Sponsored by chronosphere Google Cloud

Platform Engineering on Kubernetes

Selected Chapters

Mauricio Salatino

MANNING

chronosphere Google Cloud

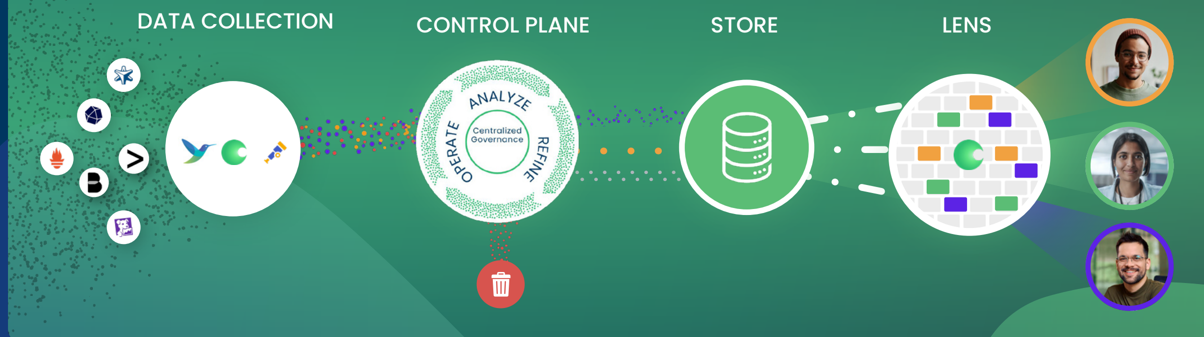
The observability platform that puts you in control

Modern, containerized environments produce overwhelming volumes of telemetry data. Most observability solutions lack the insight and control needed to separate the valuable signal from the noise, especially at scale.

The result: runaway costs and slower incident resolution.

Chronosphere, used by the teams at DoorDash, Affirm, and Robinhood, is purpose-built for large-scale Kubernetes environments. It puts you in control of your telemetry data, providing the insights and tools to reduce noise, optimize costs, and resolve issues faster.

On average, **our platform reduces data volumes and costs by 84%** while saving organizations thousands of developer hours.



Want to learn more?

Platform Engineering on Kubernetes

SELECTED CHAPTERS

MAURICIO SALATINO

MANNING
SHELTER ISLAND

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

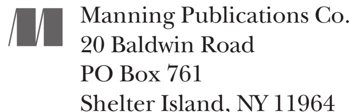
© 2025 by Manning Publications Co. All rights Reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

♾ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

The author and publisher have made every effort to ensure that the information in this book was correct at press time. The author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause, or from any usage of the information herein.



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

Development editor: Ian Hough
Technical development editor: Raphael Villela
Review editor: Dunja Nikitović
Production editor: Aleksandar Dragosavljević
Copy editor: Katie Petito
Technical proofreader: Werner Dijkerman
Typesetter: Tamara Švelić Sabljić
Cover designer: Marija Tudor

ISBN 9781633434509
Printed in the United States of America

contents

about the author *v*

1 *Cloud-native application challenges 1*

1.1 Running our cloud-native applications 2

Choosing the best Kubernetes environment for you 2
Installing the walking skeleton 4

1.2 Installing the Conference application with a single command 7

Verifying that the application is up and running 9
Interacting with your application 10

1.3 Inspecting the walking skeleton 15

Kubernetes deployments basics 15 ▪ *Exploring deployments 16*
ReplicaSets 17 ▪ *Connecting services 20* ▪ *Exploring services 20* ▪ *Service discovery in Kubernetes 21*
Troubleshooting internal services 22

1.4 Cloud-native application challenges 23

Downtime is not allowed 24 ▪ *Service's resilience built-in 28*
Dealing with the application state is not trivial 30 ▪ *Dealing with inconsistent data 32* ▪ *Understanding how the application is working 33* ▪ *Application security and identity management 34* ▪ *Other challenges 36*

1.5 Linking back to platform engineering 36

2 *Multi-cloud (app) infrastructure 40*

2.1 The challenges of managing infrastructure in Kubernetes 41

Managing your application infrastructure 42
Connecting our services to the newly provisioned infrastructure 45
I've heard about Kubernetes operators. Should I use them? 46

2.2 Declarative infrastructure using Crossplane 48

Crossplane providers 48 ▪ Crossplane compositions 50
Crossplane components and requirements 51 ▪ Crossplane behaviors 53

2.3 Infrastructure for our walking skeleton 55

Connecting our services with the new provisioned infrastructure 65

2.4 Linking back to platform engineering 69

3 *Platform capabilities: Shared application concerns 73*

3.1 What are most applications doing 95% of the time? 74

The challenges of coupling application and infrastructure 75
Service-to-service interaction challenges 76 ▪ Storing/reading state challenges 79 ▪ Asynchronous messaging challenges 81
Dealing with edge cases (the remaining 5%) 82

3.2 Standard APIs to separate applications from infrastructure 83

Exposing platform capabilities challenges 85

3.3 Providing application-level platform capabilities 87

Dapr in action 87 ▪ Dapr in Kubernetes 88 ▪ Dapr and your applications 91 ▪ Feature flags in action 92 ▪ Updating our Conference application to consume application-level platform capabilities 94

3.4 Linking back to platform engineering 100

about the author

MAURICIO SALATINO works for Diagrid (https://diagrid.io) as an Open Source Software Engineer. He is currently a Dapr OSS Contributor and Knative Steering Committee member. Before working at Diagrid, Mauricio spent the last 10 years building tools for Cloud-Native developers at companies such as Red Hat and VMware. When he is not writing tools for developers or contributing to Open Source projects in the Cloud Native space, he teaches about Kubernetes and Cloud-Native via his Blog (https://salaboy.com/) and/or LearnK8s (https://learnk8s.io).

1 Cloud-native application challenges

This chapter covers

- Working with a cloud-native application running in a Kubernetes cluster
- Choosing between local and remote Kubernetes clusters
- Understanding the main components and Kubernetes resources
- Understanding the challenges of working with cloud-native applications

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

To have a cloud-native application up and running, you will need a Kubernetes cluster. In this chapter, you will work with a local Kubernetes cluster using a project called KinD (Kubernetes in Docker, https://kind.sigs.k8s.io/). This local cluster

will allow you to deploy applications locally for development and experimentation. To install a set of microservices, you will use Helm, a project that helps package, deploy, and distribute Kubernetes applications. You will install the walking skeleton services which implements a Conference application.

Once the services for the Conference application are up and running, you will inspect its Kubernetes resources to understand how the application was architected and its inner workings by using `kubectl`. Once you get an overview of the main pieces inside the application, you will jump ahead to try to break the application, finding common challenges and pitfalls that your cloud-native applications can face. This chapter covers the basics of running cloud-native applications in a modern technology stack based on Kubernetes, highlighting the good and the bad that come with developing, deploying, and maintaining distributed applications.

1.1 Running our cloud-native applications

To understand the innate challenges of cloud-native applications, we need to be able to experiment with a simple example that we can control, configure, and break for educational purposes. In the context of cloud-native applications, "simple" cannot be a single service, so for simple applications we will need to deal with the complexities of distributed applications such as networking latency, resilience to failure on some of the applications' services, and eventual inconsistencies. To run a cloud-native application, you need a Kubernetes cluster. Where this cluster is going to be installed and who will be responsible for setting it up are the first questions that developers will have. It is quite common for developers to want to run things locally, on their laptop or workstation, and with Kubernetes, this is possible—but is it optimal? Let's analyze the advantages and disadvantages of running a local cluster against other options.

1.1.1 Choosing the best Kubernetes environment for you

This section doesn't cover a comprehensive list of all the available Kubernetes flavors, but it focuses on common patterns in how Kubernetes clusters can be provisioned and managed. There are three possible alternatives—all of them with advantages and drawbacks:

- *Local Kubernetes in your laptop/desktop computer:* I tend to discourage people from running Kubernetes on their laptops. As you will see in the rest of the book, running your software in similar environments to production is highly recommended to avoid problems that can be summed up as "but it works on my laptop." These problems are mostly caused by the fact that when you run Kubernetes on your laptop, you are not running on top of a real cluster of machines. Hence, there are no network round-trips and no real load balancing.
  - *Pros:* Lightweight, fast to get started, good for testing, experimenting, and local development. Good for running small applications.
  - *Cons:* Not a real cluster, it behaves differently, and has reduced hardware to run workloads. You will not be able to run a large application on your laptop.

- *On-premise Kubernetes in your data center:* This is a typical option for companies with private clouds. This approach requires the company to have a dedicated team and hardware to create, maintain, and operate these clusters. If your company is mature enough, it might have a self-service platform that allows users to request new Kubernetes clusters on demand.
  - *Pros:* A real cluster on top of real hardware will behave closer to how a production cluster will work. You will have a clear picture of which features are available for your applications to use in your environments.
  - *Cons:* It requires a mature operation team to set up clusters and give credentials to users, and it requires dedicated hardware for developers to work on their experiments.
- *Managed service Kubernetes offering in a cloud provider:* I tend to be in favor of this approach, because using a cloud provider service allows you to pay for what you use, and services like Google Kubernetes Engine (GKE), Azure AKS, and AWS EKS are all built with a self-service approach in mind, enabling developers to spin up new Kubernetes clusters quickly. There are two primary considerations:

  1. You need to choose one cloud provider and have an account with a big credit card to pay for what your teams will consume. This might involve setting up some caps in the budget and defining who has access. By selecting a cloud provider, you might be in a vendor lock-in situation if you are not careful.
  2. Everything is remote, and for developers and other teams that are used to work locally, this is too big of a change. It takes time for developers to adapt, because the tools and most of the workloads will run remotely. This is also an advantage, because the environments used by your developers and the applications that they are deploying are going to behave as if they were running in a production environment.

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

A final recommendation is to check the following repository, which contains free Kubernetes credits in major cloud providers: https://github.com/learnk8s/free-kubernetes. I've created this repository to keep an updated list of these free trials that you can use to get all the examples in the book up and running on top of real infrastructure. Figure 1.1 summarizes the information contained in the previous bullet points.

