imagine that we have chosen a PostgreSQL database for the sake of the example.
4. The cloud provider, after receiving the request and if the resources are enabled, will create a new PostgreSQL instance with the configured parameters in the managed resource.
5. The status of the PostgreSQL will be reported back to the managed resource, which means that we can use `kubectl` or any other tool to monitor the status of the provisioned resources. Crossplane providers will keep these in sync.
6. When the database is up and running, the Crossplane provider will create a secret to store the credentials and properties that our applications will need to connect to the newly created instance
7. Crossplane will regularly check the status of the PostgreSQL instance and update the managed resource.

By following Kubernetes design patterns, Crossplane uses the reconciliation cycle implemented by controllers to keep track of external resources. Let's see this in action! The following section will examine how we can use Crossplane with our walking skeleton application.

2.3 Infrastructure for our walking skeleton

In this section, we will use Crossplane to abstract away how we provision infrastructure for our Conference application. Because you might not have access to a cloud provider like GCP, AWS, or Azure we will work with a special provider called the Crossplane Helm provider. This Crossplane Helm provider allows us to manage Helm Charts as Cloud Resources. The idea here is to show how with Crossplane, and more specifically using Crossplane Compositions, we can enable users to request resources using a simplified Kubernetes resource to provision local or different cloud resources (hosted in different cloud providers).

For our Conference application, we need a Redis and a PostgreSQL database. From the application perspective, as soon as these two components are available and we can connect to them, all the rest are infrastructural details. The Agenda Service Helm Chart includes the Redis chart dependency:

```
apiVersion: v2
description: FMTOK8s Agenda Service Helm chart for Kubernetes
name: fmtok8s-agenda-service
version: 0.1.0
dependencies:
- name: redis
  version: 17.6.0
  repository: https://charts.bitnami.com/bitnami
  condition: redis.enabled
```

Name of the chart that this dependency will use.

Repository where the chart is hosted.

Custom condition to decide if this dependency is injected when we install the fmtok8s-agenda-service chart.

In the "Call for Proposals" service Helm Chart, the exact mechanism is used to add the dependency on the PostgreSQL Helm Chart. This kind of chart dependency works for development teams that want to install the service and the database with a single command. Still, we want to decouple all the application infrastructural concerns from application services for larger scenarios. Luckily, the services Helm Charts allow us to turn off these component dependencies, allowing us to plug us Redis and PostgreSQL instances hosted and managed by different teams (figure 2.11).

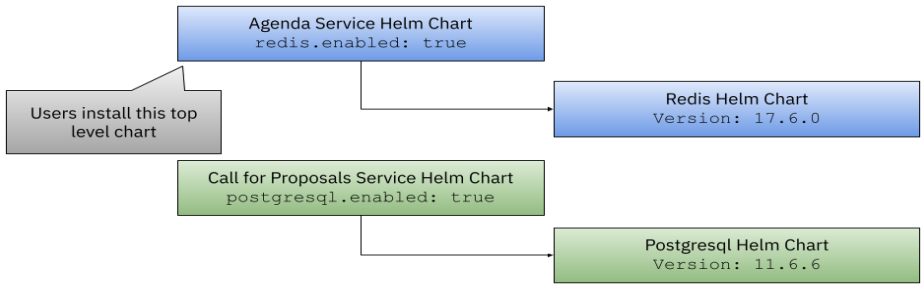


Figure 2.11 Using Helm Chart dependencies for application infrastructure.

By separating who installs and provisions these application's infrastructure components, we enable different teams to control and manage when these components are updated, backed up, or how they need to be restored in case of failure.

By using Crossplane, we can enable teams to request these databases on demand, that then can be connected to our application's services. One important aspect of the mechanisms we will use in the next sections is that the components we will request can be provisioned locally (using the Crossplane Helm provider) or remotely using Crossplane cloud providers. Let's look at what this would look like.

You can follow a step-by-step tutorial to install, configure, and create your Crossplane compositions at https://github.com/salaboy/from-monolith-to-k8s/tree/main/crossplane.

In this example, we will create a KinD cluster and configure Crossplane to allow teams to request databases on demand. More specifically, we are working with Redis and PostgreSQL instances. We want to do this in a way that teams can select where these instances are going to be provisioned, but we want to offer a unified experience for all the available options.

After having Crossplane and the Crossplane Helm Provider installed, we need to define two main things:

- *Crossplane composite resource definition*: This defines the resources that we want to expose to our teams. For this example, it is called Database. This composite resource definition defines an interface that multiple compositions can implement.
- *Crossplane composition*: the Crossplane composition allows us to group a set of resources. We can link a composition to a composite resource definition. By doing so, when the user requests new resources from the composite defined resource, all the composed resources will be created. I know it sounds confusing, so let's see it in action.

Let's look at the Database Crossplane composite resource definition:

```
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
  name: databases.salaboy.com
spec:
  group: salaboy.com
  names:
    kind: Database
    plural: databases
    shortNames:
      - "db"
      - "dbs"
  versions:
  - additionalPrinterColumns:
    - jsonPath: .spec.parameters.size
      name: SIZE
      type: string
    - jsonPath: .spec.parameters.mockData
      name: MOCKDATA
      type: boolean
    - jsonPath: .spec.compositionSelector.matchLabels.kind
      name: KIND
      type: string
```

As with every Kubernetes Resource, the CompositeResourceDefinition needs a unique name.

This CompositeResourceDefinition defines a new type of resource that needs to have a group and a kind.

Our new resource type that users can request is called Database, because we want to enable it to request new databases.

```
    name: v1alpha1
    served: true
    referenceable: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              parameters:
                type: object
                properties:
                  size:
                    type: string
                  mockData:
                    type: boolean
                required:
                - size
            required:
            - parameters
```

The new resource we are defining can also define custom parameters. For this example, and only for demonstration purposes, we are defining only two: "size" and "mockData".

Because the Kubernetes API server can validate all resources, we can define the required parameters. If these parameters are not provided, the Kubernetes API server will reject our resource request..

We have defined a new type of resource called a Database which contains two parameters that we can set, `size` and `mockData`. By setting up the `size` parameter, users can define how many resources are allocated for that instance. Instead of worrying about the specifics of how much storage they will need or how many replicas they need for the database instances, they can simply specify a size from a list of possible values (small, medium, or large). Using the `mockData` parameters, you can implement a mechanism to inject data into the instance when needed. This is just an example of what can be done, but it is up to you to define these interfaces and what parameters make sense to your teams. Let's look at what the Crossplane composition looks like:

```
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: keyvalue.db.local.salaboy.com
  labels:
    type: dev
    provider: local
    kind: keyvalue
spec:
  writeConnectionSecretsToNamespace: crossplane-system
  compositeTypeRef:
    apiVersion: salaboy.com/v1alpha1
    kind: Database
  resources:
    - name: redis-helm-release
      base:
        apiVersion: helm.crossplane.io/v1beta1
        kind: Release
        metadata:
          annotations:
            crossplane.io/external-name: # patched
        spec:
```

The composition resource also need a unique name.

For each composition we can also define labels. We will be using these then to match Compositions with the requested Database resources.

Using the "compositeTypeRef" property we link the Database CompositeResourceDefinition to this composition.

Inside the "resources" array, we can define all the resources this composition will provision. It is quite common to have more than one resource here. For this example, we are configuring a single resource of type Release defined in the Crossplane Helm provider.

```
    rollbackLimit: 3
    forProvider:
      namespace: default
      chart:
        name: redis
        repository: https://charts.bitnami.com/bitnami
        version: "17.8.0"
      values:
        architecture: standalone
    providerConfigRef:
      name: default
patches:
  - fromFieldPath: metadata.name
    toFieldPath: metadata.annotations[crossplane.io/external-name]
    policy:
      fromFieldPath: Required
  - fromFieldPath: metadata.name
    toFieldPath: metadata.name
    transforms:
      - type: string
        string:
          fmt: "%s-redis"
readinessChecks:
- type: MatchString
  fieldPath: status.atProvider.state
  matchString: deployed
```

We need to provide the parameters defined for the release resource, in this case, the Helm Chart details that we want to install using the Crossplane Helm provider. As you can see, we are pointing to the Redis Helm Chart hosted by Bitnami.

Using the "providerConfigRef", we can target different Crossplane Helm Provider configurations. This means that we can have different Helm providers pointing to different target clusters, and this composition can select which one to use. For the sake of simplicity, this composition uses the default configuration for the local Helm provider installation.

Because we are wiring multiple resources together, we can patch resources to configure them to work together. Check the Crossplane documentation for more details on what can be achieved with these mechanisms.

For each composition, we can define a condition to flag the resource status. For this example, we will mark the composition as ready when the Helm release resource "status.atProvider.state" property is set to "deployed". If you are provisioning cloud resources or multiple resources, you as the person defining the composition will need to define this condition.

With this composition, we link our `Database` resources with a set of resources, in this case, installing the Redis Helm Chart using the Default Helm provider we installed with Crossplane in our Kubernetes cluster (figure 2.12).

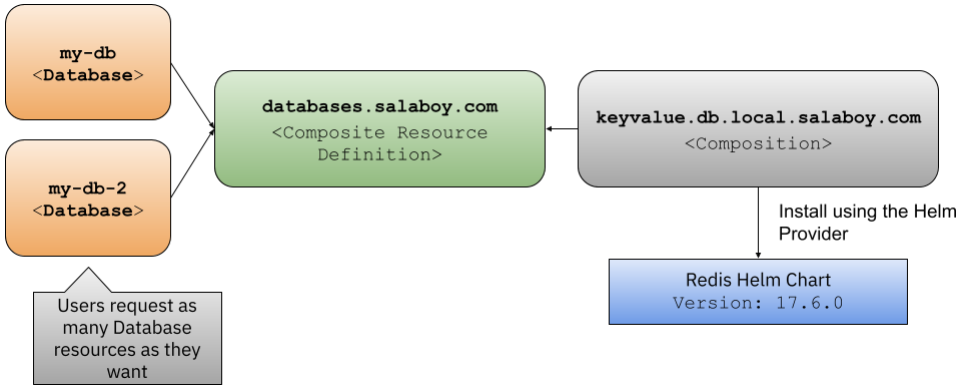


Figure 2.12 **Crossplane composition and composite resource definition working together.**

It is important to notice that this Helm Chart will be installed in the same Kubernetes cluster where Crossplane is installed. Still, nothing stops us from configuring the Helm provider to have the right credentials to install charts in a completely different cluster—see more on this in chapter 3.

In the step-by-step tutorial (https://github.com/salaboy/from-monolith-to-k8s/tree/main/crossplane), you will install the composite resource definition and the composition. Once these resources are installed, as shown in figure 2.12, you will be able to request new database resources, and for every resource all the resources defined in the composition will be provisioned. For the sake of simplicity, this composition just installs Redis, but there are no limits on how many resources you can create (except for the available hardware or quotas that you have).

A `Database` resource is just another Kubernetes resource that now our cluster understands, and it looks like this:

```
apiVersion: salaboy.com/v1alpha1
kind: Database
metadata:
  name: my-db-keyavalue
spec:
  compositionSelector:
    matchLabels:
      provider: local
      type: dev
      kind: keyvalue
  parameters:
    size: small
    mockData: false
```

This is the unique name for the resource.

We use matchLabels to select the appropriate composition.

We need to set the parameters that are required by our database resource..

Notice that the `spec.compositionSelector.matchLabels` matches with the labels used for the composition. We can use this mechanism to select a different composition for the same database definition.

If you are following the step-by-step tutorial, try to create multiple resources and look at the Crossplane official documentation to understand how to implement parameters like `small` or `mockData`, because these values are not being used and only serve for demonstration purposes.

The real power of these mechanisms comes when you have different compositions (implementations) for the same interface (composite resource definition). For example, we can now create another composition to provision PostgreSQL instances for the Call for Proposals service. The PostgreSQL composition will look the same as the one for Redis, but it will of course, install the PostgreSQL helm chart instead:

```
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: sql.db.local.salaboy.com
  labels:
    type: dev
    provider: local
    kind: sql
spec:
  ...
  compositeTypeRef:
    apiVersion: salaboy.com/v1alpha1
```

We need a unique name for our composition, so we can differentiate it from the keyvalue composition that we used for Redis.

We use a different label to describe this composition, notice that we the provider is the same as before.

```
    kind: Database
  resources:
    - name: postgresql-helm-release
      base:
        apiVersion: helm.crossplane.io/v1beta1
        kind: Release
        spec:
          forProvider:
            chart:
              name: postgresql
              repository: https://charts.bitnami.com/bitnami
              version: "12.2.7"
          providerConfigRef:
            name: default
          …
```

We want to install the PostgreSQL Helm Chart hosted by Bitnami.

Let's look at how to create a PostgreSQL instance using this composition. Creating a PostgreSQL instance will look pretty similar to what we did before for Redis:

```
apiVersion: salaboy.com/v1alpha1
kind: Database
metadata:
  name: my-db-sql
spec:
  compositionSelector:
    matchLabels:
      provider: local
      type: dev
      kind: sql
  parameters:
    size: small
    mockData: false
```

This is the unique name used for the PostgreSQL database.

We use the "sql" label to match the previously defined composition.

We are just using labels to select which composition will be triggered for our database resource. Figure 2.13 shows these concepts in action. Notice how labels are used to select the right composition based on the "kind" label value.

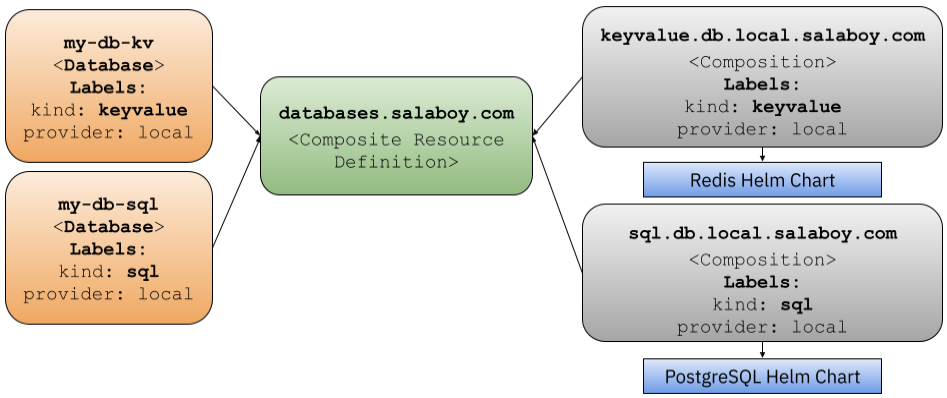


Figure 2.13 **Selecting compositions using labels.**

Hooray! We can create databases! But of course, this doesn't stop here. If you have access to a cloud provider, you can provide compositions that create database instances inside the cloud provider, and this is where Crossplane shines.

In the step-by-step tutorial directory, you will find under the databases/ directory two more compositions targeting the Google Cloud Platform services `CloudMemorystore-Instance` for Redis and `CloudSQLInstance` for PostgreSQL. Same as before, we will use different labels to match these compositions.

You can find the composition for Redis at https://github.com/salaboy/from-monolith-to-k8s/blob/main/crossplane/databases/app-database-redis-gke.yaml and the composition for PostgreSQL at https://github.com/salaboy/from-monolith-to-k8s/blob/main/crossplane/databases/app-database-postgresql-gke.yaml). You will notice that these compositions are simple and just create one resource each.

To get these compositions working (figure 2.14), you will need to install the Crossplane GCP (Google Cloud Platform) provider and configure it accordingly, as explained at https://github.com/salaboy/from-monolith-to-k8s/blob/main/crossplane/prerequisites.md#working-with-cloud-providers-example-gcp.

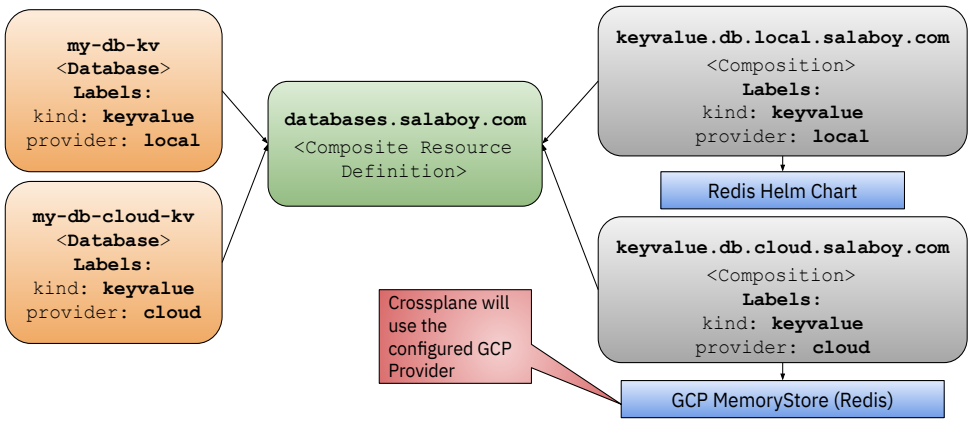


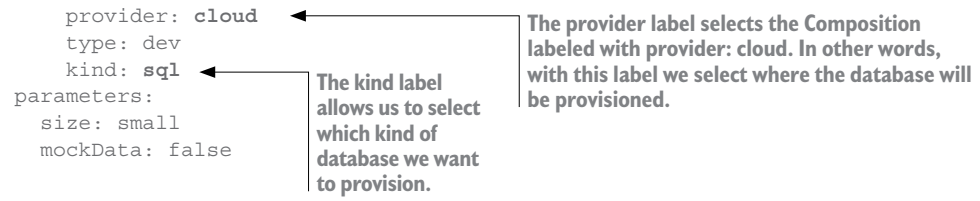
Figure 2.14 **Selecting compositions using different providers, still using labels.**

We can still select different providers by matching labels with the composition we want. By changing a label in figure 2.14, we can use the local Helm provider or the GCP provider to instantiate a Redis instance.

Then creating new database resources that will be provisioned in the Google Cloud Platform will look like this:

```
apiVersion: salaboy.com/v1alpha1
kind: Database
metadata:
  name: my-db-cloud-sql
spec:
  compositionSelector:
    matchLabels:
```

The unique name for our resource needs to be different from all the ones used before.



The beauty of all these mechanisms is that no matter where our databases are, we can connect our application's services just by pointing at them. Let's do that in the following section.

2.3.1 Connecting our services with the new provisioned infrastructure

When we create new database resources, Crossplane will monitor the status of these Kubernetes resources against the status of the provisioned components inside the specific cloud provider, keeping them in sync and making sure that the desired configurations are applied. This means that Crossplane will make sure that our databases are up and running, if for some reason that changes, Crossplane will try to reapply the configurations that we requested until what we requested is up and running.

If we don't have the application deployed in our KinD cluster, we can deploy it without installing PostgreSQL and Redis. As we have seen before, this can be disabled by setting two flags:

```
> helm install conference fmtok8s/fmtok8s-conference-chart --set agenda-
service.redis.enabled=false --set cp4-service.postgresql.enabled=false
```

I strongly recommend you to check out the step-by-step tutorial that you can find at https://github.com/salaboy/from-monolith-to-k8s/tree/main/crossplane to get your hands dirty with Crossplane and the Conference application. The best way to learn is by doing!

If we just run this command, no components (Redis and PostgreSQL) will be provisioned by Helm. Still, the application's services will not know where to connect to the Redis and PostgreSQL instances that we created using our Crossplane compositions. So, we need to add more parameters to the charts, so they know where to connect. First, check which Databases you have available in your cluster:

```
> kubectl get dbs
NAME             SIZE    MOCKDATA   KIND       SYNCED   READY   COMPOSITION                    AGE
my-db-keyavalue  small   false      keyvalue   True     True    keyvalue.db.local.salaboy.com  2m
my-db-sql        small   false      sql        True     True    sql.db.local.salaboy.com       5s
```

The following Kubernetes Pods back these database instances:

```
> kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
my-db-keyavalue-redis-master-0      1/1     Running   0          3m40s
my-db-sql-postgresql-0              1/1     Running   0          104s
```

Along with the pods, four Kubernetes secrets were created. Two to store the Helm releases used by our Crossplane compositions and two containing our new databases passwords that our applications will need to use to connect:

```
> kubectl get secret
NAME                                    TYPE                 DATA   AGE
my-db-keyavalue-redis                   Opaque               1      2m32s
my-db-sql-postgresql                    Opaque               1      36s
sh.helm.release.v1.my-db-keyavalue.v1   helm.sh/release.v1   1      2m32s
sh.helm.release.v1.my-db-sql.v1         helm.sh/release.v1   1      36s
```

Take a look at the services available in the default namespace after we provisioned our databases:

```
> kubectl get services
```

With the database service names and secrets we can configure our conference application chart to not only not deploy Redis and PostgreSQL but to connect to the right instances by running the following command:

```
> helm install conference fmtok8s/fmtok8s-conference-chart -f app-values.yaml
```

Instead of setting all the parameters in the command, we can use a file for the values to be applied to the chart, for this example, the app-values.yaml file looks like this:

```
fmtok8s-agenda-service:
  redis:
    enabled: false
  env:
    - name: SPRING_REDIS_HOST
      value: my-db-keyavalue-redis-master
    - name: SPRING_REDIS_PORT
      value: "6379"
    - name: SPRING_REDIS_PASSWORD
      valueFrom:
        secretKeyRef:
          name: my-db-keyavalue-redis
          key: redis-password #D

fmtok8s-c4p-service:
  postgresql:
    enabled: false
  env:
  - name: DB_ENDPOINT
    value: my-db-sql-postgresql
  - name: DB_PORT
    value: "5432"
  - name: SPRING_R2DBC_PASSWORD
    valueFrom:
      secretKeyRef:
        name: my-db-sql-postgresql
        key: postgres-password
```

We are disabling the Redis dependency from the Agenda Service Helm Chart.

We need to point to the Kubernetes service that points to our Redis instance.

To connect to the Redis instance, we use a secret that was created when the Helm Chart was installed. The name of the Kubernetes secret is "my-db-keyavalue-redis".

Inside the Kubernetes secret, there is a data.key property that contains the password.

In this app-values.yaml file, we are not only turning off the Helm dependencies for PostgreSQL and Redis, but we are also configuring the variables needed for the services to connect to our newly provisioned databases. The `SPRING_REDIS_HOST` and `DB_ENDPOINT` variables values are using the Kubernetes service names created when

we requested our database resources by the installed Helm Charts. Notice that if the databases were created in a different namespace, the `SPRING_REDIS_HOST` and `DB_ENDPOINT` variables values must include the fully qualified name of the service, which includes the namespace, for example, `my-db-sql-postgresql.default.svc.cluster.local` (where `default` is the namespace). The `SPRING_REDIS_PASSWORD` and `SPRING_R2DBC_PASSWORD` reference the Kubernetes secrets created specifically for these new instances (figure 2.15). Depending on your applications, and the frameworks that you are using, these environment variables names will be different, and we will look into this topic later on in the developer experience chapters.

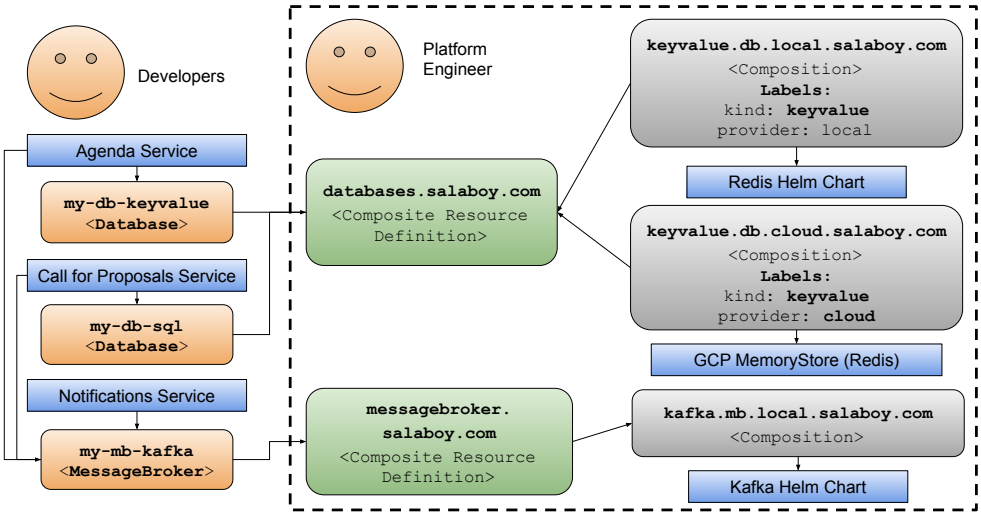


Figure 2.15 **Enabling different teams to work together and focus on their tasks at hand.**

All this effort enables us to split the responsibility of defining, configuring, and running all the application infrastructure to another team that is not responsible for working on the application's services. Services can be released independently without worrying about which databases are being used or when they need to be upgraded. Developers shouldn't be worrying about cloud provider accounts or whether they have access to create different resources, hence another team with a completely different set of skill sets can take care of creating Crossplane compositions and configuring Crossplane providers.

We have also enabled teams to request application infrastructure components by using Kubernetes resources. This enables them to create their own setups for experimentation and testing or to quickly set up new instances of the application. This is a major shift in how we (as developers) were used to doing things, because before cloud providers and in most companies today, to have access to a database, you need to use a ticketing system to request another team to provision that resource for you, and that can take weeks! To summarize what we have achieved so far, we can say that:

- We abstracted how to provision local- and cloud-specific components such as PostgreSQL and Redis databases and all the configurations needed to access the new instances.
- We have exposed a simplified interface for the application teams, which is cloud-provider independent because it relies on the Kubernetes API.
- Finally, we have connected our application service to the newly provisioned instance by relying on a Kubernetes secret that is created by Crossplane, containing all the details required to connect to the newly created instance.

You can follow a step-by-step tutorial that covers all the steps described in this section at https://github.com/salaboy/from-monolith-to-k8s/tree/main/crossplane.

If you use mechanisms like Crossplane compositions to create higher-level abstractions, you will be creating your domain-specific concepts that your teams can consume using a self-service approach. We created our database concept by creating a Crossplane composite resource that uses Crossplane compositions that knows which resources to provision (and in which cloud provider).

But we need to be cautious, we cannot expect every developer to understand or be willing to use tools like the ones we have discussed (Crossplane, Argo CD, Tekton, etc.). Hence, we need a way to reduce all the complexity these tools introduce. In the following chapters, we will be bringing all these tools together using some of the concepts around platform engineering mentioned earlier.

Let's jump into the next chapter, and let's build platforms on top of Kubernetes.

Summary

- Cloud-native applications depend on application infrastructure to run, because each service might require different persistent storages, a message broker to send messages, and other components to work.
- Creating application infrastructure inside cloud providers is easy and can save us a lot of time, but then we rely on their tools and ecosystem.
- Provisioning infrastructure in a cloud-agnostic way can be achieved by relying on the Kubernetes API and tools like Crossplane, which abstracts the underlying cloud provider and let us define which resources need to be provisioned using Crossplane compositions.
- Crossplane provides support for major cloud providers. It can be extended for other service providers, including third-party tools that might not be running on cloud providers (for example, legacy systems that we want to manage using the Kubernetes APIs).
- By using Crossplane composite resource definitions, we create an interface that application teams can use to request cloud resources using a self-service approach.

3 Let's build a platform on top of Kubernetes

This chapter covers

- Listing the main features our platform provides on top of Kubernetes
- Explaining the challenges with multi-cluster and multi-tenant setups and how to define our platform architecture
- Understanding what a platform on Kubernetes looks like

So far, we have looked at what platform engineering is, why we need to think about platforms in the context of Kubernetes, and how teams must choose the tools they can use from the CNCF landscape. Then we jumped into figuring out how our applications would run on top of Kubernetes, and how we are going to build, package, deploy, and connect these applications to other services that they need to work. This chapter will look into putting all the pieces together to create a walking skeleton for our platform. We will use some of the open-source projects introduced in the previous chapters and new tools to solve some of the challenges we will face when creating the first iteration of our platform. This chapter is divided into three main sections:

- The importance of the platform APIs.
- Kubernetes platform architecture: Multi-tenancy and multi-cluster challenges and how we can architect a scalable platform.

- Introducing our platform walking skeleton: We need to start somewhere; let's build a platform on top of Kubernetes.

Let's start by considering why defining the Platform APIs is the first step to platform building.

3.1 The importance of platform APIs

Earlier, we looked at existing platforms, such as Google Cloud Platform, to understand what key features they offer to teams that are building and running applications for the cloud. We now need to compare this to the platforms that we are building on top of Kubernetes, because these platforms share some common goals and features with cloud providers, while at the same time being closer to our organizations' domains.

Platforms are nothing other than software that we will design, create, and maintain. As with any good software, our platform will evolve to help teams with new scenarios, make our teams more efficient by providing automation, and give us the tools to make the business more successful. As with any other software, we start by looking at the platform APIs, which will not only provide us with a scope that is manageable to start with but will also define the contracts and behaviors that our platform provides to its users.

Our platform APIs are important, because good APIs can simplify the life of development teams wanting to consume services from our platform. If our platform APIs are well-designed, more tailored tools like CLIs and SDKs can be created to assist users in consuming our platform services.

If we build a custom and more domain-specific API for our platform, we can start by tackling one problem at a time and then expand these APIs/interfaces to cover more and more workflows, even for different teams.

Once we understand which workflows we want to cover and have an initial platform API, dashboards and more tooling can be created to help teams to adopt the platform itself.

Let's use an example to make it more concrete. I hope you can translate the example I am showing here into more concrete examples inside your organization. All the mechanisms should apply in the same way. Let's enable our development teams to request new development environments.

3.1.1 Requesting development environments

A common scenario where a platform can help teams get up to speed when they start working on new features is provisioning them with all they need to do their work. To achieve this task, the platform engineering team must understand what they will work on, what tools they need, and which other services must be available to succeed.

Once the platform engineering team has a clear idea of what a team requires, they can define an API enabling teams to create requests to the platform to provision new development environments. Behind this API, the platform will have the mechanisms to create, configure, and provide access to teams to all the resources created.

For our Conference Application example, if a development team is extending the application, we will need to ensure they have a running version of the application to work and test changes against. This isolated instance of the application will also need to have its databases and other infrastructural components required for the application to work. A more advanced use case includes loading the application with mock data to allow teams to test their changes with pre-populated data. The interactions between the application development team and the platform should look like the example in figure 3.1.

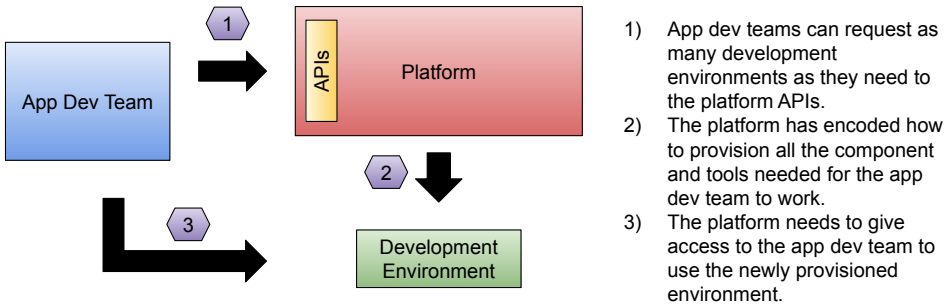


Figure 3.1 Application development team interactions with the platform.

As mentioned before, development environments are just an example. You must ask yourself what tools your teams need to do their work. A development environment might not be the best way to expose tools to a team of data scientists, because they might need other tools to gather and process data or train machine-learning models.

Implementing this simple flow in Kubernetes is not an easy task. To implement this scenario, we need to:

- Create new APIs that understand development environment requests.
- Have the mechanisms to encode what a development environment means for our teams.
- Have the mechanisms to provision and configure components and tools.
- Have the mechanisms to enable teams to connect to the newly provisioned environments.

There are several alternatives to implement this scenario, starting with creating our own custom Kubernetes extensions or using more specialized tools for development environments. But before diving into implementation details, let's define what our platform API would look like for this scenario.

As with object-oriented programming, our APIs are Interfaces that can be implemented by different classes, which provide concrete behavior. For provisioning development environments, we can define a very simple interface called "Environment". Development teams requesting a new development environment can create new

requests to the platform by creating new environment resources. The "Environment" interface represents a contract between the user and the platform. This contract can include parameters to define the type of environment the team is requesting or options and parameters they need to tune for their specific request (figure 3.2).

Figure 3.2 Environment resource defined by the platform API.

It is important to note that the Environment interface shouldn't include (or leak) any implementation detail about our environments. These resources serve as our abstraction layer to hide complexity from our platform users about how these environments will be provisioned. The simpler these resources are, the better for the platform users. In this example, the platform can use the Environment `type` parameter to decide which environment to create, and we can plug in new types as we evolve our platform mechanisms.

Once we recognize which interfaces we need, we can slowly add parameters that teams can configure. For our example, it might make sense to parameterize which services we want to deploy in our Environment if we also want the application infrastructure to be created or to connect our services to existing components. The possibilities here are endless, depending on what makes sense for your teams to parameterize. The platform team here can control what is possible and what is not. Expanding our Environment interface to cover more use cases can look like the example in figure 3.3.

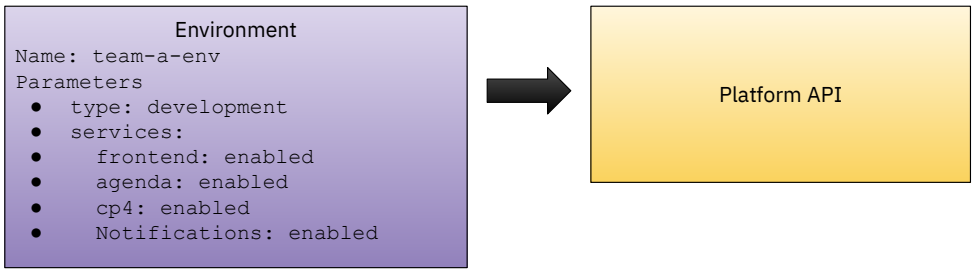


Figure 3.3 Extended Environment resource to enable/disable application's services.