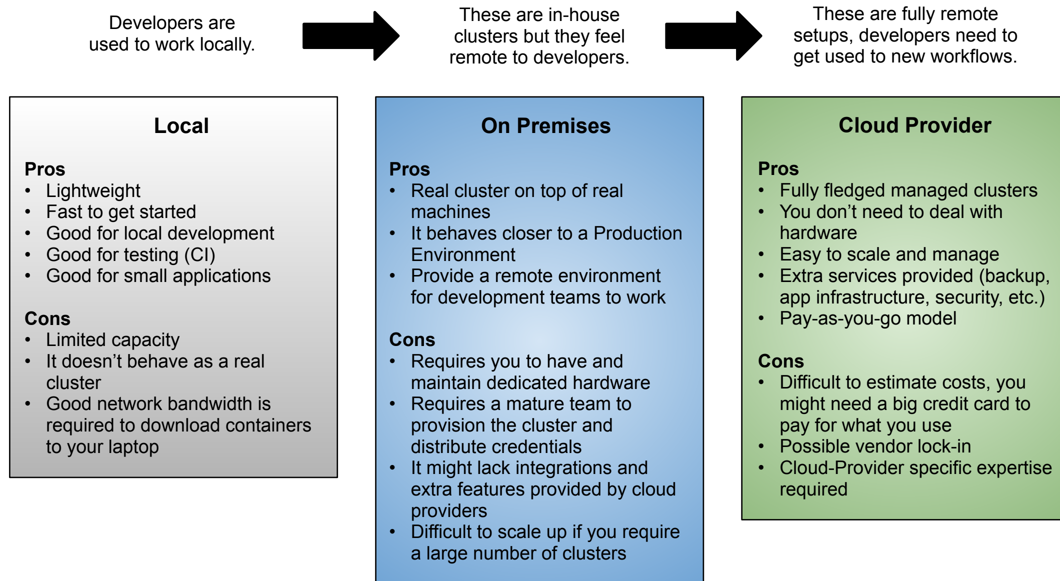


Figure 1.1 Kubernetes cluster Local vs. Remote setups.

While these three options are all valid and have drawbacks, in the next sections, you will use Kubernetes KinD (Kubernetes in Docker, https://kind.sigs.k8s.io/) to deploy the walking skeleton in a local Kubernetes environment running on your laptop/pc. Check the step-by-step tutorial located at https://github.com/salaboy/platforms-on-k8s/tree/main/chapter-2#creating-a-local-cluster-with-kubernetes-kind to create your local KinD cluster that we will use to deploy our walking skeleton, the Conference application.

Notice that the tutorial creates a local KinD cluster that simulates having three nodes and a special port mapping to allow our Ingress controller to route incoming traffic that we will send to http://localhost.

1.1.2 Installing the walking skeleton

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

- *Deployments:* Declaratively define how many replicas of your container need to be up for your application to work correctly. Deployments also allow us to choose which container (or containers) we want to run and how these containers must be configured (using environment variables).

- *Services:* Declaratively define a high-level abstraction to route traffic to the containers created by your deployments. It also acts as a load balancer between the replicas inside your deployments. Services enable other services and applications inside the cluster to use the service name instead of the physical IP address of the containers to communicate, providing what is known as service discovery.
- *Ingress:* Declaratively define a route for routing traffic from outside the cluster to services inside the cluster. Using Ingress definitions, we can expose the services that are required by client applications running outside the cluster.
- *ConfigMap/secrets:* Declaratively define and store configuration objects to set up our service instances. Secrets are considered sensitive information that should have protected access.

These YAML files will be complex and hard to manage if you have large applications with tens or hundreds of services. Keeping track of the changes and deploying applications by applying these files using `kubectl` becomes a complex job. It is beyond the scope of this book to cover a detailed view of these resources, and other resources are available such as the official Kubernetes documentation page (https://kubernetes.io/docs/concepts/workloads/). In this book, we will concentrate on how to deal with these resources for large applications and the tools that can help us with that task. The following section provides an overview of the tools to package and install components into your Kubernetes cluster.

Packaging and installing Kubernetes applications

There are different tools to package and manage your Kubernetes applications. Most of the time, we can separate these tools into two main categories: templating engines and package managers. You will probably need both kinds of tools for real-life scenarios to get things done. Let's discuss these two kinds of tools: why would you need a templating engine? What kind of packages do you want to manage?

A templating engine allows you to reuse the same resource definitions in different environments where applications might require slightly different parameters. The textbook example of the need to template your resources is database URLs. If your service needs to connect to different database instances in different environments, such as the testing database in the testing environment and the production database in the production environment, you want to avoid maintaining two copies of the same YAML file but with different URLs. Figure 1.2 shows how you can now add variables to the YAML files, and the engine will then find and replace these variables with different values depending on where you want to use the final (rendered) resource.

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

- *Kustomize:* https://kustomize.io/

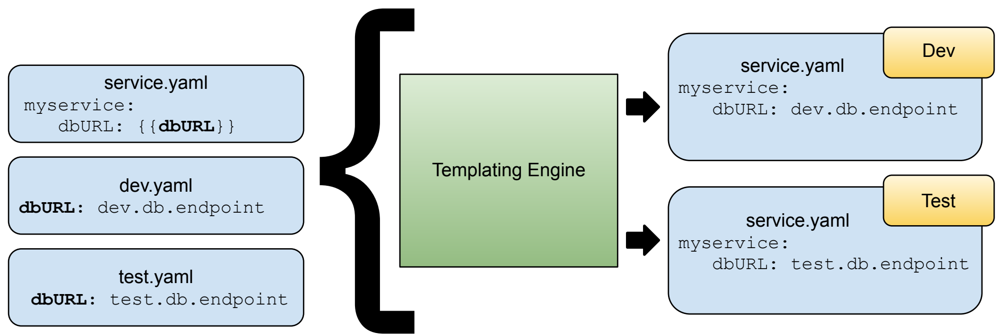


Figure 1.2 Templating engines render YAML resources by replacing variables.

- *Carvel YTT:* https://carvel.dev/ytt/
- *Helm Templates:* https://helm.sh/docs/chart_best_practices/templates/#helm

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

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

From a package provider's point of view, a package manager should offer a convention and structure to create packages and a tool to package the files you want to distribute. Package managers deal with versions and dependencies, meaning that if you create a package, you must associate a version number with it. Some package managers use the

semver (semantic versioning) approach, which uses three numbers to describe the package maturity (1.0.1 where these numbers represent the major, minor, and patch versions). A package manager doesn't need to provide a centralized package repository, but they often do. This package repository is in charge of hosting packages for users to consume. Central repositories are useful because they provide access to developers with thousands of packages ready to be used. Some examples of these central repositories are Maven Central, NPM, Docker Hub, GitHub Container Registry, etc. These repositories are in charge of indexing the package's metadata (which can include versions, labels, dependencies, and short descriptions) to make them searchable by users. These repositories also deal with access control to have public and private packages, but at the end of the day, the main responsibility of the package repository is to allow package producers to upload packages and package consumers to download packages from them (see figure 1.3).

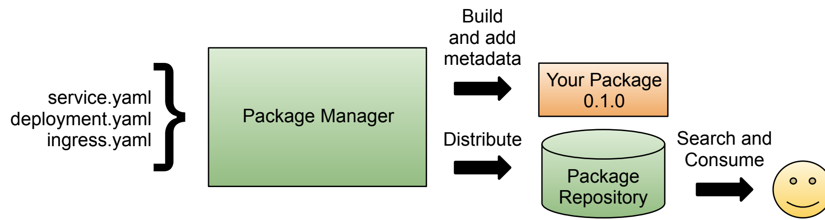


Figure 1.3 Package Managers' responsibilities: build, package, and distribute

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

- Imgpkg (https://carvel.dev/imgpkg/), which uses Container registries to store the packages.
- Kapp (https://carvel.dev/kapp/), which provides higher-level abstractions to group resources as applications.
- Tools like Terraform and Pulumi that allow you to manage infrastructure as code.

In the following section, we will look at using Helm (http://helm.sh) to install the Conference application into our Kubernetes cluster.

1.2 Installing the Conference application with a single command

Let's install the Conference application into our Kubernetes cluster using Helm. This Conference application allows conference organizers to receive proposals from potential speakers, evaluate these proposals, and keep an updated agenda with the approved submissions for the event. We will use this application throughout the book to exemplify the challenges that you will face while building real-life applications.

NOTE For the complete list of steps, follow the step-by-step tutorial located at https://github.com/salaboy/platforms-on-k8s/tree/main/chapter-2. It includes all the prerequisites to run the commands described in this section, such as creating a cluster and installing the command-line tools needed for the examples to work.

This application was built as a walking skeleton, which means it is not a complete application but has all the pieces required for the "Call for Proposals" flow to work. These services can be iterated further to support other flows and real-life scenarios. In the following sections, you will install the application into the cluster and interact with it to see how it behaves when it runs on top of Kubernetes. Let's install the application with the following line:

```
helm install conference oci://docker.io/salaboy/conference-app --version
v1.0.0
```

You should see the output similar to listing 1.1.

Listing 1.1 Helm installed the chart conference-app version 1.0.0

```
> helm install conference oci://docker.io/salaboy/conference-app --version
➥v1.0.0
Pulled: registry-1.docker.io/salaboy/conference-app:v1.0.0
Digest:
sha256:e5dd1a87a867fd7d6c6caecef3914234a12f23581c5137edf63bfd9add7d5459
NAME: conference
LAST DEPLOYED: Mon Jun 26 08:19:15 2023
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Cloud-Native Conference Application v1.0.0
Chart Deployed: conference-app - v1.0.0
Release Name: conference
For more information visit: https://github.com/salaboy/platforms-on-k8s
Access the Conference Application Frontend by running
➥'kubectl port-forward svc/frontend -n default 8080:80'
```

NOTE Since Helm 3.7+ you can package and distribute Helm Charts as OCI container images, the URL for the Helm Chart contains oci:// because this chart is hosted in Docker Hub, where the application containers are stored. Before Helm supported OCI images, you needed to manually add and fetch packages from a Helm Chart Repository, which used tar files to distribute these charts.