Encoding this Environment resource into a format like JSON or YAML to implement the platform API is straightforward:

```
"environment": {
 "name": "team-a-env"
 "type":"development",
```

```
 "services":{
     "frontend": "enabled",
     "agenda": "enabled",
     "c4p": "enabled",
     "notifications": "disabled"
  }
}
```

Once the interface is defined, the next logical step is to provide one implementation to provision these environments for our platform users. Before jumping into implementation details, we need to cover two of the main challenges you will face when deciding where the mechanisms for implementing these environments will reside. We need to talk about something important: How will we architect our platform?

3.2 Platform architecture

On the technical side of building platforms, we will encounter challenges requiring the platform team to make some hard choices. In this section, we will talk about how we can architect a platform that allows us to encapsulate a set of tools behind our platform APIs and enable development teams to perform their tasks without worrying about which tools are being used by the platform to provision and wire up complex resources.

Because we are already using Kubernetes to deploy our workloads (for the Conference Application), it makes sense also to run the platform services on top of Kubernetes, right? But would you run the platform services and components right beside your workloads? Probably not. Let's step back a bit.

If your organization adopts Kubernetes, you will likely already work with multiple Kubernetes clusters. As we discussed in chapter 1 for environment pipelines, your organization probably has a production environment, staging, or QA environment already in place. If you want your application development teams to work on environments that feel like the production environment, you will need to enable them with Kubernetes clusters (figure 3.4).

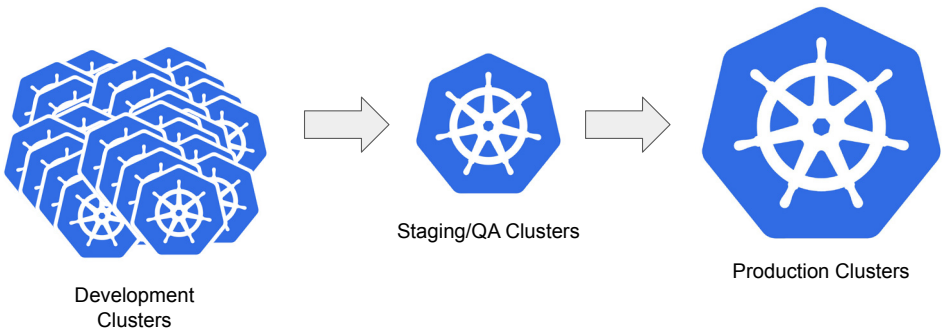


Figure 3.4 Environment clusters, do you want to enable developers to have their own environments?

While the Production Cluster(s) and Staging/QA Cluster(s) should be handled carefully and hardened to serve real-life traffic, development environments tend to be more ephemeral and sometimes even run on the development team laptops. One thing is certain, you don't want to be running any platform-related tools in any of these environments. The reason is simple: tools like Crossplane, Argo CD, or Tekton shouldn't be competing for resources with our application's workloads.

When looking at building platforms on top of Kubernetes, teams tend to create one or more special clusters to run platform-specific tools. The terms are not standardized yet but creating a platform or management cluster to install platform-wide tools is becoming increasingly popular (figure 3.5).

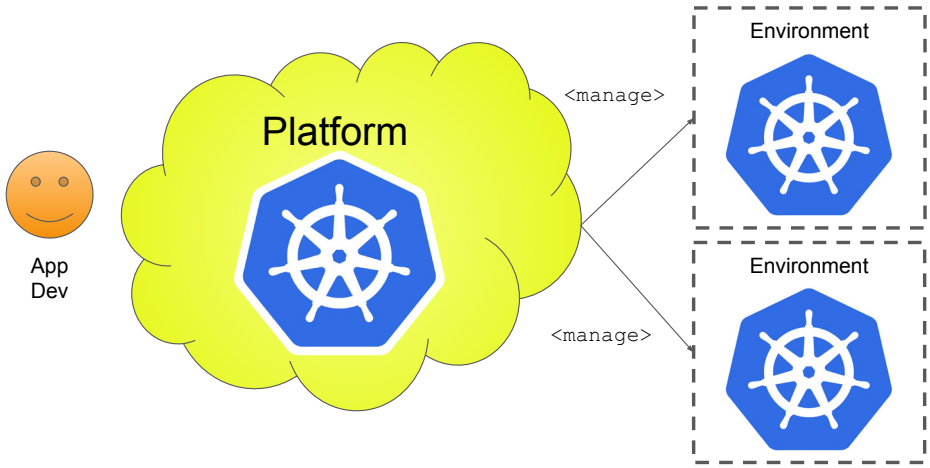


Figure 3.5 **Platform cluster with platform tools managing environments.**

By having separate platform cluster(s), you can install the tools that you need to implement your platform capabilities while at the same time build a set of management tools to control environments where your workloads run.

Now that you have a separate place to install these tools, you can also host the platform API on this cluster, once again, to not overload your workload clusters with platform components. Wouldn't it be great to reuse or extend the Kubernetes API to serve also as our platform APIs? There are pros and cons to this approach. For example, suppose we want our platform API to follow Kubernetes conventions and behaviors. In that case, our platform will use the declarative nature of Kubernetes and promote all the best practices followed by the Kubernetes APIs, such as versioning, the resource model, etc. For non-Kubernetes users, this API might be too complex, or the organization might follow other standards when creating new APIs that do not match the Kubernetes style. *If we reuse the Kubernetes APIs for our platform APIs, all the CNCF tools designed to work with these APIs will automatically work with our platform.* Our platform automatically

becomes part of the ecosystem. In the last couple of years, I've seen a trend around teams adopting the Kubernetes APIs as their platform APIs (figure 3.6).

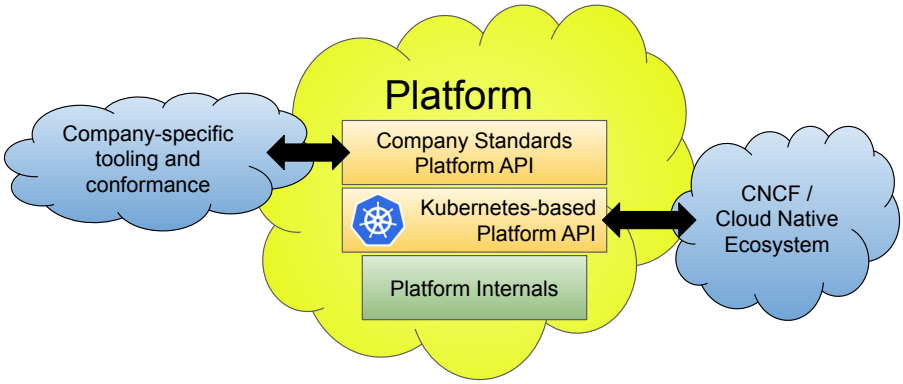


Figure 3.6 **Kubernetes-based platform APIs complemented by company-specific APIs.**

Adopting the Kubernetes APIs for your platform API doesn't stop you from building a layer on top for other tools to consume or to follow the company's standards. By having a Kubernetes-based API layer, you can access all the amazing tools that are being created in the CNCF and cloud-native space. On top of the Kubernetes-based APIs, another layer can follow company standards and conformance checks, enabling easier integrations with other existing systems.

Following our previous example, we can extend Kubernetes to understand our environment requests and provide the mechanisms to define how these environments will be provisioned (figure 3.7).

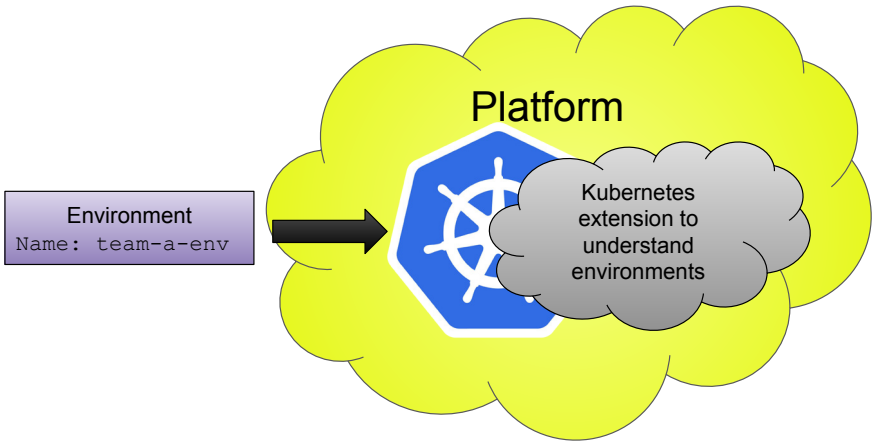


Figure 3.7 **Extending Kubernetes to understand environments and serve as our platform APIs.**

In principle, this looks good and doable. Still, before implementing these Kubernetes extensions to serve our platform API and central hub of platform tooling, we need to understand the questions that our platform implementation will try to answer. Let's take a look at the main platform challenges that teams in these scenarios will face.

3.2.1 Platform challenges

Sooner or later, if you are dealing with multiple Kubernetes clusters, you will need to manage them, and all the resources related to these clusters. What does it take to manage all these resources? The first step to start understanding the underlying problems is to understand who the users of our platforms are. Are we building a platform for external customers or internal teams? What are their needs and the level of isolation that they need to operate autonomously without bothering their neighbors? What guardrails do they need to be successful?

While I cannot answer these questions for all use cases, one thing is clear, platform tools and workloads need to be separated. We need to encode in our platform our tenant boundaries based on each tenant's expectations. No matter if these tenants are customers or internal teams. We must set clear expectations for our platform users about our tenancy model and guarantees, so they understand the limitations of the resources the platform gives them to do their work.

One thing is clear, the platform that we are going to build needs to take care of encoding all these decisions. Let's start by looking at one of the first challenges you will encounter when trying to manage multiple clusters.

3.2.2 Managing more than one cluster

The platform we will build needs to manage and understand which environments are available for teams to use. More importantly, it should enable the team to request their own environments whenever needed.

Using the Kubernetes APIs as our platform API to request environments, we can use tools like Argo CD (covered in chapter 1) to persist and sync our environment configurations to live Kubernetes clusters. Managing our clusters and environments becomes just managing Kubernetes resources that must be synced to our platform cluster(s) (figure 3.8).

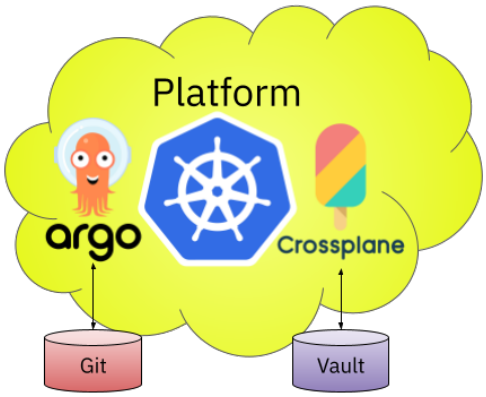


Figure 3.8 Combining GitOps and Crossplane for managing environments and clusters.

By combining tools like Argo CD and Crossplane inside our platform clusters, we promote using the same techniques that we already discussed in chapter 1 for environment pipelines, which sync application-level components, but now we use them to manage high-level platform concerns.

As you can see in the previous figure, our platform configuration itself will become more complex, because it will need to have its source of truth (Git repository) to store the environment and resources that the platform is managing. It will also need to have access to a secret store such as HashiCorp Vault to enable Crossplane to connect and create resources in different cloud providers. In other words, you now have two extra concerns. You will need to define, configure, and give access to one or more Git repositories to contain the configurations for the resources created in the platform. You must manage a set of cloud provider accounts and their credentials so the platform cluster(s) can access and use these accounts.

While the example in section 3.3 doesn't go in-depth on configuring all these concerns, it provides a nice playground to build on top of and experiment with more advanced setups depending on your teams' requirements.

My recommendation to prioritize which configurations make sense first is to understand what your teams or tenants will do with the resources and their expectations and requirements. Let's dig a bit more into that space.

3.2.3 Isolation and multi-tenancy

Depending on your tenants' (teams, internal or external customers) requirements, you might need to create different isolation levels, so they don't disturb each other when working under the same platform roof.

Multi-tenancy is a complicated topic in the Kubernetes community. Using Kubernetes RBAC (role-based access control), Kubernetes Namespaces, and multiple Kubernetes controllers that might have been designed with different tenancy models in mind, makes it hard to define isolation levels between different tenants inside the same cluster.

Companies embarking on adopting Kubernetes tend to take one of the following approaches for isolation:

- Kubernetes Namespaces:
  - Pros:
    - Creating namespaces is very easy, and it has almost zero overhead.
    - Creating namespaces is cheap, because it is just a logical boundary that Kubernetes uses to separate resources inside the cluster.
  - Cons:
    - Isolation between namespaces is very basic, and it will require RBAC roles to limit users' visibility outside the namespaces they have been assigned. Resource quotas must also be defined to ensure that a single namespace is not consuming all the cluster resources.

- Providing access to a single namespace requires sharing access to the same Kubernetes APIs endpoints that admins and all the other tenants are using. This limits the operations clients can execute on the cluster, such as installing cluster-wide resources.

- Kubernetes clusters:
  - Pros:
    - Users interacting with different clusters can have full admin capabilities, enabling them to install as many tools as they need.
    - You have full isolation between clusters, and tenants connecting to different clusters will not share the same Kubernetes API server endpoints. Hence, each cluster can have different configurations for how scalable and resilient they are. This allows you to define different tenants' categories based on their requirements.
  - Cons:
    - This approach is expensive, because you will pay for computing resources to run Kubernetes. The more clusters you create, the more you will spend running Kubernetes.
    - Managing multiple Kubernetes clusters becomes complex if you enable teams to create (or request) their own. Zombie clusters (clusters nobody uses and have abandoned) start to pop up, wasting valuable resources.
    - Sharing resources and installing and maintaining tools across a fleet of different Kubernetes clusters is challenging and a full-time job

Based on my experience, teams will create isolated Kubernetes clusters for sensitive environments such as production environments and performance testing. These sensitive environments tend not to change and are only managed by operation teams. When you shift towards development teams and more ephemeral environments for testing or day-to-day development tasks, using a big cluster with namespaces is a common practice.

Choosing between these two options is hard, but what is important is not to over-commit to just a single option. Different teams might have different requirements, so in the next section, we will look at how the platform can abstract these decisions, enabling teams to access different setups depending on their needs.

3.3 Our platform walking skeleton

This section looks into creating a very simple platform that allows internal teams to create development environments. Because our teams are deploying the conference application to Kubernetes clusters, we want to offer them the same developer experience.

You can follow a step-by-step tutorial, where you will install and interact with the Platform walking skeleton at https://github.com/salaboy/from-monolith-to-k8s/tree/main/platform/prototype.

To achieve this, we will use some tools that we used before, like Crossplane, to extend Kubernetes to understand development environments (figure 3.9). Then we will use a project called `vcluster` (https://vcluster.com) to provision small Kubernetes clusters for our teams. These clusters are isolated, allowing teams to install extra tools without worrying about what other teams are doing. Because teams will have access to the Kubernetes APIs, they can do whatever they need with the cluster without requesting complicated permissions to debug their workloads.

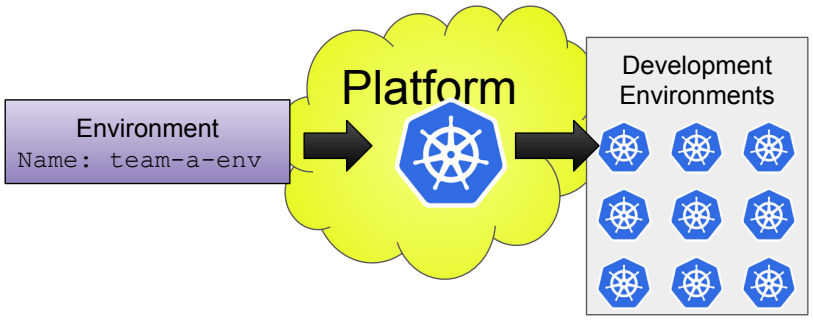


Figure 3.9 Building a platform prototype to provision development environments.

We will keep things simple for the walking skeleton, but the platforms are complicated.

I can't stress enough the importance that this example, on purpose, is using existing tools instead of creating our custom Kubernetes extensions. If you create custom controllers to manage environments, you create a complex component that will require maintenance and probably overlaps 95% with the mechanisms shown in this example.

In the same way that we started this chapter talking about our platform APIs, let's look at how we can build these APIs without creating our custom Kubernetes extensions. We will use Crossplane compositions in the same way we did for our databases in chapter 2, but now we will implement our environment custom resource. We can keep the environment resource simple and use Kubernetes label matchers and selectors to match a resource with one of the possible compositions we can create to provision our environments (figure 3.10).