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

```
helm list
```

The output should look like figure 1.4.

```
chapter-2 — -zsh — 138×6
salaboy@salaboys-MacBook-Pro chapter-2 % helm list
NAME            NAMESPACE       REVISION        UPDATED                                 STATUS          CHART                   APP VERSION
conference      default         1               2023-10-02 17:50:54.019418 +0800 CST    deployed        conference-app-v1.0.0   v1.0.0
salaboy@salaboys-MacBook-Pro chapter-2 %
```

Figure 1.4 List Helm releases

> **NOTE** If instead of using `helm install` you run `helm template oci://docker.io/salaboy/conference-app --version v1.0.0`. Helm will output the YAML files, which will apply against the cluster. There are situations where you might want to do that instead of `helm install`, for example, if you want to override values that the Helm Charts don't allow you to parameterize or apply any other transformations before sending the request to Kubernetes.

1.2.1 Verifying that the application is up and running

Once the application is deployed, containers will be downloaded to run on your laptop, which can take a while. Depending on your internet connection, the process can take up to 10 minutes because Kafka, PostgreSQL, and Redis will be downloaded alongside the application's containers. The RESTARTS columns show how often the container has been restarted due to an error. In distributed applications, this is normal, as components might depend on each other, and when they are started at the same time, connections can fail. By design, applications should be able to recover from problems, and Kubernetes will automatically restart a failing container.

You can monitor the progress by listing all the pods running in your cluster, once again, using the `-owide` flag to get more information:

```
kubectl get pods -owide
```

The output should look like figure 1.5.

```
chapter-2 — -zsh — 164×11
salaboy@salaboys-MacBook-Pro chapter-2 % kubectl get pods -owide
NAME                                                       READY   STATUS    RESTARTS        AGE     IP            NODE          NOMINATED NODE   READINESS GATES
conference-agenda-service-deployment-746bddcdb4-5nzsn      1/1     Running   1 (4h43m ago)   4h44m   10.244.1.5    dev-worker2   <none>           <none>
conference-c4p-service-deployment-67bc96879c-cjknm         1/1     Running   4 (4h43m ago)   4h44m   10.244.3.7    dev-worker3   <none>           <none>
conference-frontend-deployment-797fc494c4-hgjv4            1/1     Running   1 (4h43m ago)   4h44m   10.244.3.6    dev-worker3   <none>           <none>
conference-kafka-0                                         1/1     Running   0               4h44m   10.244.2.12   dev-worker    <none>           <none>
conference-notifications-service-deployment-f76c4578f-8brqb 1/1    Running   2 (4h43m ago)   4h44m   10.244.2.10   dev-worker    <none>           <none>
conference-postgresql-0                                    1/1     Running   0               4h44m   10.244.1.9    dev-worker2   <none>           <none>
conference-redis-master-0                                  1/1     Running   0               4h44m   10.244.1.8    dev-worker2   <none>           <none>
salaboy@salaboys-MacBook-Pro chapter-2 %
```

Listing 1.5 Listing application pods

Something that you might notice in the list of pods is that we are not only running the application's services, but we are also running Redis, PostgreSQL, and Kafka, because the C4P (Call for Proposals) and Agenda services need persistent storage. The application will be using Kafka to exchange asynchronous messages between services. Besides the services, we will have these two databases and a message broker (Kafka) running inside our Kubernetes cluster.

In the output shown in figure 1.5 you need to pay attention to the READY and STATUS columns, where 1/1 in the READY column means that one replica of the container is running, and one is expected to be running. As you can see the RESTART column is showing 7 for the Call for Proposals Service (`conference-c4p-service`). This is because the service depends on Redis to be up and running for the service to be able to connect to it. While Redis is bootstrapping the application will try to connect, and if it fails, it will try to keep reconnecting. As soon as Redis is up, the service will connect to it. The same applies to Kafka and PostgreSQL. To quickly recap, our application services, the databases, and the message broker that we are running are shown in figure 1.6.

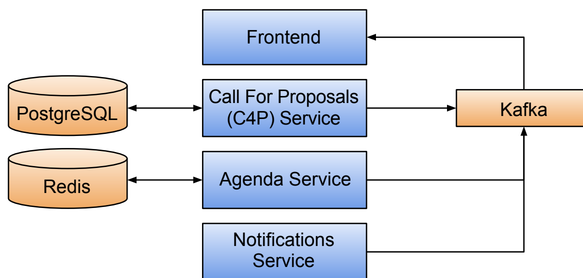


Figure 1.6 Application services, databases, and message broker

Notice that Pods can be scheduled in different nodes. You can check this in the `NODE` column; this is Kubernetes efficiently using the cluster resources. If all the Pods are up and running, you've made it! The application is now up and running, and you can access it by pointing your favorite browser to `http://localhost`.

If you are interested in Helm and building your own Conference application Helm Chart, I recommend you to check the source code provided with the tutorials: https://github.com/salaboy/platforms-on-k8s/tree/main/conference-application/helm/conference-app.

1.2.2 Interacting with your application

In the previous section, we installed the application into our local Kubernetes cluster. In this section, we will quickly interact with the application to understand how the services interact to accomplish a simple use case: Receiving and approving proposals. Remember that you can access the application by pointing your browser to `http://localhost`. The Conference application should look like figure 1.7.

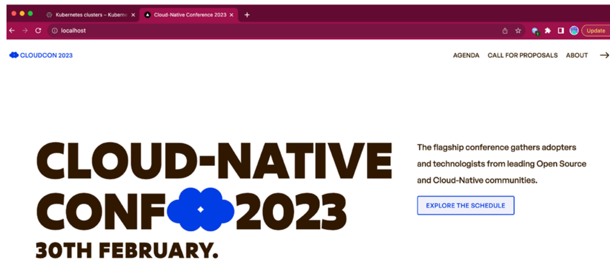


Figure 1.7 Conference landing page

If you switch to the Agenda section now, you should see something like figure 1.8.

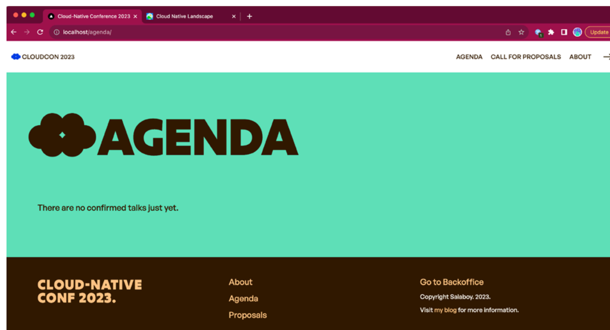


Figure 1.8 Conference empty Agenda when we first install the application

The application's Agenda page lists all the talks scheduled for the conference. Potential speakers can submit proposals that the conference organizers will review. When you start the application for the first time, there will be no talks on the agenda, but you can now go ahead and submit a proposal from the Call from Proposals section. Check figure 1.9.

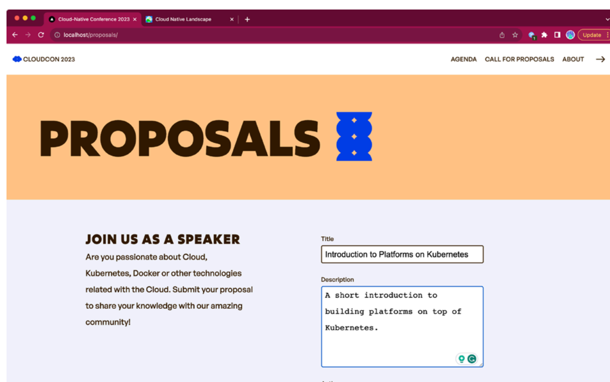


Figure 1.9 Submitting a proposal for organizers to review

Notice that there are four fields (Title, Description, Author, and Email) in the form to submit a proposal. Fill in all the fields and submit by clicking the Submit Proposal button at the bottom of the form. The organizers will use this information to evaluate your proposal and get in touch with you via email if your proposal gets approved or rejected. Once the proposal is submitted, you can go to the Back Office (click the arrow pointing to the right at the top menu) and check the Review Proposals tab, where you can Approve or Reject submitted proposals. You will be acting as a conference organizer on this screen; see figure 1.10.

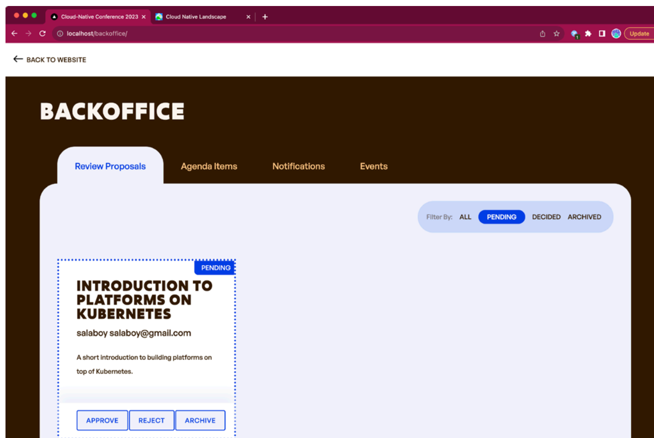


Figure 1.10 Conference organizers can Accept or Reject incoming proposals

Approved proposals will appear on the Main Agenda page. Attendees who visit the page at this stage can glance at the conference's main speakers. Figure 1.11 shows our freshly approved proposal in the Agenda section of the main conference page.

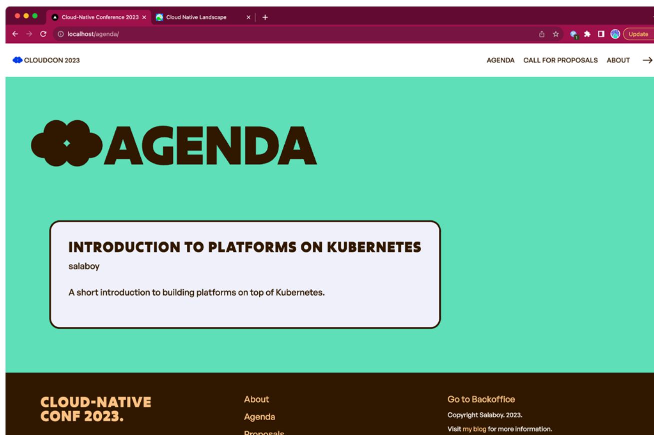


Figure 1.11 Your proposal is now live on the agenda!