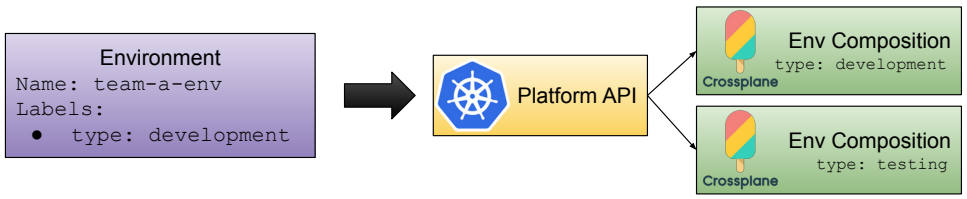


Figure 3.10 Mapping environment resources to Crossplane compositions.

Crossplane compositions offer the flexibility to use different providers to provision and configure resources together, and as we saw in chapter 2, multiple compositions (implementations) can be provided for different kinds of environments.

For this example, we want each environment to be isolated from the other to avoid teams unintentionally deleting others' team resources. The two most intuitive ways of creating isolated environments would be to create a new namespace per environment or a full-blown Kubernetes cluster for each environment (figure 3.11).

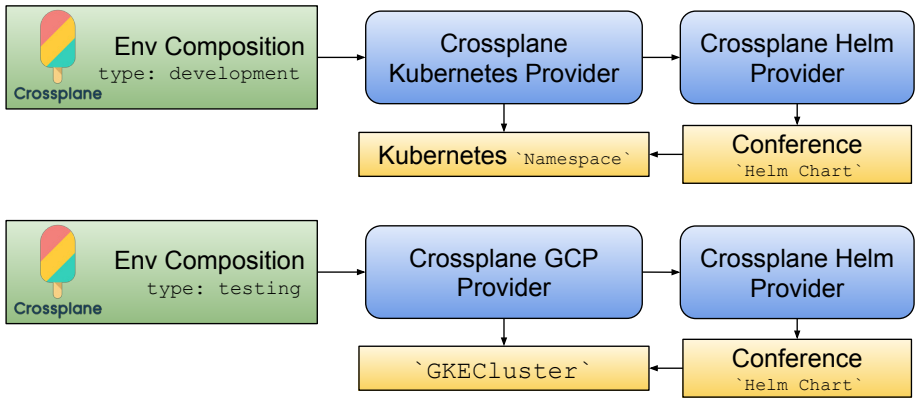


Figure 3.11 **Different environment compositions, namespace, and GKECluster.**

While creating a fully-fledged Kubernetes cluster might be overkill for every development team, a Kubernetes Namespace might not provide enough isolation for your use case, because all teams will interact with the same Kubernetes API server.

For this reason, we will use `vcluster` in conjunction with the Crossplane Helm Provider, which gives us the best of both worlds without the costs of creating new clusters (figure 3.12).

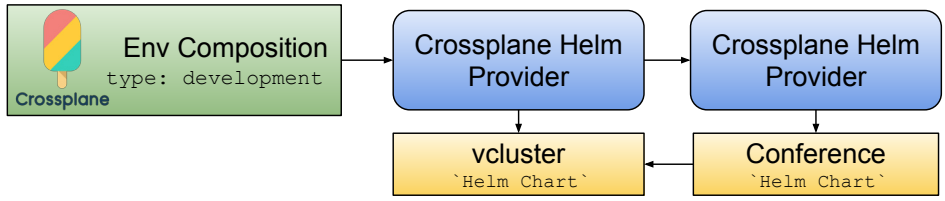


Figure 3.12 **Using "vcluster" to create isolated environments.**

You might be wondering: what is a `vcluster`? And why are we using the Crossplane Helm Provider to create one? While `vcluster` is just another option that you can use to build your platforms, I consider it a key tool for every platform engineer toolbox.

3.3.1 "vcluster" for virtual Kubernetes clusters

I am a big fan of the `vcluster` project. If you are discussing multi-tenancy on top of Kubernetes, `vcluster` tends to pop up in the conversation because it offers a really nice alternative to the Kubernetes Namespaces vs. Kubernetes clusters discussions.

`vcluster` focuses on providing Kubernetes API server isolation between different tenants by creating virtual clusters inside your existing Kubernetes cluster (host cluster), as shown in figure 3.13.

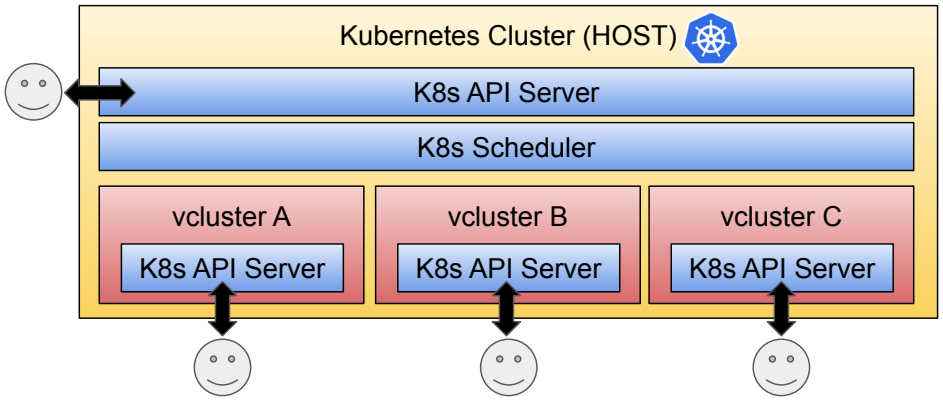


Figure 3.13 **"vcluster" provides isolation at the Kubernetes (K8s) API server.**

By creating new virtual clusters, we can share an isolated API server with tenants where they can do whatever they need without worrying about what other tenants are doing or installing. For scenarios where you want each tenant to have cluster-wide access and full control of the Kubernetes API Server, `vcluster` provides a simple alternative to implement this.

Creating a `vcluster` is easy, you can create a new `vcluster` installing the `vcluster` Helm Chart. Alternatively, you can use the "`vcluster`" CLI to create and connect to it.

Finally, a great table comparing `vcluster`, Kubernetes Namespaces, and Kubernetes clusters can be found in their documentation, so if you are already having these conversations with your teams, this table makes it crystal clear to explain the advantages and tradeoffs (figure 3.14).

	Separate Namespace For Each Tenant	vcluster	Separate Cluster For Each Tenant
Isolation	very weak	strong	very strong
Access For Tenants	very restricted	vcluster admin	cluster admin
Cost	very cheap	cheap	expensive
Resource Sharing	easy	easy	very hard
Overhead	very low	very low	very high

Figure 3.14 Kubernetes Namespaces vs vcluster vs Kubernetes cluster tenants pros and cons.

Let's see what our platform walking skeleton looks like for teams that want to create, connect and work against new environments that use `vcluster`.

3.3.2 Platform experience and next steps

The platform walking skeleton implemented in the GitHub repository: https://github.com/salaboy/from-monolith-to-k8s/tree/main/platform/prototype allows teams connected to the platform API to create new environment resources and submit a request for the platform to provision it for them (figure 3.15).

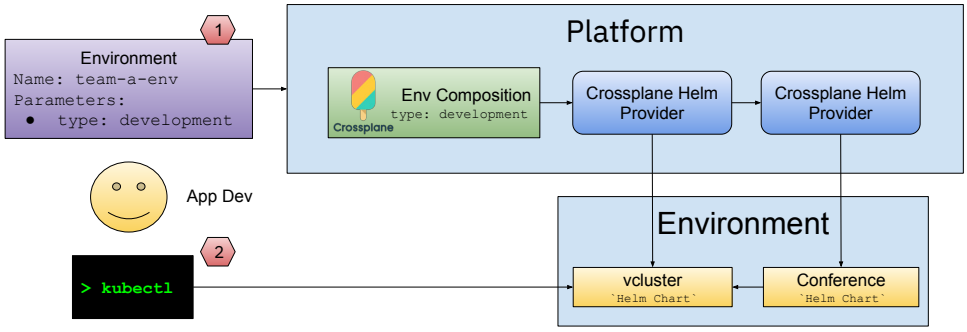


Figure 3.15 Using Crossplane and vcluster to create isolated environments for application development teams.

The platform cluster uses Crossplane and Crossplane compositions to define the provision of the environment. To run the composition in a local Kubernetes cluster (and not require access to a specific cloud provider), the walking skeleton uses `vcluster` to provision each environment on its own (virtual) Kubernetes cluster. Having separate

Kubernetes clusters enables teams to connect to these environments and do the work they need to do with our Conference Application, which is by default installed when the environment is created.

A natural extension to the walking skeleton would be to create the compositions to create environments spawning Kubernetes clusters on a cloud provider or managing the environment resources inside a Git repository, in contrast, to require application development teams to connect directly with the platform APIs.

Now that we have a simple walking skeleton of our platform, let's look at what we can do to make application development teams more efficient when working with these environments.

In the next couple of chapters, we will explore the capabilities the platform should provide when creating environments for application development teams. Different tools can implement these capabilities, but it is important to recognize what kind of functionalities should be offered to teams so they can be more efficient. Chapter 4 covers release strategies and why they are important to enable teams to experiment and release more software.

Summary

- Building platforms on top of Kubernetes is a complex task that involves combining different tools to serve teams with different requirements.
- Platforms are software projects as your business applications. Starting by understanding who the main users will be and defining clear APIs is the key to prioritizing tasks on how to build your platform.
- Managing multiple Kubernetes clusters and dealing with tenant isolations are two main challenges that platform teams face early on in the platform-building journey
- Having a platform walking skeleton can help you to demonstrate to internal teams what can be built to speed up their cloud-native journey
- Using tools like Crossplane, Argo CD, `vcluster`, and others can help you to promote cloud-native best practices at the platform level but, most importantly, avoid the urge to create your own custom tools and ways to provision and maintain complex configurations of cloud-native resources

4 Platform capabilities I: Enabling developers to experiment

This chapter covers

- Enabling teams to experiment and release more software
- Using deployments, services, and Ingress as release strategies
- Implementing rolling updates, canary releases, blue/green deployments, and A/B testing
- Using Kubernetes built-in resources for releasing software efficiently
- Reducing release risks and speeding up release cadence

If you and your teams are worried about deploying a new version of your service, you are doing something wrong. You are slowing down your release cadence due to the risk of making or deploying changes to your running applications.

This chapter focuses on one of the capabilities that good platforms should offer their users. If your platform doesn't enable teams with the right mechanisms to experiment by releasing software using different strategies, you are not fully using the power of Kubernetes and its building blocks.

Reducing risk and having the proper mechanisms to deploy new versions drastically improves confidence in the system. It also reduces the time from a requested change until it is live in front of your users. New releases with fixes and new features directly correlate to business value, because software is not valuable unless it serves our company's users.

While Kubernetes built-in resources such as deployments, services, and Ingresses provide us with the basic building blocks to deploy and expose our services to our users, a lot of manual and error-prone work must happen to implement well-known release strategies. This chapter is divided into two main sections:

- Release strategies using Kubernetes built-in mechanisms
  - Rolling updates
  - Canary releases
  - Blue/green deployments
  - A/B testing
  - Limitations and complexities of using Kubernetes built-in mechanisms
- Knative Serving: Advanced traffic management and release strategies
  - Introduction to Knative Serving
  - Advanced traffic-splitting features
  - Knative Serving applied to our Conference Application
- Argo rollouts: Release strategies automated with GitOps

In this chapter, you will learn about Kubernetes built-in mechanisms and Knative Serving to implement release strategies such as rolling updates, canary releases, blue-green deployments, and A/B testing. This chapter covers these patterns, understanding the implementation implications and limitations.

The second half of the chapter introduces Knative Serving, which allows us to implement more fine-grained control on how traffic is routed to our application's services. We explore how Knative Serving traffic-splitting mechanisms can be used to improve how we release new versions of our cloud-native application.

4.1 Kubernetes built-in mechanisms for releasing new versions

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

Deployments and StatefulSets are like cookie cutters; they define how the container(s) for our service(s) needs to be configured and, based on the replicas specified, will create that number of containers. We will need to create a service to route traffic to these containers.

Using a service for each deployment is standard practice, and it is enough to enable different services to talk to each other by using a well-known service name. But if we want to allow external users (outside our Kubernetes cluster) to interact with our services, we will need an Ingress resource (plus an Ingress controller). The Ingress resource will be in charge of configuring the networking infrastructure to enable external traffic into our Kubernetes cluster; this is usually done for a handful of services. In general, not every service is exposed to external users (figure 4.1).

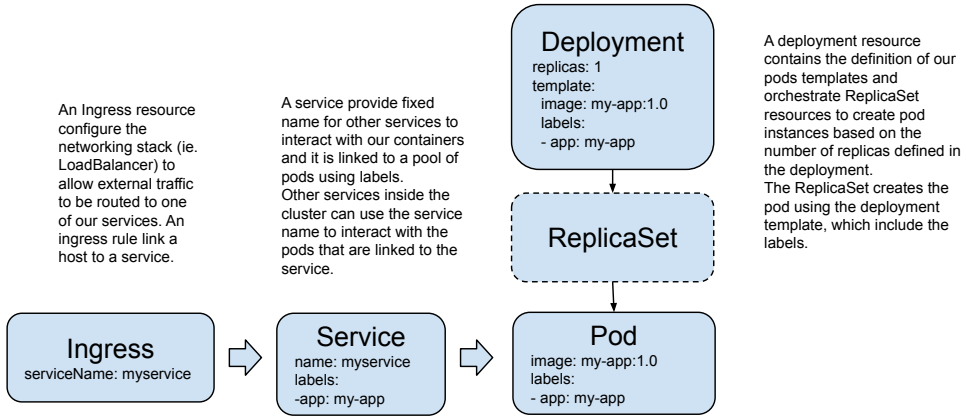


Figure 4.1 **Kubernetes built-in resources for routing traffic.**

Now, imagine what happens when you want to update the version of one of these external-facing services. Commonly, these services are user interfaces. And it will be pretty good to upgrade the service without downtime. The good news is that Kubernetes was designed with zero-downtime upgrades in mind, so both deployments and StatefulSets come equipped with a rolling update mechanism.

If you have multiple replicas of your application Pods, the Kubernetes service resource acts as a load balancer. This allows other services inside your cluster to use the service name without caring which replica they interact with (or which IP address each replica has) (figure 4.2).

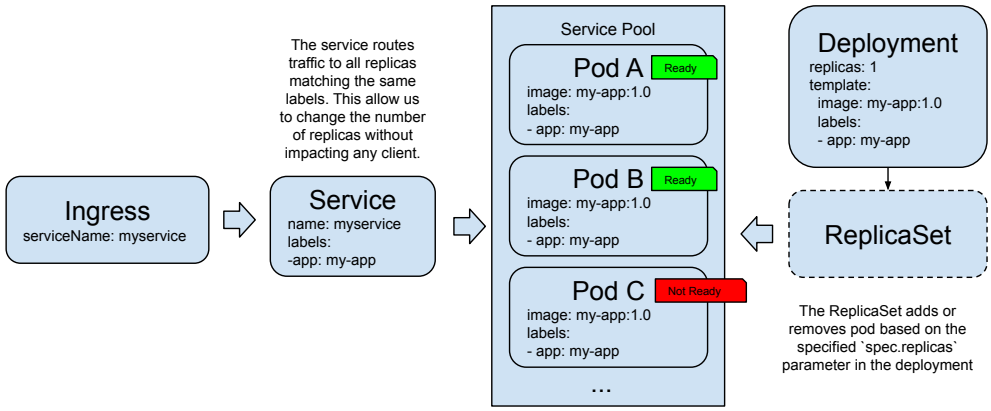


Figure 4.2 **Kubernetes service acting as a load balancer.**

To attach each of these replicas to the load balancer (Kubernetes service) pool, Kubernetes uses a probe called "Readiness Probe" (https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/) to make sure that the container running inside the pod is ready to accept incoming requests; in other

words, it has finished bootstrapping. In figure 4.2, Pod C is not ready yet, so 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 ensure we don't lose any incoming traffic while doing the update. We need to start a new replica using the `my-app:1.1` image and ensure that this new instance is up and running and ready to receive traffic before removing the old version. If we have multiple replicas, we probably don't want to start all the new replicas simultaneously, 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:1.0` replicas in one go. We need to guarantee that the new version is working and handling the load correctly before we shut down the previous version that was working fine. Luckily for us, Kubernetes automates all the starting and stopping of containers using a rolling update strategy (https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/).

4.2 Rolling updates

Deployments and StatefulSets come with these mechanisms built-in, and we need to understand how these mechanisms work to know when to rely on them and their limitations and challenges.

A rolling update consists of well-defined checks and actions to upgrade any number of replicas managed by a Deployment. Deployment resources orchestrate ReplicaSets to achieve rolling updates, and the following figure shows how this mechanism works (figure 4.3).