At this stage, the potential speaker should have received an email about the approval or rejection of their proposal. You can check this by looking at the notification service logs, using `kubectl` from your terminal; see figure 1.12 for the output of the command:

```
kubectl logs -f conference-notifications-service -deployment-<POD_ID>
```

```
chapter-2 — kubectl logs -f conference-notifications-service-deployment-f76c4578f-8brqb — 152×25
2023/10/02 14:39:29 "GET http://10.244.2.10:8080/health/readiness HTTP/1.1" from 10.244.2.1:45332 - 200 12B in 134.834µs
2023/10/02 14:39:30
 To: salaboy@gmail.com
 Subject: Your proposal  Introduction to Platforms on Kubernetes was accepted
 Body: We are happy to inform that your proposal: `53254302-a4b1-4066-9784-07fc0c1136f0` with title: `Introduction to Platforms on Kubernetes` was accep
ted
         We will send further instructions closer to your presentation date.
         You session has been published into the conference website: e9a89841-16f5-4b82-bd81-1a19a81f7792

         Thanks and see you soon.

         - The Conference Organizers -

2023/10/02 14:39:31 "POST http://notifications-service.default.svc.cluster.local/notifications/ HTTP/1.1" from 10.244.3.7:35892 - 200 744B in 1.0107265s
2023/10/02 14:39:31 Notification Sent - Event emitted to Kafka: {a2df2af6-e5ab-46db-8e69-49296d89ef06 53254302-a4b1-4066-9784-07fc0c1136f0 e9a89841-16f5
-4b82-bd81-1a19a81f7792 Introduction to Platforms on Kubernetes salaboy@gmail.com true Your proposal  Introduction to Platforms on Kubernetes was accept
ed We are happy to inform that your proposal: `53254302-a4b1-4066-9784-07fc0c1136f0` with title: `Introduction to Platforms on Kubernetes` was accepted
         We will send further instructions closer to your presentation date.
         You session has been published into the conference website: e9a89841-16f5-4b82-bd81-1a19a81f7792

         Thanks and see you soon.

         - The Conference Organizers -
}
2023/10/02 14:39:39 "GET http://10.244.2.10:8080/health/liveness HTTP/1.1" from 10.244.2.1:45710 - 200 12B in 60.792µs
```

Figure 1.12 Notifications service logs (emails and events)

These logs show two important aspects of the application. First, notifications are sent via emails to potential speakers. The organizers need to keep track of these communications. On the conference Back Office page, you can find the Notifications tab, where the content of the notifications is shown to the organizers (see figure 1.13).

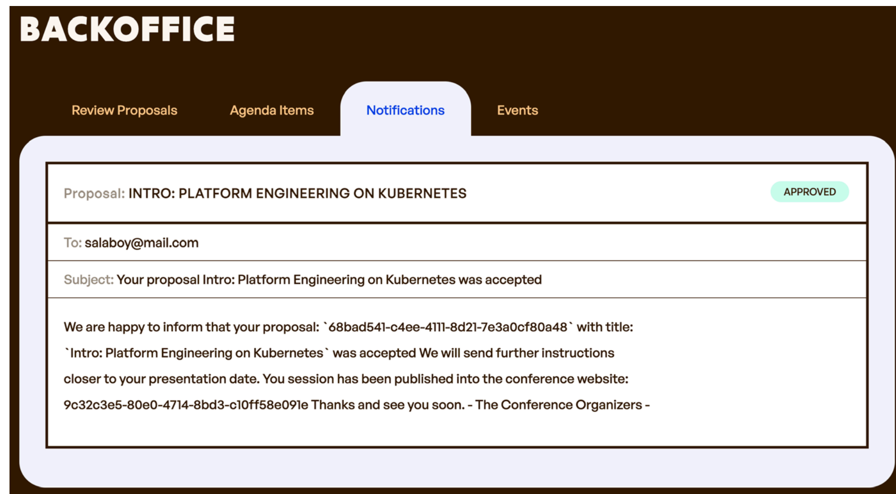


Figure 1.13 Notifications displayed in the Back Office

The second aspect displayed here is Events. All services from this application are emitting events when relevant actions are performed. The notification service is emitting an event, in this case to Kafka, for every notification that is being sent. This allows other services and applications to integrate with the application services asynchronously. Figure 1.14 shows the Events section of the Back Office.

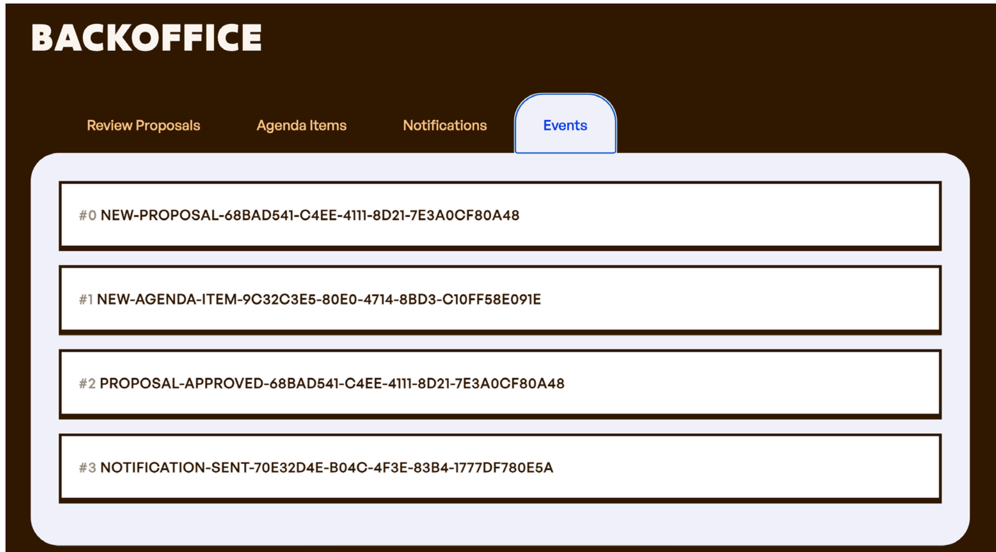


Figure 1.14 All service events in the Back Office

Figure 1.14 shows all events emitted by the application services; notice that you can see all the meaningful operations being performed by the services to fulfill the Call for Proposals flow (New Proposal > New Agenda Item > Proposal Approved > Notification Sent).

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

In this section, you installed the Conference application using Helm. Then we verified that the application is up and running and that potential speakers can submit proposals, while conference organizers can approve or reject these proposals. The decisions will send notifications to potential speakers via email.

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

1.3 Inspecting the walking skeleton

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

Usually, instead of just looking at the pods running (with `kubectl get pods`), to understand and operate the application, you will be looking at services and deployments. Let's explore the deployment resources first.

1.3.1 Kubernetes deployments basics

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

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

First, let's list all the available deployments with:

```
kubectl get deployments
```

The output should look like listing 1.2.

Listing 1.2 Listing your application's deployments

```
NAME                                         READY   UP-TO-DATE   AVAILABLE
conference-agenda-service-deployment         1/1     1            1
conference-c4p-service-deployment            1/1     1            1
conference-frontend-deployment               1/1     1            1
conference-notifications-service-deployment  1/1     1            1
```

1.3.2 *Exploring deployments*

In the following example, you will describe the Frontend deployment. You can describe each deployment in more detail with `kubectl describe deploy conference-frontend-deployment` (see listing 1.3).

Listing 1.3 Describing a deployment to see its details

```
> kubectl describe deploy conference-frontend-deployment
Name:                   conference-frontend-deployment
Namespace:              default
CreationTimestamp:      Tue, 27 Jun 2023 08:21:21 +0100
Labels:                 app.kubernetes.io/managed-by=Helm
Annotations:            deployment.kubernetes.io/revision: 1
                        meta.helm.sh/release-name: conference
                        meta.helm.sh/release-namespace: default
Selector:               app=frontend
Replicas:               1 desired | 1 updated | 1 total | 1 available
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=frontend
  Containers:
   frontend:
    Image:      salaboy/frontend-go...
    Port:       8080/TCP
    Host Port:  0/TCP
    ...
    ...
    Environment:
      AGENDA_SERVICE_URL:         agenda-service.default.svc.cluster.local
      C4P_SERVICE_URL:            c4p-service.default.svc.cluster.local
      NOTIFICATIONS_SERVICE_URL:  notifications-service.default.svc.cluster.
local
      KAFKA_URL:                  conference-kafka.default.svc.cluster.local
      POD_NODENAME:                (v1:spec.nodeName)
      POD_NAME:                    (v1:metadata.name)
      POD_NAMESPACE:               (v1:metadata.namespace)
      POD_IP:                      (v1:status.podIP)
      POD_SERVICE_ACCOUNT:         (v1:spec.serviceAccountName)
    Mounts:                     <none>
  Volumes:                      <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   conference-frontend-deployment-<ID> (1/1 replicas created)
```

Shows the replicas available for this deployment. This gives you a quick indication about the state of your deployment.

The container image, including the name and tag used for this service.

The environment variables used to configure this container.

Events shows us relevant information about our Kubernetes resources—in this case, when the replica was created.

```
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  48m   deployment-controller  Scaled up replica
➥ set conference-frontend-deployment-59d988899 to 1
```

Listing 1.3 shows that describing deployments in this way is very helpful if for some reason the deployment is not working as expected. For example, if the number of replicas required is not met, describing the resource will give you insights into where the problem might be. Always check at the bottom for the events associated with the resource to get more insights about the resource status. In this case, the deployment was scaled to have one replica 48 minutes ago.

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

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

1.3.3 ReplicaSets

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

```
kubectl get replicaset
```

The output should look like listing 1.4.

Listing 1.4 Listing the deployment's ReplicaSets

```
> kubectl get replicasets
NAME                                              DESIRED   CURRENT   READY
conference-agenda-service-deployment-7cc9f58875         1         1       1
conference-c4p-service-deployment-76dfc94444            1         1       1
conference-frontend-deployment-59d988899                1         1       1
conference-notifications-service-deployment-7cbcb8677b 1         1       1
```

These `ReplicaSet` objects are fully managed by the deployment's resource, and usually, you shouldn't need to deal with them. ReplicaSets are also essential when dealing with rolling updates, and you can find more information about this topic at https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/. You will be performing updates to the application with Helm in later chapters, where these mechanisms will kick in.

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

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

You can try this out with the Frontend deployment:

```
> kubectl scale --replicas=2 deployments/conference-frontend-deployment
```

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

```
conference-frontend-deployment-<ID>-8gpgn  1/1     Running    7 (53m ago)    59m
conference-frontend-deployment-<ID>-z4c5c  1/1     Running    0              13s
```

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

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

```
kubectl set env deployment/conference-frontend-deployment
➥FEATURE_DEBUG_ENABLED=true
```

Notice that when you change the deployment object configuration (anything inside `spec.template.spec` block) the rolling update mechanism of the Deployment resource will kick in. All the existing pods managed by this deployment will be upgraded to have the new specification (in this example to include the new `FEATURE_DEBUG_ENABLED` environment variable). This upgrade, by default, will start a new pod with the new specification and wait for it to be ready before terminating the old version of the pod. This process will be repeated until all the pods (replicas for the deployment) are using the new configuration.

If you access the application again in your browser (you might need to access using Incognito Mode if the browser cached the website), in the Back Office section, there is a

new Debug tab. You can see the Pod Name, Pod IP, the Namespace, and the Node name where the Pod is running for all services (figure 1.15).

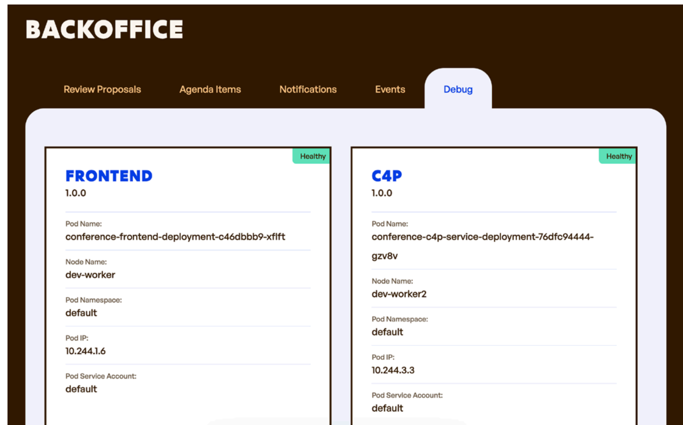


Figure 1.15 First replica of the Frontend answering your request (running on Node Name: dev-worker)

If you wait for 3 seconds, the page will automatically refresh, and you should see the second replica answering this time, if not wait for the next cycle (figure 1.16).

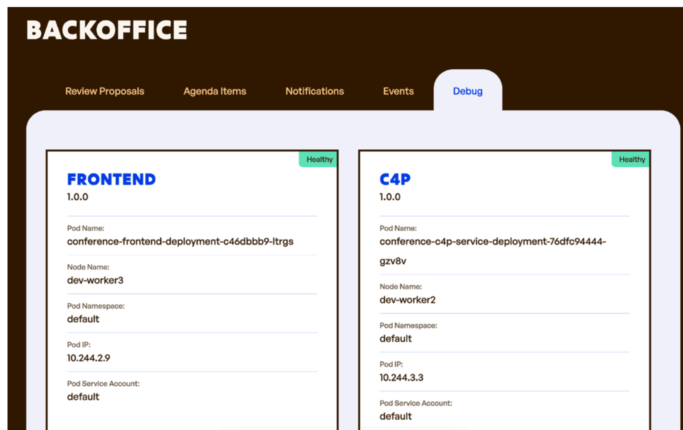


Figure 1.16 Second replica of the Frontend answering your request (running on Node Name: dev-worker3)

By default, Kubernetes will load-balance the requests between the replicas. Being able to scale by just changing the number of replicas, there is no need to deploy anything

new, Kubernetes will provision a new pod (with a new container in it) to deal with more traffic. Kubernetes will also make sure that there is the amount of desired replicas at all times. You can test this by deleting one pod and watching how Kubernetes recreates it automatically. For this scenario, you need to be careful, because the web application frontend is executing several requests to fetch the HTML, CSS, and JavaScript libraries; hence, each of these requests can land in a different replica.

1.3.4 Connecting services

We have looked at deployments, which are in charge of getting our containers up and running and keeping them that way, but so far, these containers can only be accessed inside the Kubernetes cluster. If we want other services to interact with these containers, we need to look at another Kubernetes resource called `Service`. Kubernetes provides an advanced service-discovery mechanism that allows services to communicate with each other by just knowing their names. This is essential for connecting many services without knowing IP addresses of Kubernetes pods that can change over time, as they can be upgraded, rescheduled to a different node, or just restarted with a new IP address when something goes wrong.

1.3.5 Exploring services

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

```
kubectl get services
```

After running the command, you should see something like listing 1.5.

Listing 1.5 Listing application's services

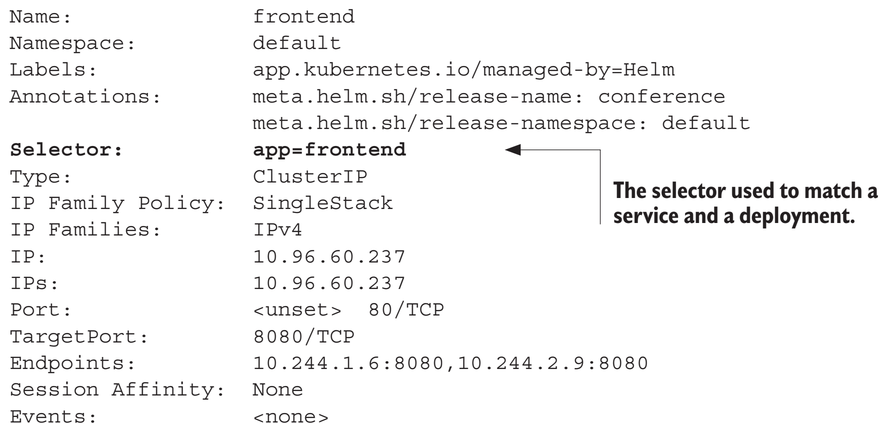
```
NAME                         TYPE        CLUSTER-IP      PORT(S)
agenda-service               ClusterIP   10.96.90.100    80/TCP
c4p-service                  ClusterIP   10.96.179.86    80/TCP
conference-kafka             ClusterIP   10.96.67.2      9092/TCP
conference-kafka-headless    ClusterIP   None      9092/TCP,9094/TCP,9093/TCP
conference-postgresql        ClusterIP   10.96.121.167   5432/TCP
conference-postgresql-hl     ClusterIP   None            5432/TCP
conference-redis-headless    ClusterIP   None            6379/TCP
conference-redis-master      ClusterIP   10.96.225.138   6379/TCP
frontend                     ClusterIP   10.96.60.237    80/TCP
kubernetes                   ClusterIP   10.96.0.1       443/TCP
notifications-service        ClusterIP   10.96.65.248    80/TCP
```

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

```
kubectl describe service frontend
```

This should give you something like we see in listing 1.6. Services and deployments are linked by the Selector property, highlighted in the following image. In other words, the service will route traffic to all the pods created by a deployment containing the label `app=frontend`.

Listing 1.6 Describing the Frontend service



```
Name:              frontend
Namespace:         default
Labels:            app.kubernetes.io/managed-by=Helm
Annotations:       meta.helm.sh/release-name: conference
                   meta.helm.sh/release-namespace: default
Selector:          app=frontend
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.96.60.237
IPs:               10.96.60.237
Port:              <unset>  80/TCP
TargetPort:        8080/TCP
Endpoints:         10.244.1.6:8080,10.244.2.9:8080
Session Affinity:  None
Events:            <none>
```


1.3.6 Service discovery in Kubernetes

By using services, if your application service needs to send a request to any other service, it can use the Kubernetes service's name and port; in most cases, you can use port 80 if you are using HTTP requests, so you only need to use the service name. If you look at the source code of the services, you will see that HTTP requests are created against the service name; no IP addresses or Ports are needed.

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

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

```
kubectl get ingress
```

The output should look like listing 1.7.

Listing 1.7 Listing the application's Ingress resources

```
NAME                         CLASS   HOSTS   ADDRESS     PORTS   AGE
conference-frontend-ingress  nginx   *       localhost   80      84m
```

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

```
kubectl describe ingress conference-frontend-ingress
```

You should expect the output to look like listing 1.8.

Listing 1.8 Describing the Ingress resource

```
Name:              conference-frontend-ingress
Labels:            app.kubernetes.io/managed-by=Helm
Namespace:         default
Address:           localhost
Ingress Class:     nginx
Default backend:   <default>
Rules:
  Host        Path  Backends
  ----        ----  --------
  *
              /   frontend:80 (10.244.1.6:8080,10.244.2.9:8080)
Annotations:  meta.helm.sh/release-name: conference
              meta.helm.sh/release-namespace: default
              nginx.ingress.kubernetes.io/rewrite-target: /
Events:       <none>
```

All traffic going to '/' will go to the frontend:80 service.

As you can see, Ingress also uses the service's name to route traffic. For this to work, you need an Ingress controller, like we installed when we created the KinD cluster. If you are running in a cloud provider, you might need to install an Ingress controller.

The following spreadsheet is a community resource created to keep track of the different options of Ingress controllers that are available for you to use: http://mng.bz/K9Bn.

With Ingresses, you can configure a single entry-point and use path-based routing to redirect traffic to each service you need to expose. The previous Ingress resource in listing 2.8 routes all the traffic sent to / to the `frontend` service. Notice that Ingress rules are pretty simple, and you shouldn't add any business logic routing at this level.

1.3.7 Troubleshooting internal services

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

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

You should see the following output (listing 1.9) and make sure that you don't kill the command.

Listing 1.9 kubectl port-forward allows you to expose a service for debugging purposes

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

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

Listing 1.10 curl localhost:8080 to access Agenda service using port-forward

```
> curl -s localhost:8080/service/info | jq --color-output
{
  "Name": "AGENDA",
  "Version": "1.0.0",
  "Source": "https://github.com/salaboy/platforms-on-k8s/tree/main/
            conference-application/agenda-service",
  "PodName": "conference-agenda-service-deployment-7cc9f58875-28wrt",
  "PodNamespace": "default",
  "PodNodeName": "dev-worker3",
  "PodIp": "10.244.2.2",
  "PodServiceAccount": "default"
}
```

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

For everyday operations, the `kubectl` command line tool might not be optimal, and different dashboards can be used to explore and manage your Kubernetes workloads, such as k9s (https://k9scli.io/), the Kubernetes dashboard (https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/) and Skooner (https://github.com/skooner-k8s/skooner).