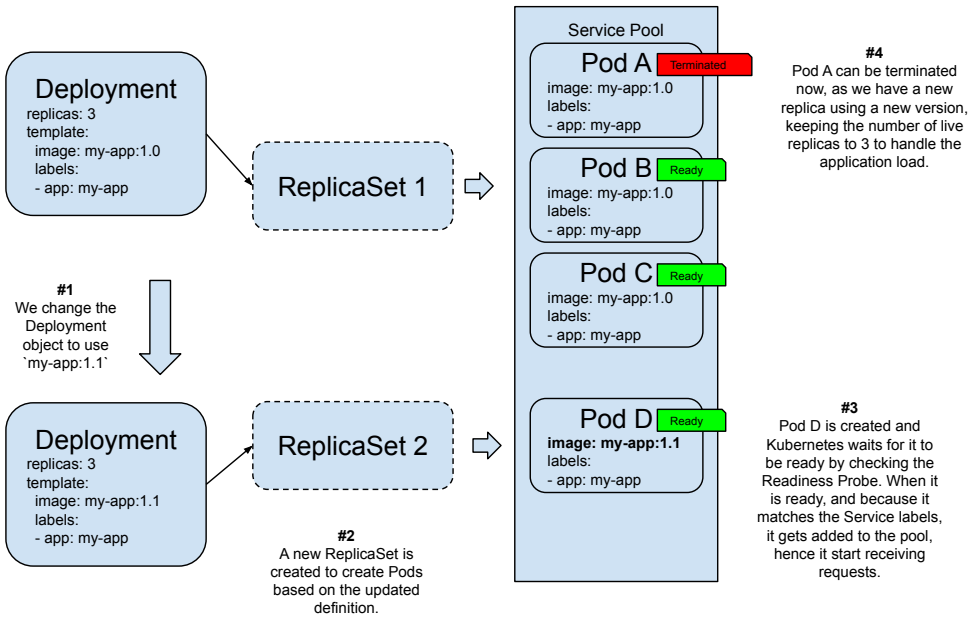


Figure 4.3 Kubernetes deployments rolling updates.

The rolling update mechanism kicks off by default whenever you update a deployment resource. A new ReplicaSet is created to handle the creation of Pods using the newly updated configuration defined in `spec.template`. This new ReplicaSet will not start all three replicas immediately, because this will cause a surge in resource consumption. Hence, it will create a single replica, validate that it is ready, attach it to the service pool, and terminate a replica with the old configuration.

By doing this, the deployment object guarantees three replicas active at all times handling clients' requests. Once Pod D is up and running, and Pod A is terminated, ReplicaSet 2 can create Pod E, wait for it to be ready, and then terminate Pod B. This process repeats until all pods in ReplicaSet 1 are drained and replaced with new versions managed by ReplicaSet 2. If you change the Deployment resource again, a new ReplicaSet (ReplicaSet 3) will be created, and the process will repeat again.

A benefit of rolling updates is that ReplicaSets contain all the information needed to create pods for a specific deployment configuration. If something goes wrong with the new container image (in this example, `my-app:1.1`) we can easily revert (roll back) to a previous version. You can configure Kubernetes to keep a certain number of revisions (changes in the Deployment configuration), so changes can be rolled back or forward.

Changing a deployment object will trigger the rolling update mechanism, and you can check some of the parameters that you can configure to the default behavior here (https://kubernetes.io/docs/concepts/workloads/controllers/deployment/). StatefulSets have a different behavior as the responsibility for each replica is related to the state that is handled. The default rolling update mechanism works a bit differently. You can find the differences and a detailed explanation of how this works here (https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/).

Check out the following commands to review the deployment revisions history and do rollbacks to previous versions (https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#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 many online and interactive examples and tutorials where you can see this in action.

4.2.1 Canary releases

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

Canary releases (https://codefresh.io/learn/software-deployment/what-are-canary-deployments/) 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. Kubernetes will not check that the latest versions perform the same or better than the previous, so we can introduce issues to our applications. More control over how these updates are done is needed.

If we start with a deployment configured to use a Docker image called `my-app:1.0`, have two replicas, and label it with `app: myapp`, a service will route traffic as soon as we use the selector `app:myapp`. Kubernetes will be in charge of matching the service selectors to our pods. In this scenario, 100% of the traffic will be routed to both replicas using the same Docker image (`my-app:1.0`) shown in figure 4.4.

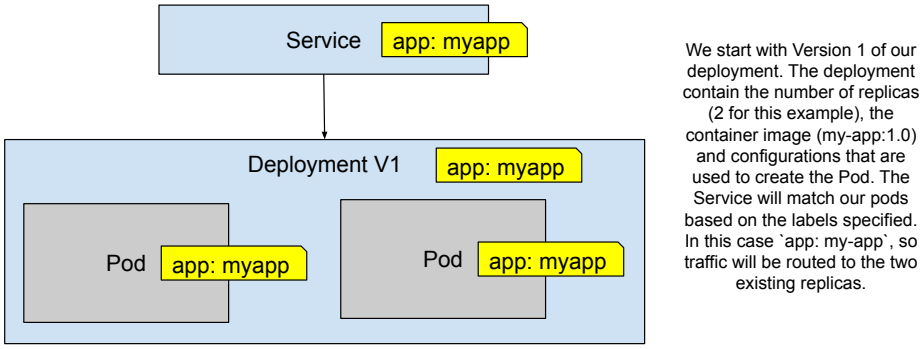


Figure 4.4 Kubernetes service routes traffic to two replicas by matching labels.

Imagine changing the configuration or having a new Docker image version (maybe `my-app:1.1`). We don't want to automatically migrate our Deployment V1 to the new version of our Docker image. Alternatively, we can create a second Deployment (V2) resource and use the service `` `selector` `` to route traffic to the new version (figure 4.5).

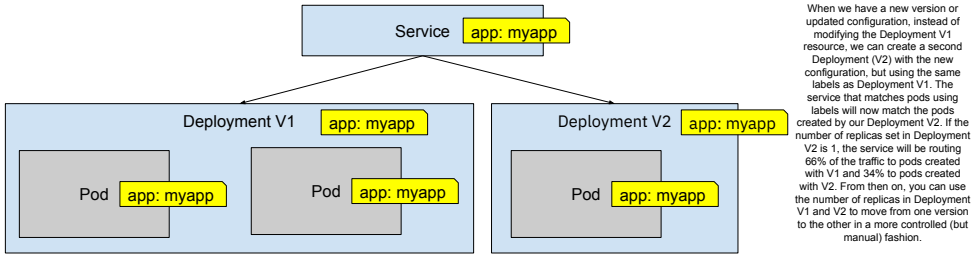


Figure 4.5 One service and two deployments sharing the same labels.

By creating a second deployment (using the same labels), we are routing traffic to both versions simultaneously, and how much traffic goes to each version is defined by the number of replicas configured for each deployment. By starting more replicas on Deployment V1 than Deployment V2 you can control what percentage of the traffic will be routed to each version. Figure 4.5 shows a 66%/34% (2 to 1 pods) traffic split

between V1 and V2. Then you can decrease the number of replicas for Deployment V1 and increase the replicas for V2 to move towards V2 slowly. Notice that you don't have a fine-grain control about which requests go to which version—the service forwards traffic to all the matched pods in a round-robin fashion.

Because we have replicas ready to receive traffic at all times, there shouldn't be any downtime of our services when we do canary releases.

A significant point about rolling updates and canary releases is that they depend on our services to support traffic being forwarded to different versions simultaneously without breaking. This usually means that we cannot have breaking changes from version 1.0 to version 1.1 that will cause the application (or the service consumers) to crash when switching from one version to another. Teams making the changes must be aware of this restriction when using rolling updates and canary releases, 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.

4.2.2 Blue/green deployments

Whenever you face a situation where you 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://codefresh.io/learn/software-deployment/what-is-blue-green-deployment/).

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 4.6.

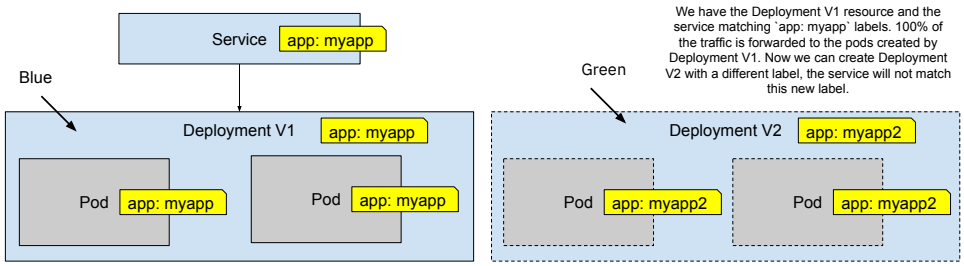


Figure 4.6 Two deployments using different labels.

In the same way, as we did with canary releases, we start with a service and a deployment. The first version of these resources is what we call "blue". For blue/green deployments, we can create a separate Deployment (V2) resource to start and prepare

for our service's new version when we have a new version. This new deployment needs different labels for the pods it will create, so the service doesn't match these pods yet. We can connect to Deployment V2 pods by using `kubectl port-forward` or running other in-cluster tests until we are satisfied that this new version is working. When we are happy with our testing, we can switch from blue to green by updating the `selector` labels defined in the service resource (figure 4.7).

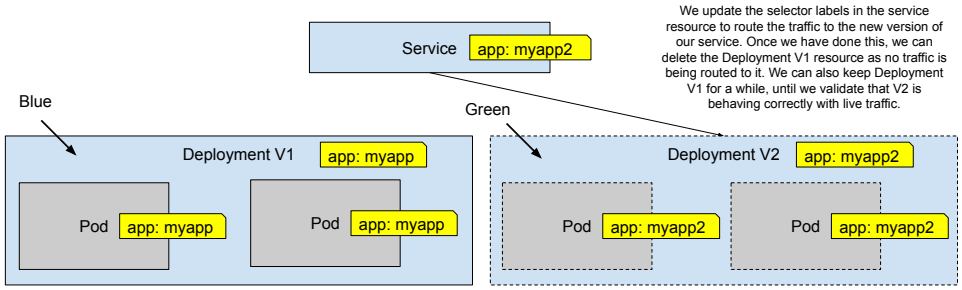


Figure 4.7 When the new version is ready, we switch the label from the service selector to match version V2 labels.

Blue/green deployments make a lot of sense when we cannot send traffic to both versions simultaneously, but it has the drawback of requiring both versions to be up simultaneously to switch traffic from one to the other. When we do blue/green deployments, the same number of replicas is recommended for the new version. We need to ensure that when traffic is redirected to the new version, this version is ready to handle the same load as the old version.

When we switch the label selector in the service resource, 100% of the traffic is routed to the new version, and Deployment V1 pods stop receiving traffic. This is quite an important detail because if some state was being kept in V1 Pods, you will need to drain the state from the pods, migrate, and make this state available for V2 pods. In other words, if your application holds a state in memory, you should write that state to persistent storage so V2 pods can access it and resume whatever work was done with that data. Remember that most of these mechanisms were designed for stateless workloads, so you need to make sure that you follow these principles to make sure that things work as smoothly as possible.

For blue/green deployments, we are interested in moving from one version to the next at a given time, but what about scenarios when we want to actively test two or more versions with our users simultaneously to decide which one performs better? Let's take a look at A/B testing next.

4.2.3 A/B testing

It is quite a common requirement to have two versions of your services running simultaneously, and we want to test these two versions to see which one performs better. (https://blog.christianposta.com/deploy/blue-green-deployments-a-b-testing-and-canary-releases/). Sometimes you want to try a new user interface theme, place some

UI elements in different positions or add new features to your app, and gather user feedback to decide which works best.

To implement this kind of scenario in Kubernetes, you need two different services pointing to two different deployments. Each deployment handles just one version of the application/service. If you want to expose these two versions outside the cluster, you can define an Ingress resource with two rules. Each rule will be in charge of defining the external path or subdomain to access each service (figure 4.8).

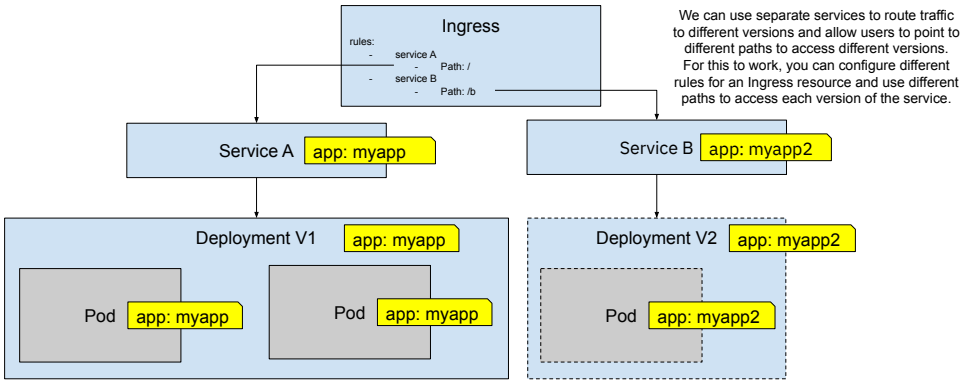


Figure 4.8 **Using two Ingress rules for pointing to A and B versions.**

If you have your application hosted under the www.example.com domain, the Ingress resource defined in figure 4.8 will direct traffic to Service A, allowing users to point their browsers to www.example.com/b to access Service B. Alternatively, and depending on how you have configured your Ingress controller, you can also use subdomains instead of path-based routing, meaning that to access the default version you can keep using www.example.com but to access Service B, you can use a subdomain such as test.example.com

I can hear you saying, what a pain. Look at all the Kubernetes resources I need to define and maintain to achieve something that feels basic and needed for everyday operations. And if things are complicated, teams will not use them. Let's quickly summarize the limitations and challenges we have found so far, so we can look at Knative Serving and how it can help us to streamline these strategies.

4.2.4 Limitations and complexities of using Kubernetes built-in building blocks

Canary releases, blue/green deployments, and A/B testing can be implemented using built-in Kubernetes resources. But as you have seen in the previous sections, creating different deployments, changing labels, and calculating the number of replicas needed to achieve percentage-based distribution of the requests is quite a major and error-prone task. Even if you use a GitOps approach, as shown with Argo CD or with other tools in chapter 1, creating the required resources with the right configurations is quite hard, and it takes a lot of effort.

We can summarize the drawbacks of implementing these patterns using Kubernetes building blocks as follows:

- Manual creation of more Kubernetes resources, such as deployments, services, and Ingress rules to implement these different strategies can be error-prone and cumbersome. The team implementing the release strategies must understand how Kubernetes behaves to achieve the desired output.
- No automated mechanisms are provided out-of-the-box to coordinate and implement the resources required by each release strategy.
- They are 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 auto scaler (more on this at the end of this chapter). Unfortunately, if you set the number of replicas to 0, there will not be any instance to answer requests, requiring you to have at least one replica running all the time.

Out of the box, Kubernetes doesn't include any mechanism to automate or facilitate these release strategies, which becomes a problem quite quickly if you are dealing with many services that depend on each other.

One thing is clear, your teams need to be aware of the implicit contracts imposed by Kubernetes regarding 12-factor apps and how their services APIs evolve to avoid downtime. Your developers need to know how Kubernetes' built-in mechanisms work to have more control over how your applications are upgraded.

If we want to reduce the risk of releasing new versions, we want to empower our developers to have these release strategies available for their daily experimentation.

In the next sections, we will look at Knative Serving and Argo rollouts, a set of tools and mechanisms built on top of Kubernetes to simplify all the manual work and limitations that we will find when trying to set up Kubernetes building blocks to enable teams with different release mechanisms.

Let's start first with Knative Serving that provides us with a flexible set of features to easily implement the release strategies described before.

4.3 Knative Serving: Advanced traffic management and release strategies

Knative is one of these technologies that are hard to go back to and not use when you learn about what it can do for you. After working with the project for almost three years and observing the evolution of some of its components, I can confidently say that every Kubernetes cluster should have Knative Serving installed; your developers will appreciate it. Knative provides higher-level abstractions on top of Kubernetes built-in resources to implement good practices and common patterns that enable your teams to go faster, have more control over their services, and facilitate the implementation of event-driven applications.

As the title of this section specifies, the following sections focus on a subset of the functionality provided by Knative, called Knative Serving. Knative Serving allows you to define "Knative Services", which dramatically simplifies implementing the release strategies exemplified in the previous sections. Knative Services will create Kubernetes built-in resources for you and keep track of their changes and versions, enabling scenarios that require multiple versions to be present simultaneously. Knative Services also provides advanced traffic handling and autoscaling to scale down to zero replicas for a serverless approach.

To install Knative Serving on your cluster, I highly recommend you check the official Knative documentation that you can find at https://knative.dev/docs/admin/install/serving/install-serving-with-yaml/.

If you want to install Knative locally, you should read the getting started guide for local development (https://knative.dev/docs/getting-started/#install-the-knative-quickstart-environment). Optionally, you can install `kn`, a small binary that allows you to interact with Knative more naturally. You can always use `kubectl` to interact with Knative resources.

When installing Knative Serving, you are installing a set of components in the `knative-serving` namespace that understand how to work Knative Serving custom resource definitions (figure 4.9).