1.4 Cloud-native application challenges

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

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

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

- *Downtime is not allowed*: If you are building and running a cloud-native application on top of Kubernetes, and you are still suffering from application downtime, then you are not capitalizing on the advantages of the technology stack that you are using.
- *Service's built-in resiliency*: Downstream services will go down, and you need to ensure that your services are prepared for that. Kubernetes helps with dynamic service discovery, but that is not enough for your application to be resilient.
- *Dealing with the application state is not trivial*: We must understand each service's infrastructural requirements to allow Kubernetes to scale up and down our services efficiently.
- *Inconsistent data*: A common problem of working with distributed applications is that data is not stored in a single place and tends to be distributed. The application will need to be ready to deal with cases where different services have different views of the state of the world.
- *Understanding how the application is working (monitoring, tracing, and telemetry)*: Having a clear understanding of how the application is performing and that it is doing what it is supposed to be doing is essential for quickly finding problems when things go wrong.
- *Application security and identity management*: Dealing with users and security is always an afterthought. For distributed applications, having these aspects clearly documented and implemented early on will help you to refine the application requirements by defining "who can do what and when."

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

1.4.1 Downtime is not allowed

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

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

```
kubectl scale --replicas=2 deployments/conference-frontend-deployment
```

If one of the replicas stops running or breaks for any reason, Kubernetes will try to start another one to ensure that two replicas are up all the time. Figure 1.17 shows two Frontend replicas serving traffic to the user.

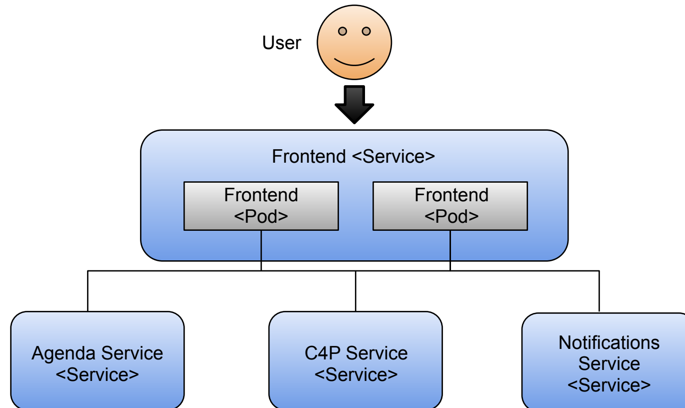


Figure 1.17 By having two replicas of the Frontend container running, we allow the application to tolerate failures and also to increase the number of concurrent requests that the application can handle.

You can quickly try this self-healing feature of Kubernetes by killing one of the two pods of the application Frontend. You can do this by running the following commands, as shown in listings 1.11 and 1.12.

Listing 1.11 Checking that the two replicas are up and running

```
> kubectl get pods
NAME                                             READY   STATUS    RESTARTS       AGE
conference-agenda-service-deployment-<ID>        1/1     Running   7 (92m ago)    100m
conference-c4p-service-deployment-<ID>           1/1     Running   7 (92m ago)    100m
conference-frontend-deployment-<ID>              1/1     Running   0              25m
conference-frontend-deployment-<ID>              1/1     Running   0              25m
conference-kafka-0                               1/1     Running   0              100m
conference-notifications-service-deployment-<ID> 1/1     Running   7 (91m ago)    100m
conference-postgresql-0                          1/1     Running   0              100m
conference-redis-master-0                        1/1     Running   0              100m
```

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

```
> kubectl delete pod conference-frontend-deployment-c46dbbb9-ltrgs
```

Then list the pods again (listing 1.12).

Listing 1.12 A new replica is automatically created as soon as one goes down

```
> kubectl get pods

NAME                                                READY   STATUS             RESTARTS     AGE
conference-agenda-service-deployment-<ID>           1/1     Running            7 (92m ago)  100m
conference-c4p-service-deployment-<ID>              1/1     Running            7 (92m ago)  100m
conference-frontend-deployment-<NEW ID>             0/1     ContainerCreating 0             1s
conference-frontend-deployment-<ID>                 1/1     Running            0            25m
conference-kafka-0                                  1/1     Running            0            100m
conference-notifications-service-deployment-<ID>   1/1     Running            7 (91m ago)  100m
conference-postgresql-0                             1/1     Running            0            100m
conference-redis-master-0                           1/1     Running            0            100m
```

You can see how Kubernetes (the ReplicaSet, more specifically) immediately creates a new pod when it detects only one running. While this new pod is being created and started, you have a single replica answering your requests until the second one is up and running. This mechanism ensures that at least two replicas answer your users' requests. Figure 1.18 shows that the application still works, because we still have one pod serving requests.

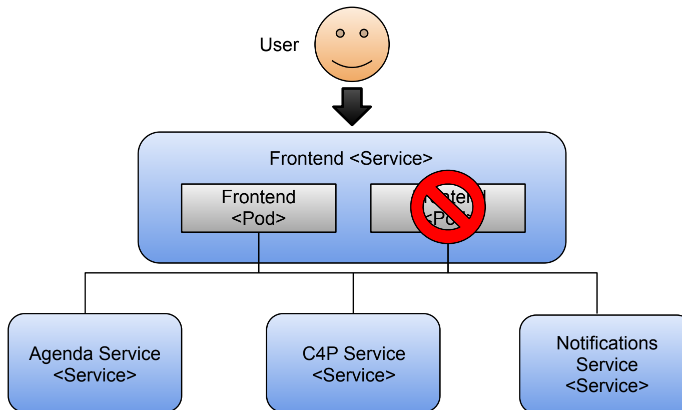


Figure 1.18 If one of the instances fails, Kubernetes will automatically kill and recreate that instance. But at least the other running container can keep answering requests.

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

```
> kubectl scale --replicas=1 deployments/conference-frontend-deployment
```

Go ahead and try this out. Delete only the replica available for the Frontend pod:

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

Figure 1.19 shows the application is not working anymore, because there are no Frontend pods to serve incoming requests from users.

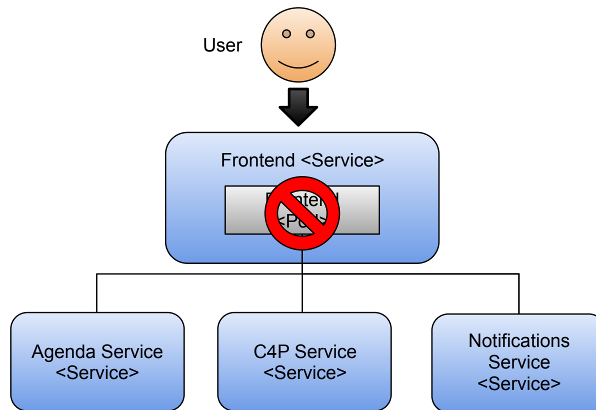


Figure 1.19 With a single replica being restarted, there is no backup to answer user requests. If there is no replica available to serve your users' requests, you will experience downtime. This is exactly what we want to avoid.

After killing the pod, try to access the application by refreshing your browser (http://localhost). You should see "503 Service Temporarily Unavailable" in your browser, because the Ingress controller (not shown in the previous figure for simplicity) cannot find a replica running behind the Frontend service. If you wait for a bit, you will see the application come back up. Figure 1.20 shows the 503 "Service Temporarily Unavailable" being returned by the NGINX Ingress controller component that was in charge of routing traffic to the Frontend service.

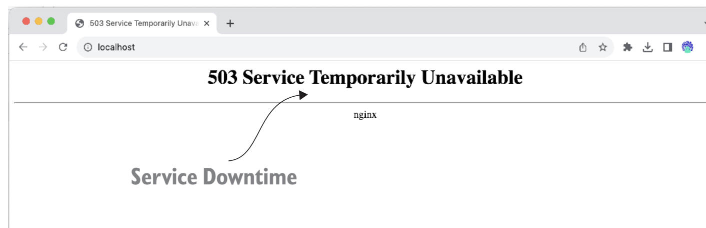


Figure 1.20 With a single replica being restarted, there is no backup to answer user requests

This error message is quite tricky, because the application takes about a second to get restarted and to be fully functional, so if you didn't manage to see it, you can try to downscale the frontend service to zero replicas with `kubectl scale --replicas=0 deployments/conference-frontend-deployment` to simulate downtime.

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

In summary, pay special attention to user-facing services exposed outside of your cluster. Whether they are user interfaces or just APIs, ensure you have as many replicas as needed to deal with incoming requests. Having a single replica should be avoided for most use cases besides development.

1.4.2 Service's resilience built-in

But now, what happens if the other services go down? For example, the Agenda service, is just in charge of listing all the accepted proposals to the conference attendees. This service is also critical, because the Agenda List is right there on the main page of the application. So, let's scale the service down:

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

Figure 1.21 shows how the application can keep working, even if one of the services is misbehaving.

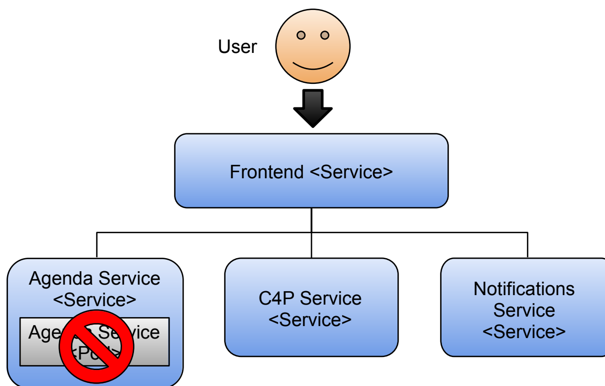


Figure 1.21 No pods for the Agenda service. If a service is failing, the user should be able to keep using the application with limited functionality.

Right after running this command, the container will be killed, and the service will not have any container answering its requests. Try refreshing the application in your browser, you should see a cached response as shown in figure 1.22.

As you can see, the application is still running, but the Agenda service is not available right now. Check the Debug tab in the Back Office section, which should show that the Agenda service is unhealthy (figure 1.23).