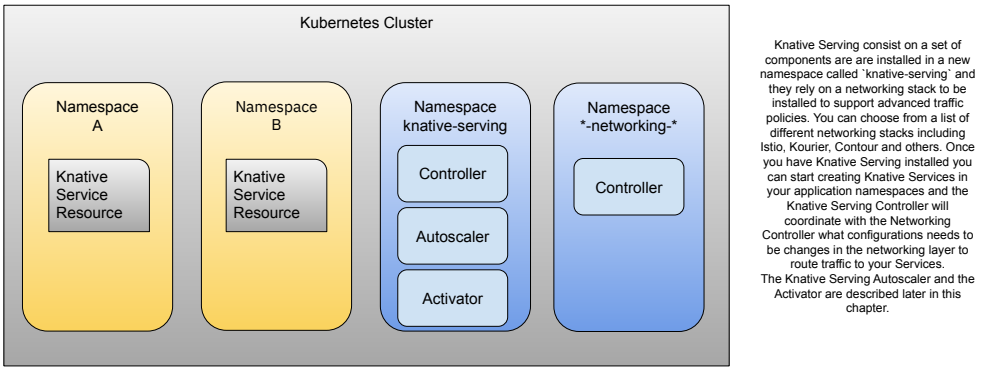


Figure 4.9 Knative Serving components in your Kubernetes cluster.

It is outside of the scope of this book to explain how Knative Serving components and resources work; my recommendation is that if I manage to get your attention with the examples in the following sections, you should check out the Knative In Action book by Jacques Chester (https://www.manning.com/books/knative-in-action).

4.3.1 Knative Services

Once you have Knative Serving installed, you can create *Knative Services*. I can hear you thinking: "But we already have Kubernetes Services, why do we need Knative Services?" believe me, I had the same feeling when I saw the same name but follow along, it does make sense.

When deploying each service in the Conference Application before, you needed to create at least two Kubernetes resources, a deployment and a service. We covered in a previous section that when using ReplicaSets internally, a deployment can perform rolling updates by keeping track of the configuration changes in the deployment resources and creating new ReplicaSets for each change we make. Earlier, we also discussed the need for creating an Ingress resource if we want to route traffic from outside the cluster. Usually, you will only create an Ingress resource to map the publicly available services, such as the user interface of the Conference platform.

Knative Services build on top of these resources and simplify how we define these resources that you need to create for every application service. While it simplifies the task and reduces the amount of YAML that we need to maintain, it also adds some exciting features, but before jumping into the features, let's look at what a Knative Service looks like in action. Let's start simple and use the email service from the Conference platform to demonstrate how Knative Services work:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: email-service
spec:
  template:
    spec:
      containers:
        - image: salaboy/fmtok8s-email-rest:0.1.0
```

You need to specify a name for the resource, as with any other Kubernetes resource.

You need to specify which container image you want to run.

In the same way as a deployment will pick the `spec.template.spec` field to cookie-cut pods, a Knative Service defines the configuration for creating other resources using the same field. Nothing too strange so far, but how is this different from a Kubernetes Service?

If you create this resource using `kubectl apply -f`, you can start exploring the differences.

You can also list all Knative Services using `kubectl get ksvc` (`ksvc` stands for Knative Service), and you should see your newly created Knative Service there:

```
NAME           URL                                          LATEST CREATED         READY
email-service  http://email-service.default.X.X.X.X.sslip.io  email-service-00001    True
```

There are a couple of details to notice right here; first, there is a URL that you can copy into your browser and access the service. If you were running in a cloud provider and configured DNS while installing Knative, this URL should be accessible immediately. The `LASTCREATED` column shows the name of the latest Knative revision of the service. Knative revisions are pointers to the specific configuration of our service, meaning that we can route traffic to them.

You can go ahead and test the Knative Service URL by using curl or by pointing your browser to `http://email-service.default.X.X.X.X.sslip.io/info` (X.X.X.X is the IP assigned by the Knative networking layer). You should see the following output:

```
{"name":"Email Service (REST)","version":"v0.1.0","source":"https://
github.com/salaboy/fmtok8s-email-rest/releases/tag/
v0.1.0","podId":"","podNamepsace":"","podNodeName":""}
```

As with any other Kubernetes resource, you can also use `kubectl describe ksvc email-service` to get a more detailed description of the resource.

If you list other well-known resources such as deployment, services, and pods, you will find out that Knative Serving is creating them for you and managing them. Because these are managed resources now, it is usually not recommended to change them manually. If you want to change your application configurations, you should edit the Knative Service resource.

A Knative Service, as we applied it before to our cluster, by default behaves differently from creating a service, a deployment, and an Ingress manually. A Knative Service by default:

- *Is accessible*: It exposes itself under a public URL so you can access it from outside the cluster. It doesn't create an Ingress resource, because it uses the available Knative networking stack that you installed previously. Because Knative has more control over the network stack and manages deployments and services, it knows when the service is ready to serve requests, reducing configuration errors between services and deployments.
- *Manages Kubernetes resources*: It creates two services and a deployment: Knative Serving allows us to run multiple versions of the same service simultaneously. Hence, it will create a new Kubernetes service for each version (which in Knative Serving are called revisions)
- *Collects service usage*: It creates a od with the specified `user-container` and a sidecar container called `queue-proxy`.
- *Scale up and down based on demand*: It automatically downscales itself to zero if no requests are hitting the service (by default after 90 seconds).
  - It achieves this by downscaling the deployment replicas to 0 using the data collected by the `queue-proxy`.
  - If a request arrives and there is no replica available, it scales up while queuing the request, so it doesn't get lost.
- *Configuration changes history is managed by Knative Serving*: If you change the Knative Service configuration, a new revision will be created. By default, all traffic will be routed to the latest revision.

Of course, these are the defaults, but you can fine-tune each of your Knative Services to serve your purpose and, for example, implement the previously described release strategies.

In the next section, we will look at how Knative Serving advanced traffic-handling features can be used to implement canary releases, blue/green deployments, A/B testing, and header-based routing.

4.3.2 Advanced traffic-splitting features

Let's start by looking at how you can implement a canary release for one of our application's services with a Knative Service.

This section starts by looking into doing canary releases using percentage-based traffic splitting. Then it goes into A/B testing by using tag-based and header-based traffic splitting.

CANARY RELEASES USING PERCENTAGE-BASED TRAFFIC SPLITTING

If you get the Knative Service resource (with `kubectl get ksvc email-service -oyaml`), you will notice that the `spec` section now also contains a `spec.traffic` section that was created by default, because we didn't specify anything:

```
traffic:
  - latestRevision: true
    percent: 100
```

By default, 100% of the traffic is being routed to the latest Knative Revision of the service.

Now imagine that you made a change in your service to improve how emails are sent, but your team is not sure how well it will be received by people, and we want to avoid having any backlash from people not wanting to sign into our conference because of the website. Hence, we can run both versions side by side and control how much of the traffic is being routed to each version (revision in Knative terms).

Let's edit (`kubectl edit ksvc email-service`) the Knative Service and apply the changes:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: email-service
spec:
  template:
    spec:
      containers:
        - image: salaboy/fmtok8s-email-rest:0.1.0-improved
  traffic:
  - percent: 80
    revisionName: email-service-00001
  - latestRevision: true
    percent: 20
```

You have updated the Docker image that the service will use to `fmtok8s-email-rest:0.1.0-improved.

You have created an 80% / 20% traffic split where 80% of the traffic will keep going to your stable version and 20% to the newest version that you just updated.

If you try now with `curl` you should be able to see the traffic split in action:

```
salaboy> curl http://email-service.default.x.x.x.x.sslip.io/info
{"name":"Email Service (REST)","version":"v0.1.0",
"source":"https://github.com/salaboy/fmtok8s-email-rest/releases/tag/v0.1.0"}
salaboy> curl http://email-service.default.x.x.x.x.sslip.io/info
{"name":"Email Service (REST)","version":"v0.1.0",
"source":"https://github.com/salaboy/fmtok8s-email-rest/releases/tag/v0.1.0"}
salaboy> curl http://email-service.default.x.x.x.x.sslip.io/info
{"name":"Email Service (REST)","version":"v0.1.0",
```

```
"source":"https://github.com/salaboy/fmtok8s-email-rest/releases/tag/v0.1.0"}
salaboy> curl http://email-service.default.x.x.x.x.sslip.io/info
{"name":"Email Service (REST)","version":"v0.1.0",
"source":"https://github.com/salaboy/fmtok8s-email-rest/releases/tag/v0.1.0"}
salaboy> curl http://email-service.default.x.x.x.x.sslip.io/info
{"name":"Email Service (REST) - IMPROVED!!","version":"v0.1.0",
 "source":"https://github.com/salaboy/fmtok8s-email-rest/releases/tag/
v0.1.0"}
```

One in five requests will go to the new "IMPROVED!!" version. Notice that this can take a while until the new Knative revision is running.

Once you have validated that the new version of your service is working correctly, you can start sending more traffic until you feel confident to move 100% of the traffic to it. If things go wrong, revert the traffic split to the stable version.

Notice that you are not limited to just two service revisions; you can create as many as you want as long as the traffic percentage is 100%. Knative will follow these rules and scale up the required revisions of your services to serve requests. You don't need to create any new Kubernetes resources, because Knative will create those for you, reducing the likelihood of errors that come with modifying multiple resources simultaneously.

By using percentages, you don't have control over where subsequent requests will land. Knative will make sure to maintain a fair distribution based on the percentages that you have specified. This can become a problem if, for example, you have a user interface instead of a simple REST endpoint (figure 4.10).

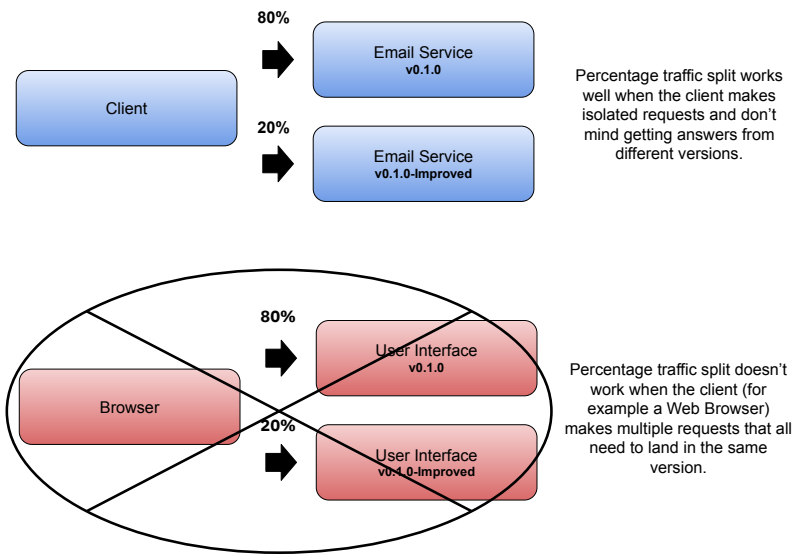


Figure 4.10 Percentage based traffic split scenarios and challenges.

User interfaces are complex because a browser will perform several GET requests to render the page HTML, CSS and images, and so forth. You can quickly end up in a situation where each request hits a different version of your application. Let's look at a different approach that might be better suited for testing user interfaces or scenarios when we need to ensure that several requests end up in the correct version of our application.

A/B TESTING WITH TAG-BASED ROUTING

If you want to perform A/B testing for a user interface, you will need to give Knative some way to differentiate where to send the requests. You have two options. First, you can point to a special URL for the service you want to try out, and the second is to use a request header to differentiate where to send the request. Let's look at these two alternatives in action. But let's now use the user interface for the Conference platform:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: ui-service
spec:
  template:
    spec:
      containers:
        - image: salaboy/fmtok8s-api-gateway:0.1.0
```

A You need to specify a name for this service.

You now define the API gateway / user interface container image, because we are going to test multiple requests going to the same version.

Once again, we have just created a Knative Service, but we cannot specify percentage-based routing rules because this container image contains a web application. Knative will not stop you from doing so. Still, you will notice requests going to different versions and errors popping up because a given image is not in one of the versions, or you end up with the wrong stylesheet (CSS) coming from the wrong version of the application.

Let's start by defining a Tag that you can use to test a new stylesheet. You can do that by modifying the Knative Service resource as we did before. First, change the image to `salaboy/fmtok8s-api-gateway:0.1.0-color` and then let's create some new traffic rules using tags:

```
traffic:
- percent: 100
  revisionName: ui-service-00001
- latestRevision: true
  tag: color
```

100% of the traffic will go to our stable version; no request will be sent to our newly updated revision.

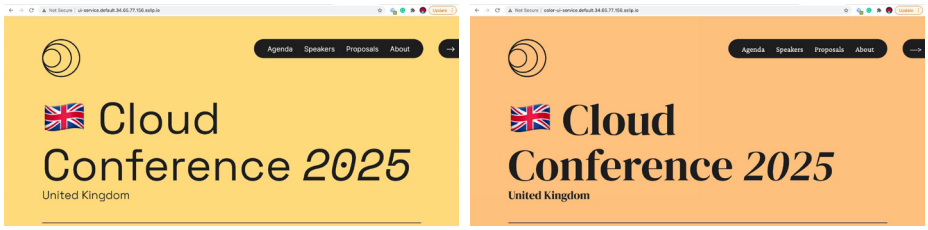
You have created a new tag called "color", you can find the URL for this new tag by describing the Knative Service resource.

If you describe the Knative Service (`kubectl describe ksvc ui-service`) you will find the URL for the Tag that we just created:

```
Traffic:
   Latest Revision:  false
   Percent:          100
   Revision Name:    ui-service-00001
   Latest Revision:  true
   Percent:          0
```

```
Revision Name:    ui-service-00001
Tag:              color
URL:              http://color-ui-service.default.x.x.x.x.sslip.io
```

Knative Serving had created a new URL (`http://color-ui-service.default.x.x.x.x.sslip.io`, where X.X.X.X is the IP assigned by the Knative networking layer.) by prepending the tag name to the service name. Check using the browser that can consistently access the modified version and the original one as well (figure 4.11).



User Interface for all traffic (A) Tag: Color (B)

Figure 4.11 A/B testing with tag-based routing.

Using tags guarantees that all requests are hitting the URL to the correct version of your service. One more option avoids you pointing to a different URL for doing A/B testing, and it might be useful for debugging purposes. The next section looks at tag-based routing using HTTP headers instead of different URLs.

A/B TESTING WITH HEADER-BASED ROUTING

Finally, let's look at an experimental feature (at the time of writing) that allows you to use HTTP headers to route requests. This experimental (at the time of writing) feature also uses tags to know where to route traffic, but instead of using a different URL to access a specific revision, you can add an HTTP Header that will do the trick.

Imagine that you want to enable developers to access a debugging version of the application. Application developers can set a special header in their browsers and then access a specific revision.

To enable this experimental feature, you or the administrator that installs Knative needs to patch a ConfigMap inside the `knative-serving` namespace:

```
kubectl patch cm config-features -n knative-serving -p '{"data":{"tag-header-
based-routing":"Enabled"}}'
```

Once the feature is enabled you can test this by changing again the Knative Service that we created in the previous section as follows:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
 name: ui-service
spec:
 template:
   spec:
```

```
    containers:
      - image: salaboy/fmtok8s-api-gateway:0.1.0-debug
traffic:
- percent: 100
  revisionName: ui-service-00001
- revisionName: ui-service-00002
  tag: color
- latestRevision: true
  tag: debug
```

You have another version for the API gateway / user interface that gives more debug information.

Now you pin down to revision 00002 to the tag color.

With this latest changes (using the -debug image) you create a new tag called "debug".

If you point your browser to the Knative Service URL (`kubectl get ksvc`), you will see the same application as always as shown in figure 4.12, but if you use a tool like ModHeader extension (https://chrome.google.com/webstore/detail/modheader/idgpnmonknjnojddfkpgkljpfnnfcklj?hl=en) for Chrome, you can set your custom HTTP headers that will be included in every request that the browser produce. For this example, and because the tag that you created is called `debug` you need to set the following HTTP header: `Knative-Serving-Tag: debug`. As soon as the HTTP header is present, Knative Serving will route the incoming request to the `debug` tag (figure 4.13).

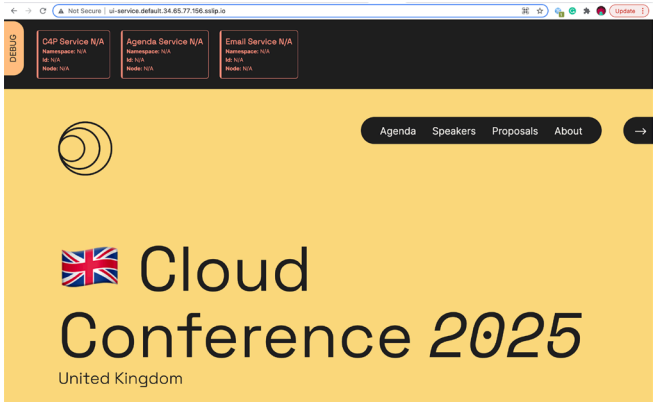


Figure 4.12 A/B testing with HTTP header-based routing to debug version.