You can prepare your application for such scenarios; in this case, the Frontend has a cached response to at least show something to the user. If, for some reason, the Agenda service is down, at least the user will be able to access other services and other sections of the application. From the application perspective, it is important not to propagate

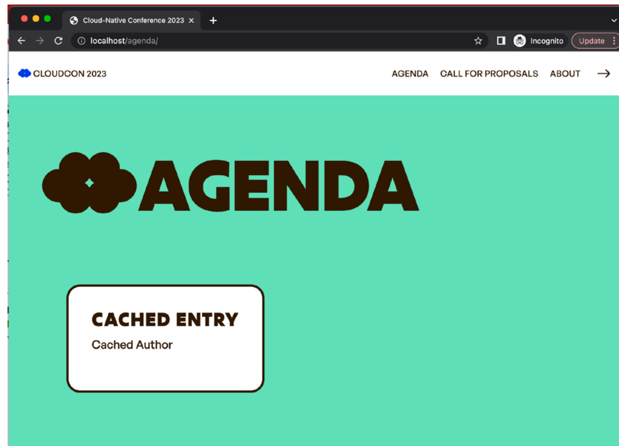


Figure 1.22 If the Agenda service has no replica running, the Frontend is wise enough to show the user some cached entries.

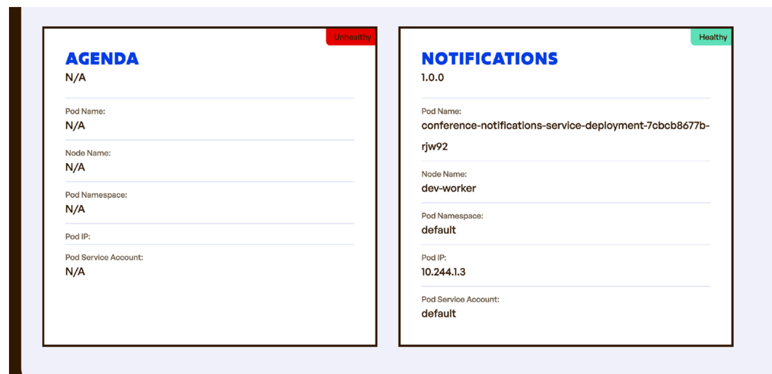


Figure 1.23 If in Debug mode, the Back Office should show the unhealthy services.

the error back to the user. The user should be able to keep using other application services, for example, the Call for Proposals form, until the Agenda service is restored.

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

Let's now switch to talking about dealing with the state in our applications and how it is critical to understand how our application's services handle the state from a scalability

point of view. Since we will be talking about scalability, data consistency is the challenge we will try to solve next.

1.4.3 Dealing with the application state is not trivial

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

```
> kubectl scale --replicas=1 deployments/conference-agenda-service-deployment
```

If you have created proposals before, you will notice that as soon as the Agenda service goes back up, you see the accepted proposals again on the Agenda page. This works only because both the Agenda service and C4P Service store all the proposals and agenda items in external databases (PostgreSQL and Redis). In this context, external means outside of the pod memory. What will happen if we scale the Agenda service up to two replicas? See listing 1.13.

Listing 1.13 Running with two replicas of the Agenda service

```
> kubectl scale --replicas=2 deployments/conference-agenda-service-deployment
NAME                                                READY   STATUS    AGE
conference-agenda-service-deployment-<ID>           1/1     Running   2m30s
conference-agenda-service-deployment-<ID>           1/1     Running   22s
conference-c4p-service-deployment-<ID>              1/1     Running   150m
conference-frontend-deployment-<ID>                 1/1     Running   8m55s
conference-kafka-0                                  1/1     Running   150m
conference-notifications-service-deployment-<ID>    1/1     Running   150m
conference-postgresql-0                             1/1     Running   150m
conference-redis-master-0                           1/1     Running   150m
```

Figure 1.24 shows the Agenda service running two replicas of the service concurrently.

With two replicas dealing with your user requests, now the Frontend will have two instances to query. Kubernetes will do the load balancing between the two replicas, but your application will have no control over which replica the request hits. Because we are using a database to back up the data outside of the pod's context, we can scale the replicas to many pods dealing with the application demand. Figure 1.25 shows how the Agenda service relies on Redis to store the application state, while the Call for Proposals uses PostgreSQL to do the same.

One of the limitations of this approach is the number of database connections that your database supports in its default configuration.

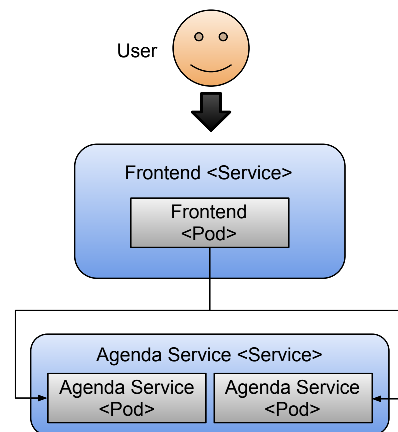


Figure 1.24 Two replicas can now deal with more traffic. The requests being forwarded by the Frontend can be answered by the two available replicas, allowing the application to handle more load.

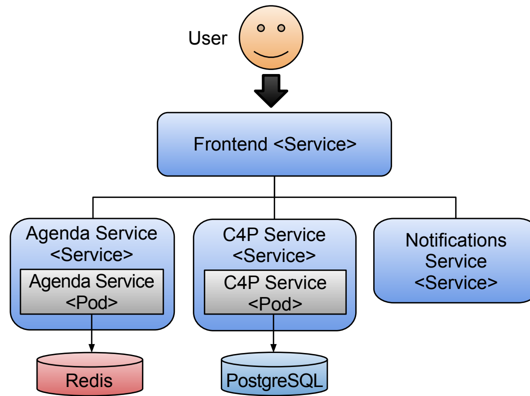


Figure 1.25 Both data-sensitive services use persistent stores. Delegating state storage to external components, make your service stateless and easier to scale.

If you keep scaling up the replicas, always consider reviewing the database connection pool settings to ensure that your database can handle all the connections created by all the replicas. But for the sake of learning, let's imagine that we don't have a database, and our Agenda service keeps all the agenda items in memory. How would the application behave if we started scaling up the Agenda service pods? Figure 1.26 shows the hypothetical case of having in-memory data inside our applications.

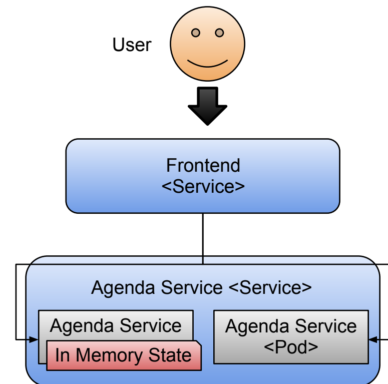


Figure 1.26 What would happen if the Agenda service keeps the state in-memory? If state is kept in memory it is quite hard to share across replicas. This makes scaling the service much harder.

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

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

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

As we have seen before, we have installed Redis and PostgreSQL as part of the application Helm Chart. This is usually not a good idea because databases and tools like message brokers will need special care from the operation team, who can choose not to run these services inside Kubernetes.

1.4.4 Dealing with inconsistent data

Having stored data in a relational data store like PostgreSQL or a NoSQL approach like Redis doesn't solve the problem of having inconsistent data across different stores. Because these stores should be hidden away by the service API, you will need to have mechanisms to check that the data that the services are handling is consistent. In distributed systems, it is quite common to talk about "eventual consistency," meaning that eventually the system will be consistent. Having eventual consistency is better than not having consistency at all. For this example, we can build a simple check mechanism that once in a while (imagine once a day) checks for the accepted talks in the Agenda service to see if they have been approved in the Call for Proposals service. If there is an entry that the Call hasn't approved for the Proposal Service (C4P), then we can raise some alerts or send an email to the conference organizers (figure 1.27).

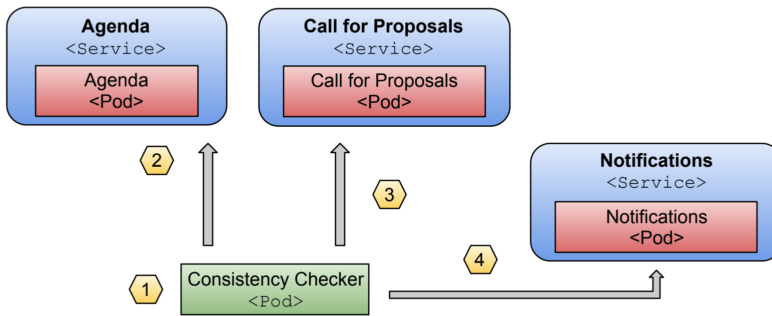


Figure 1.27 Consistency checks can run as CronJobs. We can execute checks against the application services on fixed intervals to make sure that the state is consistent. For example: (1) every day at midnight we query the Agenda Service (2) to verify that the published sessions are approved in the (3) Call For Proposals Service and a corresponding notification has been sent by the (4) Notifications Service.

In figure 1.27, we can see how a CronJob (1) will be executed every X period, depending on how important it is for us to fix consistency problems. Then it will query the Agenda service public APIs (2) to check which accepted proposals are being listed and compare that with the Call for Proposals service approved list (3). Finally, if any inconsistency is found, an email can be sent using the Notifications service public APIs (4).

Think of the simple use case this application was designed for; what other checks would you need? One that immediately comes to mind is verifying that emails were sent correctly for Rejected and Approved proposals. For this use case, emails are really

important, and we need to ensure those emails are sent to our accepted and rejected speakers.

1.4.5 Understanding how the application is working

Distributed systems are complex beasts, and fully understanding how they work from day one can help you save time when things go wrong. This has pushed the monitoring, tracing, and telemetry communities hard to develop solutions that help us understand how things are working at any given time.

The OpenTelemetry community (http://opentelemetry.io) has evolved alongside Kubernetes, and it can now provide most of the tools you will need to monitor how your services are working. As stated on their website, "You can use it to instrument, generate, collect, and export telemetry data (metrics, logs, and traces) for analysis to understand your software's performance and behavior." Figure 1.28 shows a common use case where services all push metrics, traces, and logs to a centralized place that stores and aggregates the information so it can be displayed in dashboards or used by other tools.