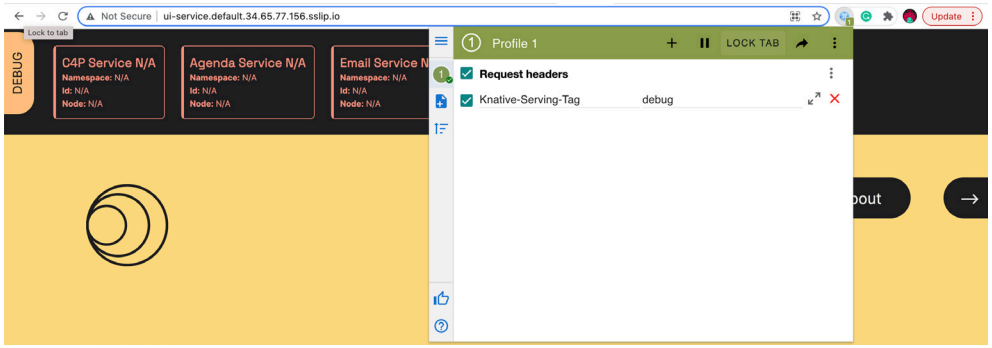


Figure 4.13 Using ModHeader Chrome extension to set custom HTTP headers.

The header-based routing experimental feature also uses tags to route traffic. If you still have the `color` tag created, you can test traffic is being routed by changing the `Knative-Serving-Tag` header value to `color` and see if the correct page comes back.

Both tag and header-based routing are designed to ensure that all requests will be routed to the same revision if you hit a specific URL (created for the tag), or if one particular header is present. Finally, let's take a look at how to do blue/green deployments with Knative Serving.

BLUE/GREEN DEPLOYMENTS

For situations where we need to change from one version to the next at a very specific point in time because there is no backward compatibility, we can still use tag-based routing with percentages. Instead of going gradually from one version to the next, we use percentages as a switch from 0 to 100 on the new version and from 100 to 0 on the old version.

Most blue/green deployment scenarios require coordination between different teams and services to make sure that both the service and the clients are updated at the same time. Knative Serving allows you to declaratively define when to switch from one version to the next in a controlled way (figure 4.14).

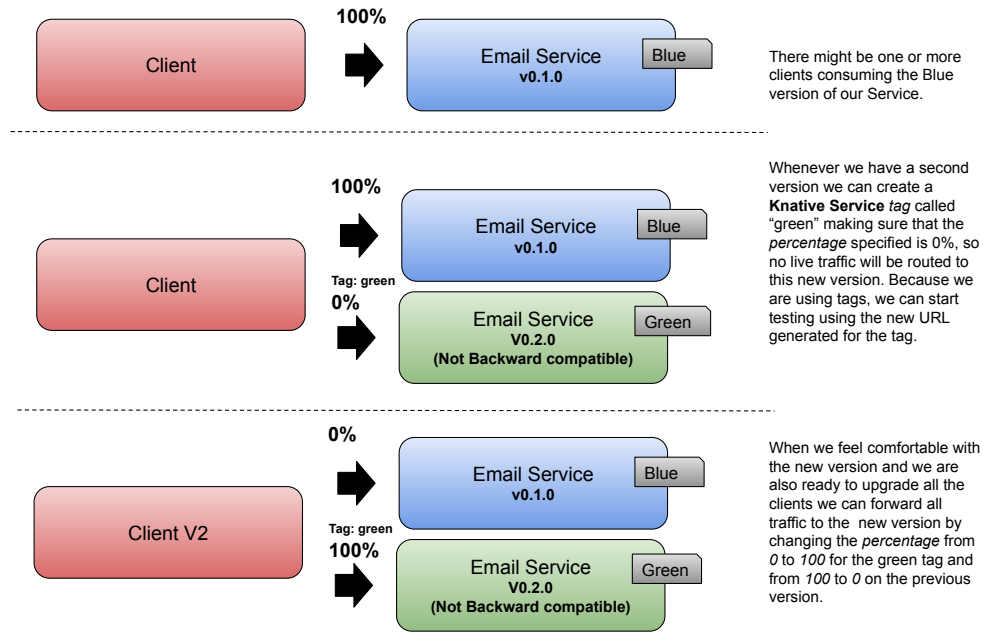


Figure 4.14 **Blue/green deployments using Knative Serving tag-based routing.**

To achieve this scenario described in figure 4.14, we can create the "green" tag for the new version inside our Knative Service as follows:

```
...
traffic:
    - revisionName: <first-revision-name>
      percent: 100 # All traffic is still being routed to the first revision
    - revisionName: <second-revision-name>
      percent: 0 # 0% of traffic routed to the second revision
      tag: green # A named route
```

By creating a new tag (called "green") we will have now available a new URL to access the new version for testing. This is particularly useful to test new versions of the clients, as if the Service API is changing with a non-backward compatible change, clients might need to be updated as well.

Once all tests are performed, we can safely switch all traffic to the "green" version of our service:

```
...
traffic:
    - revisionName: <first-revision-name>
      percent: 0 # All traffic is still being routed to the first revision
    - revisionName: <second-revision-name>
      percent: 100 # 100% of traffic routed to the second revision
      tag: green # A named route
```

Generally, we cannot progressively move traffic from one version to the next, because the client consuming the service will need to understand that requests might land into different (and non-compatible) versions of the service.

In the previous sections, we looked into how Knative Serving simplifies the implementation of different release strategies for your teams to continuously deliver features and new versions of your services. Knative Serving reduces the need to create several Kubernetes built-in resources to manually implement the release strategies described in this chapter. It does so by providing high-level abstractions, such as Knative Services, which create and manage Kubernetes built-in resources and a network stack for advanced traffic management.

In the final section of this chapter, we will look at the Knative Autoscaler, which adds to the benefits of using Knative.

4.3.3 The Knative Serving Autoscaler

An important part of Knative Serving is the Autoscaler, which allows your Knative Services by default to scale down to zero and handle the logic to queue requests if no replica is available to answer incoming requests until at least one replica is up (figure 4.15).

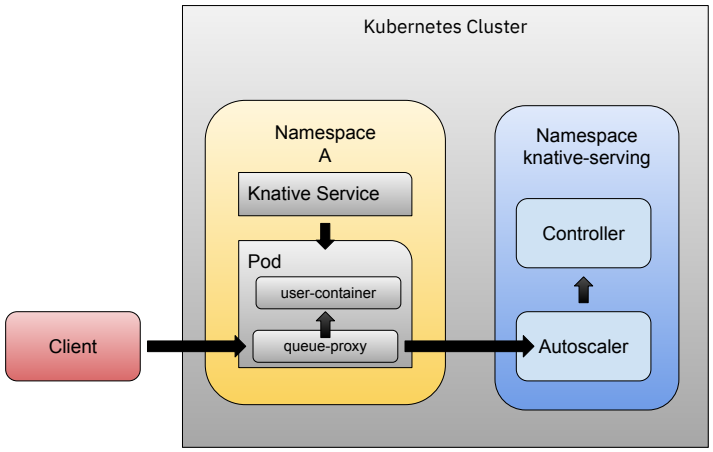


Figure 4.15 Knative Serving Autoscaler and "queue-proxy".

The Knative Autoscaler is installed by default when you install Knative Serving, and it continuously monitors your Knative Service requests. As figure 4.15 shows, monitoring all inbound traffic occurs when a sidecar container is attached to your application container, and all traffic is routed via the "queue-proxy" container. The "queue-proxy" container collects metrics and informs the Autoscaler about how much traffic the application container ("user-container") is receiving.

Based on this information, the Autoscaler can notify the Knative Service controller to scale up or down the replicas of your service. If no requests are being handled by the "user-container" for a while, the Autoscaler will notify the Knative Service controller to scale down to zero the number of replicas.

Scaling up and down works out of the box, but you can always fine-tune the behavior based on your application capabilities. In general, to configure the Autoscaler, you can annotate your Knative Services with the amount of concurrent requests that your application can handle per replica. This information will be used by the Autoscaler to decide when new replicas are needed:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/target: "100" # 100 concurrent request by
default
```

You can also change this for all Knative Services by changing a global parameter inside the `'config-autoscaler'` ConfigMap:

```
apiVersion: v1
kind: ConfigMap
metadata:
 name: config-autoscaler
 namespace: knative-serving
data:
 container-concurrency-target-default: "100" # 100 concurrent request by
default
```

Knative Serving also allows you to configure how to deal with sudden burst of traffic doing buffering and making sure that no request is lost. I recommended you check the official documentation for more details about how these mechanisms can be configured at https://knative.dev/docs/serving/autoscaling/concurrency/.

Finally, you might be wondering how Knative Serving deals with traffic when there is no replica available, as both, the `user-container` and the `queue-proxy` are downscaled to zero if no requests are going to the service for some time. Here is where the Activator component jumps into the queue and buffers the requests until a replica is spawned up by the Knative service controller, and the requests can be forwarded to that replica (figure 4.16).

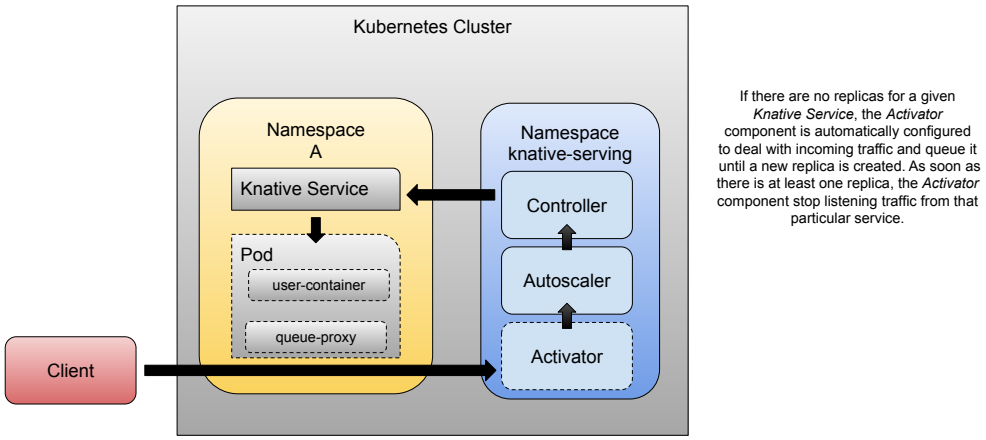


Figure 4.16 Knative Serving Activator queues traffic for services with no replicas.

On top of helping us implement different release strategies without the burden of creating tons of different Kubernetes resources, Knative Serving also allows us to scale up and down based on the concurrent requests that our services are handling at a given time. These mechanisms work out of the box and are based on live traffic, so there is no need to change the number of replicas of a given service manually, Knative Serving will adapt to the demand, making sure that the number of replicas alive can handle the current traffic and saving resources when they are not needed.

Let's now switch to another alternative to manage release strategies in Kubernetes with Argo rollouts.

4.4 Argo rollouts: Release strategies automated with GitOps

In most cases you will see Argo rollouts working hand in hand with Argo CD. This makes a lot of sense because we want to enable a delivery pipeline that removes the need to manually interact with our environments to apply configuration changes. For the examples in the following sections, we will focus only on Argo rollouts, but in real-life scenarios, you shouldn't be applying resources to the environments using `kubectl`, Argo CD will do it for you.

Argo rollouts, as defined on the project website, is: "a Kubernetes controller and set of CRDs which provide advanced deployment capabilities such as blue-green, canary, canary analysis, experimentation, and progressive delivery features to Kubernetes". As we have seen with other projects, Argo rollouts extend Kubernetes with the concepts of rollouts, analysis, and experimentations to enable progressive delivery features. The main idea with Argo rollouts is to use the Kubernetes built-in blocks without the need to manually modify and keep track of deployment and services resources.

Argo rollouts is composed of two big parts, the Kubernetes controller that implements the logic to deal with our rollouts definitions (also analysis and experimentations) and then a "kubectl" plugin 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 it 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 or using Knative Services.

4.4.1 Argo rollouts canary rollouts

First, we'll begin by creating our first rollout resource. With Argo rollouts, we will be not defining deployments as we will be delegating this responsibility to the Argo rollouts controller. Instead, we define an Argo rollout resource that also provides our pod specification (PodSpec in the same way that a deployment defines how pods need to be created).

For these examples, we will be using only the email service from the Conference platform application, and we will not 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 manages the creation of pods using what we define inside the `spec.template` and `spec.replicas` fields. But it adds the `spec.strategy` section which for this case is set to `canary` and defines the steps (amount of traffic, or weight) that will be sent to the canary) in which the rollout will happen. As you can see, you can also define a pause between each step. The `duration` is expressed in seconds and allows us to have a fine grain control on how the traffic is shifted to the canary version. If you don't specify the `duration` parameter, the rollout will wait there until manual intervention happens. Let's see how this rollout works in action. Let's apply the rollout resource to our Kubernetes cluster:

```
> kubectl apply -f rollout.yaml
```

Remember that if you are using Argo CD, instead of manually applying the resource, you will 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 replica sets, 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 on the main page (figure 4.17).

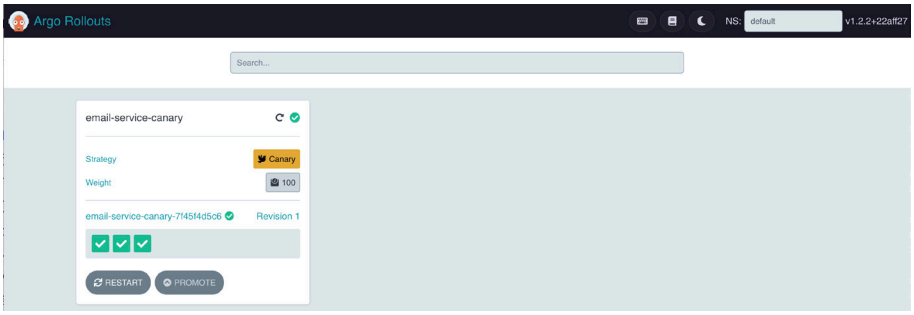


Figure 4.17 **Argo Rollouts Dashboard.**

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, as shown in figure 4.18.

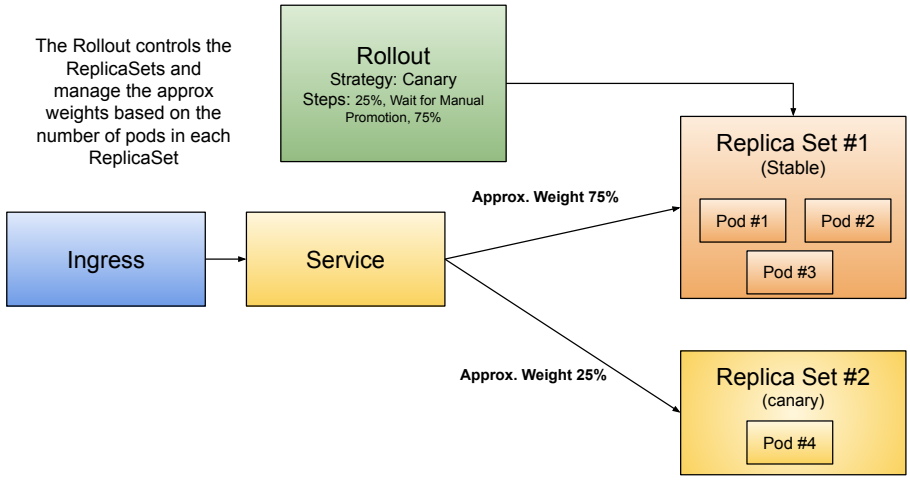


Figure 4.18 **Argo rollouts in a canary release with Kubernetes resources.**

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 yet. Now if we want to update the rollout `spec.template` section with a new container image reference or change environment variables a new revision will be created, and the canary strategy will kick in.

In a new terminal, we can watch the rollout status before doing any modification, so we can see the rollout mechanism in action when we change the rollout specification. If we want to watch how the rollout progress after we make some changes, we 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 "‖ 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 4.19).

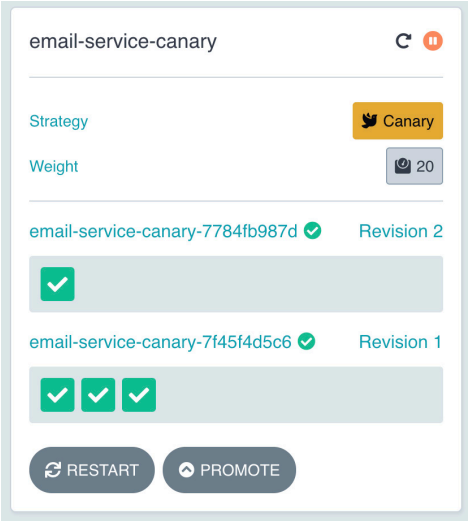


Figure 4.19 A canary release has been created with approximately 20% of the traffic being routed to it.

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 continuing the rollout process. To do that, we can query the "info" endpoint again to see that approximately 25% of the time we hit the canary:

```
salaboy> http localhost/info
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 230
Content-Type: application/json
Date: Fri, 12 Aug 2022 07:56:56 GMT

{
    "name": "Email Service", # stable
    "podId": "email-service-canary-7f45f4d5c6-fhzzt",
    "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 underlying 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, so 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 this, 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 just manually promoted the rollout, the best practice would be utilizing Argo rollouts automated analysis steps, which we will dig into in section 4.3.2.

If you take a look at the dashboard, you will notice that you can also promote the rollout to move forward using the Button Promote in the rollout. Promotion in this context only means that the rollout can continue to execute the next steps defined in the "spec. strategy" section:

```
 strategy:
   canary:
     steps:
     - setWeight: 25
     - pause: {}
     - setWeight: 75
     - pause: {duration: 10}
```

After the manual promotion, the weight is going to be set to 75% followed by a pause of 10 seconds, to finally set the wait to 100%. At that point, you should see that revision 1 is being downscaled while progressively revision 2 is being upscaled to take all the traffic:

```
NAME                                                KIND        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
```

You can see this rollout progression live in the dashboard as well (figure 4.20).

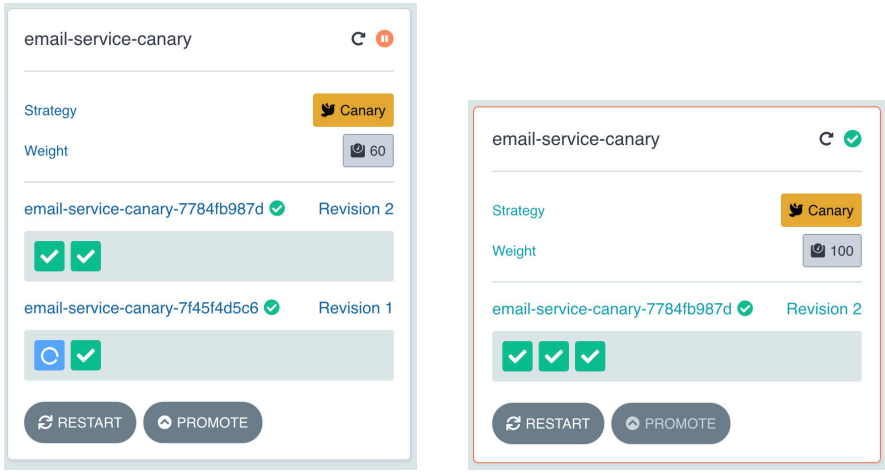


Figure 4.20 The canary revision is promoted to be the stable version.

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 4.1.2, using two deployment resources to implement the same pattern, with Argo rollouts you have full control over how your canary release is promoted, how much time do you want to wait before shifting more traffic to the canary and how many manual interventions steps do you want to add? Let's now jump to see how a blue-green deployment works with Argo rollouts.

4.4.2 Argo rollouts and blue-green deployments

Earlier, we covered the advantages and the reasons behind why you would be interested in doing a blue-green deployment using Kubernetes' basic building blocks. We

have also seen how manual the process is and how these manual steps can open the door for silly mistakes that can bring our services down. In this section, we will look at how Argo rollouts allows us to implement blue-green deployments following the same approach that we used previously for canary deployments.

Let's look at what our rollout with a blue-green 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 Argo CD:

```
> 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 4.21.

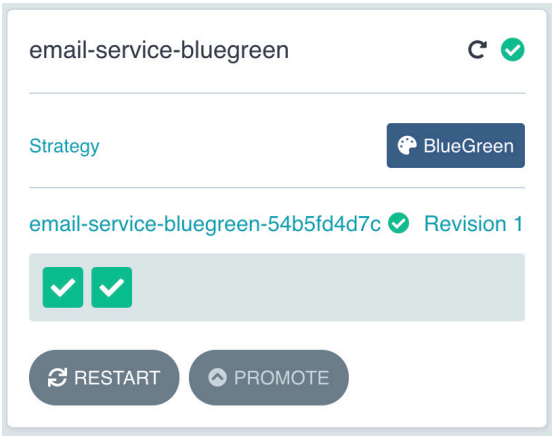


Figure 4.21 Blue-green deployment in the Argo Rollout Dashboard.

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 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 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 the selector and labels, the rollout with `blueGreen` strategy is handling these links automatically for us. This avoids us manually creating these labels and making sure they match.

You can check now that there are two revisions (and ReplicaSets) with 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 4.22).

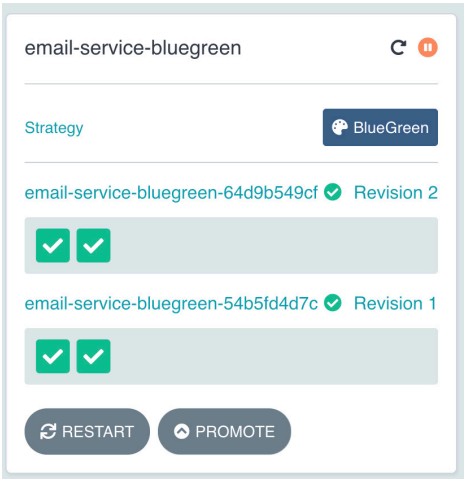


Figure 4.22 Argo Rollout Dashboard blue and green revisions are up.

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 4.23).

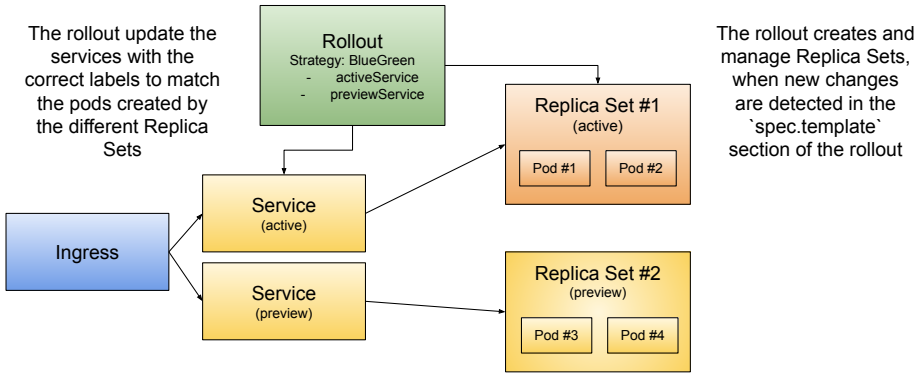


Figure 4.23 **Blue-green deployment of Kubernetes resources.**

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 once again use the Argo rollouts promotion mechanism from the terminal using the CLI or from the Argo Rollouts Dashboard. Try to promote the Rollout using the Dashboard now instead of using `kubectl`.

Notice that a 30 seconds delay is added by default before the scaling down of our revision #1 (this can be controlled using the property called: `scaleDownDelaySeconds`), but the promotion (switching labels to the services) happens the moment we hit the `PROMOTE` button (figure 4.24).

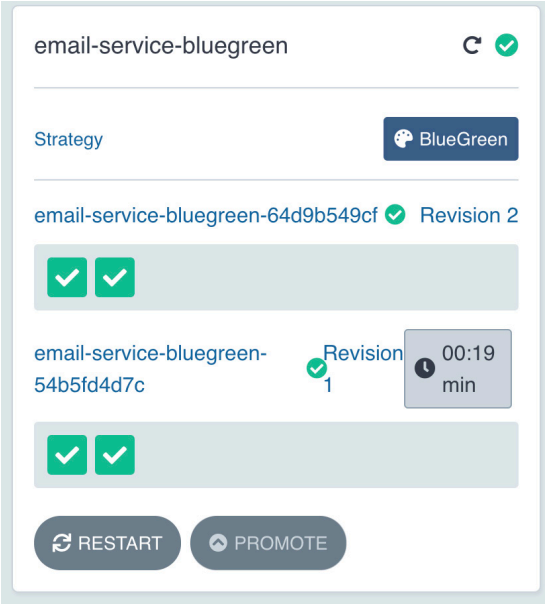


Figure 4.24 Green service promotion using the Argo Rollouts Dashboard (delay 30 seconds).

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 that will include these changes.

Now that we have seen the basics of canary releases and blue-green deployments with Argo rollouts, let's look at more advanced mechanisms provided by Argo rollouts.

4.4.3 Argo rollouts analysis for progressive delivery

So far, we have managed to have more control over our different release strategies, but where Argo rollouts really shine is by providing the `AnalysisTemplate` CRD, which 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, Datadog, and New Relic among others, providing maximum flexibility to define these automated tests against the new revisions of our services (figure 4.25).

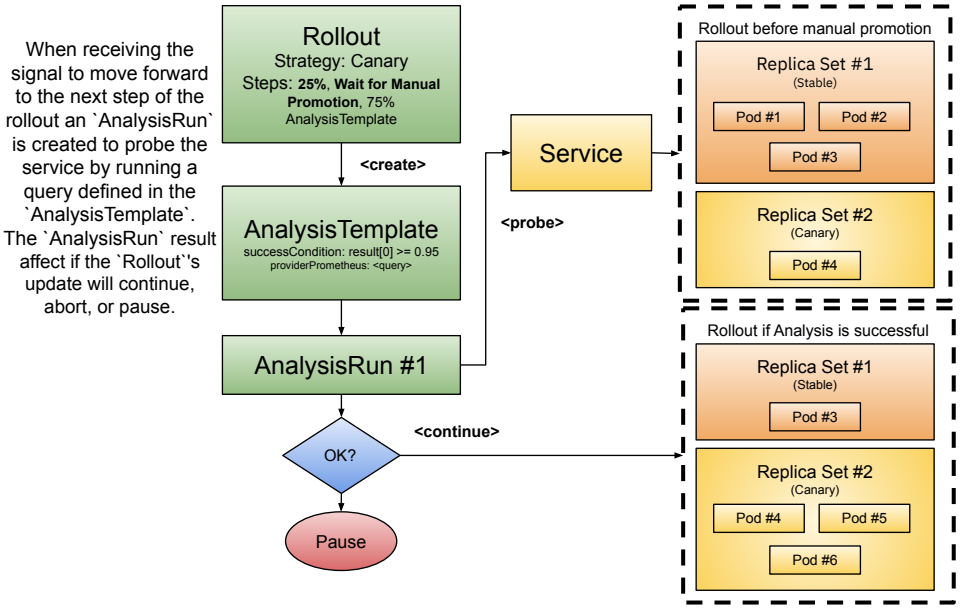


Figure 4.25 **Argo rollouts and analysis working together to make sure that our new revisions are sound before shifting more traffic to them.**

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>
```

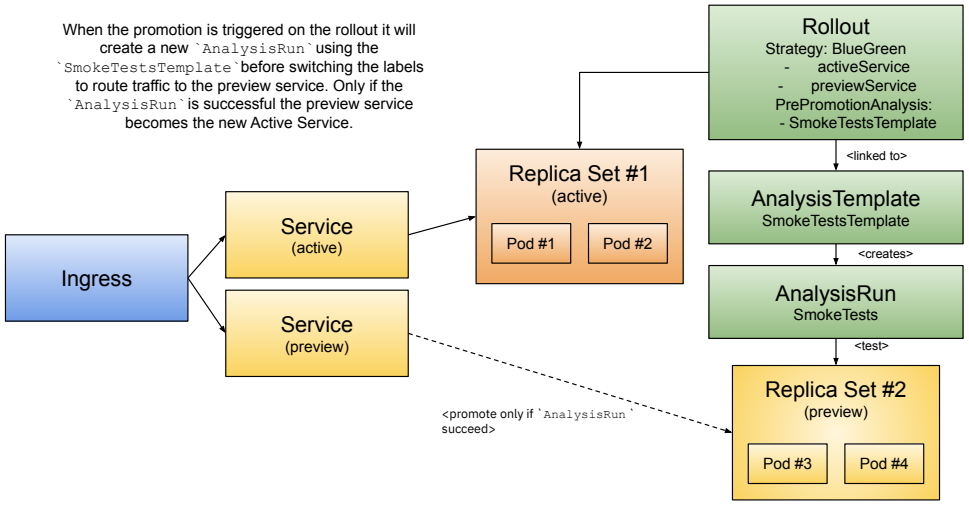
Then in our Rollout we can make reference to this template and define when a new AnalysisRun will be created, for example if we want to run the first analysis after step 2:

```
strategy:
    canary:
      analysis:
        templates:
        - templateName: success-rate
        startingStep: 2 # delay starting analysis run until 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 BlueGreen strategy, we can trigger analysis runs pre and post promotion (figure 4.26).



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 to 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/.

4.4.4 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 use 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), Traefik Proxy, among others. As described in the documentation, if we have more advanced traffic management capabilities, we can implement techniques such as:

- 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:
            name: "mirror-route" # 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 the scope for this section (figure 4.27). 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.

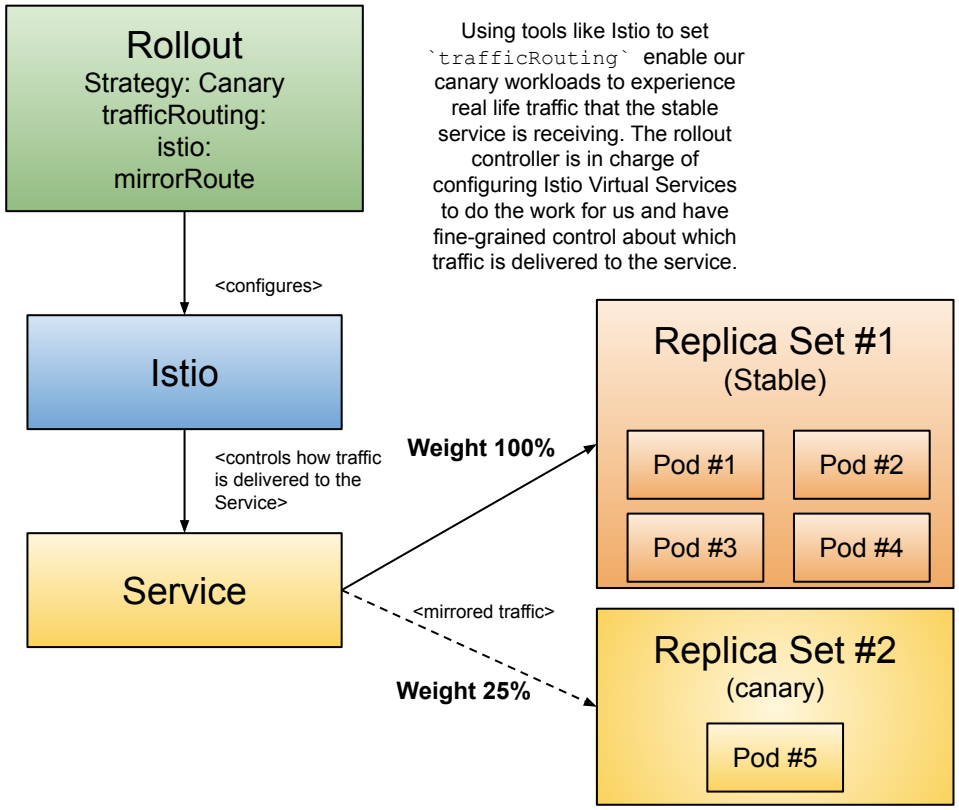


Figure 4.27 Traffic mirroring to canary release using Istio.

Something that you should know is that when using "trafficManagement" features, the rollout canary strategy will behave differently than when we are not using any rules. More specifically, the Stable version of the service will not be downscaled when going through a canary release rollout. This ensures that the Stable service can handle 100% of the traffic; the usual calculations apply for the canary replica count.

I strongly recommend checking the official documentation (https://argoproj.github.io/argo-rollouts/features/traffic-management/) and following the examples there, because the rollouts need to be configured differently depending on the service mesh that you have available.

4.5 Final thoughts

In this book, we've explored the concepts of pipelines, infrastructure, and enabling developers with Kubernetes, Argo CD, and Crossplane. The current state of software delivery is the best it's ever been. With tools like Crossplane to help us manage infrastructure in a cloud-native fashion and Argo CD for handling application delivery, you have a two-tier software delivery system for handling every kind of infrastructure.

Crossplane for handling legacy infrastructure alongside the long-running underlying infrastructure for applications. Argo CD for handling applications that change frequently and need more fine-tuning.

A major motivator for adopting Kubernetes is the ability to improve software delivery. The declarative configuration makes it much easier to manage state and changes. But taking the old approach of just extending CI pipelines to deploy software will leave the benefits of Kubernetes unrealized. 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 software more frequently. Using these tools with the right strategies will dramatically improve your software delivery experience and reliability. Make sure you plan time for your teams to research and choose which tools they will use to implement these release strategies. Luckily there are lots of software vendors available to help you get it done quicker.

Summary

- Kubernetes' built-in deployments and services can be used to implement rolling updates, canary releases, blue-green deployments, and A/B testing patterns. This allows us to manually move from one version of our service to the next without causing service disruption
- Canary releases can be implemented by having a Kubernetes service, two deployments (one for each version), and a shared label that routes traffic to both versions at the same time.
- Blue-green deployments can be implemented by having a single Kubernetes service, two independent deployments, and switching labels when the new version has been tested and is ready to be promoted.
- A/B testing can be implemented by having two services and two deployments and two Ingress resources.
- Knative Serving introduces an advanced networking layer that allows us to have fine-grained control over how traffic is routed to different versions of our services that can be deployed at the same time. This feature is implemented on top of Knative Services and reduces the manual work of creating several Kubernetes resources for implementing canary releases, blue-green deployments, and A/B testing release strategies. Knative Serving simplifies the operational burden of moving traffic to new versions and, with the help of the Autoscaler, can scale up and down based on demand.
- Argo rollouts integrate with Argo CD (that we discussed in chapter 1) and provide an alternative to implement different release strategies using the concept of rollouts. Argo rollouts also include features to automate the testing of new releases to make sure that we move safely between versions.