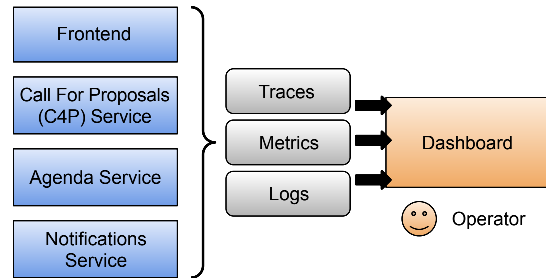


Figure 1.28 Aggregating observability from all our services in a single place reduces the cognitive load on the teams responsible for keeping the application up and running.

It is important to notice that OpenTelemetry focuses on both the behavior and performance of your software, because they will both affect your users and user experience. From the behavior point of view, you want to make sure that the application is doing what it is supposed to do, and by that, you will need to understand which services are calling which other services or infrastructure to perform tasks.

Using Prometheus and Grafana allows us to see the service telemetry and build domain-specific dashboards to highlight certain application-level metrics, for example, the amount of Approved vs. Rejected proposals over time, as shown in figure 1.29.

From the performance point of view, you need to ensure that services are respecting their Service Level Agreements (SLAs), which means that they are taking only a short time to answer requests. If one of your services misbehaves and takes more than usual, you want to know.

For tracing, you must modify your services to understand the internal operations and their performance. OpenTelemetry provides drop-in instrumentation libraries in most languages to externalize service metrics and traces. Figure 1.30 shows the OpenTelemetry architecture, where you can see the OpenTelemetry collector receiving information from each application agent, but also from shared infrastructure components.

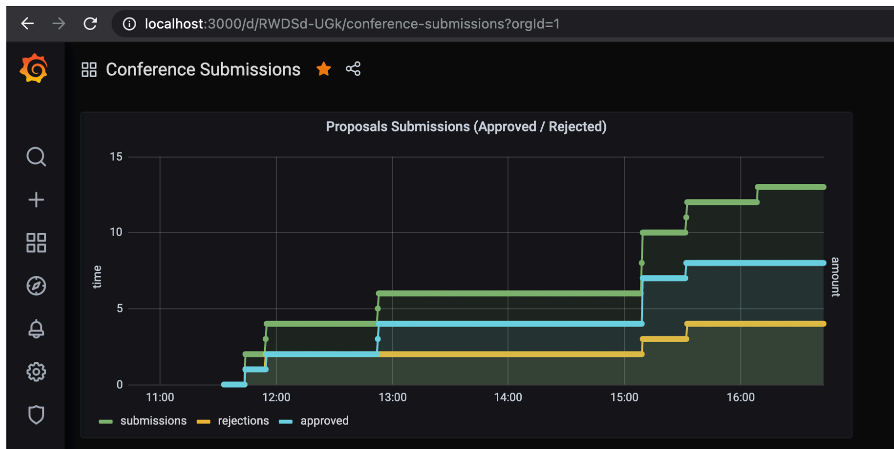


Figure 1.29 Monitoring telemetry data with Prometheus and Grafana

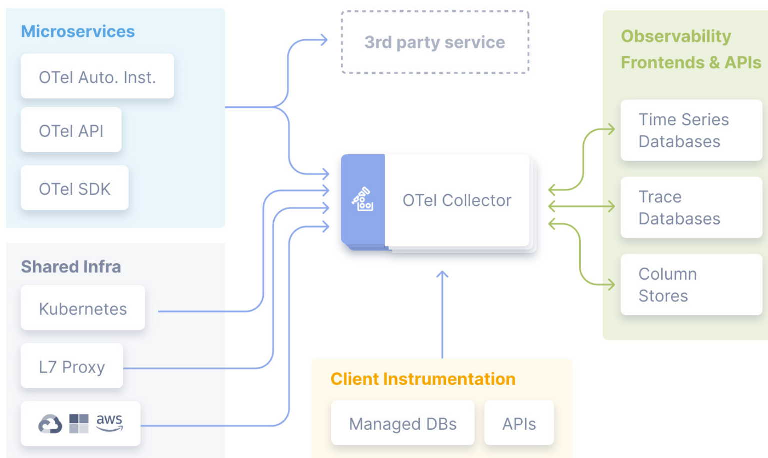


Figure 1.30 OpenTelemetry architecture and library (Source: https://opentelemetry.io/docs/)

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

1.4.6 Application security and identity management

If you have ever built a web application, you know that providing identity management (user accounts and user identity) plus authentication and authorization is quite an

endeavor. A simple way to break any application (cloud-native or not) is to perform actions you are not supposed to do, such as deleting all the proposed presentations unless you are a conference organizer.

This also becomes challenging in distributed systems, because authorization and user identity must be propagated across different services. In distributed architectures, it is quite common to have a component that generates requests on behalf of a user instead of exposing all the services for the user to interact directly. In our example, the Frontend service is this component. Most of the time, you can use this external-facing component as the barrier between external and internal services. For this reason, it is quite common to configure the Frontend service to connect with an authorization and authentication provider commonly using the OAuth2 protocol. Figure 1.31 shows the Frontend service interacting with an identity management service, which is responsible for connecting to an Identity Provider (Google, GitHub, your internal LDAP server) to validate the user credentials as well as to provide roles or group memberships that define what the user can and can't do in different services. The Frontend service handles the login flow (authentication and authorization), but only the context is propagated to the backend services once that is done.

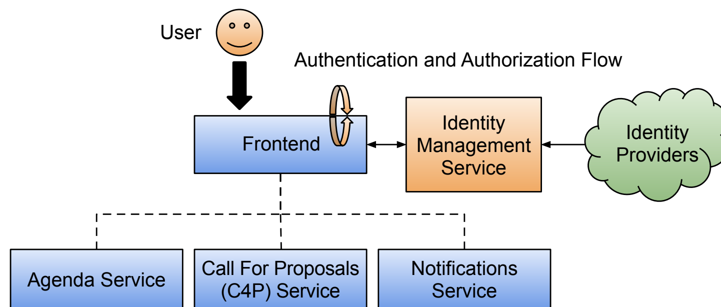


Figure 1.31 Identity management: The Role/Group is propagated to the backend services.

On the identity management front, you have seen that the application doesn't handle users or their data, which is good for regulations such as GDPR. We might want to allow users to use their social media accounts to log in to our applications without the need for them to create separate accounts. This is usually known as social login.

Some popular solutions bring both OAuth2 and identity management together, such as Keycloak (https://www.keycloak.org/) and Zitadel (https://zitadel.com/opensource). These open-source projects provide a one-stop-shop for single sign-on solutions and advanced identity management. In the case of Zitadel, it also provides a managed service that you can use if you don't want to install and maintain an SSO and identity management component inside your infrastructure.

The same is true with tracing and monitoring. If you are planning to have users (and you will probably do, sooner or later), including single sign-on and identity management into the walking skeleton will push you to think about the specifics of "who will be able to do what," refining your use case even more.

1.4.7 Other challenges

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

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

1.5 Linking back to platform engineering

In previous sections, we have covered many topics. We reviewed options for packaging and distributing Kubernetes applications, and then installing our walking skeleton in a Kubernetes cluster using Helm. We tested the application functionality by interacting with it, and finally, we jumped into analyzing common cloud-native challenges that teams will face when building distributed applications.

But you might be wondering how all these topics relate to the title of this book, continuous delivery, and platform engineering in general. In this section, we will make more explicit connections.

First, the intention behind creating a Kubernetes cluster and running an application on top of it was to ensure we cover Kubernetes built-in mechanisms for resilience and scaling up our application services. Kubernetes provides the building blocks to run our applications with zero downtime, even when we are constantly updating them. This allows us, Kubernetes users, to release new versions of our components more frequently, because we are not supposed to stop the entire application from updating one of its parts.

If you are not using the capabilities offered by Kubernetes to keep releasing software in front of your customers, then you need to raise a red flag. Quite often, this can be due to old practices from before Kubernetes that are getting in the way, lack of automation, or not having clearly defined contracts between services that block dependent services from being released independently. We will touch on this topic several times in future chapters because this is a fundamental principle when trying to improve your continuous delivery practice and something that the platform engineering team needs to prioritize.

In this chapter, we have also seen how to install a cloud-native application using a package manager that encapsulates the configuration files required to deploy our application. These configuration files (Kubernetes resources expressed as YAML files) describe our application topology and contain links to the containers used by each

application's service. These YAML files also contain configuration for each service, such as the environment variables to configure each service.

I highly recommend the book *Grokking Continuous Delivery* by Christie Wilson (Manning Publications, 2018) if you want to get more insights into the continuous delivery aspects of how configuration as code can help you deliver more software reliably.

Because I wanted to make sure that you have an application to play around with and because we needed to cover Kubernetes built-in mechanisms, I've made a conscious decision to start with an already packaged application that can be easily deployed into any Kubernetes cluster (no matter if it is running locally or in a cloud provider). We can identify two different phases. One we haven't covered yet is how to produce these packages that can be deployed to any Kubernetes cluster, and the second, which we started playing with, is when we run this application in a concrete cluster (we can consider this cluster an environment, maybe a development environment), as shown in figure 1.32.

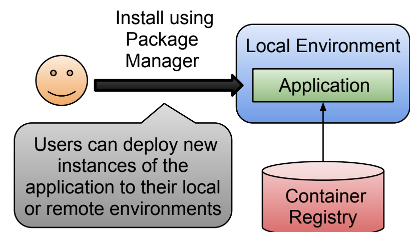


Figure 1.32 Applications' lifecycle from building and packaging to running inside an environment

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

Before moving on, let's be clear that pushing developers to configure application instances might not be the best use of their time. A developer accessing the production environment that users/customers are accessing might also not be optimal. We want to ensure that developers are focused on building new features and improving our application. Figure 1.33 shows how we should be automating all the steps involved in building, publishing, and deploying the artifacts that developers are creating, making sure that they can focus on adding features to the application instead of manually dealing with packaging, distributing, and deploying new versions when they are ready. This is the primary focus of this chapter.

Understanding the tools we can use to automate the path from source code changes to running software in a Kubernetes cluster is fundamental to enabling developers to focus on what they do best "code new features." Another big difference we will tackle is that cloud-native applications are not static. As you can see in the previous diagram, we will not install a static application definition. We want to release and deploy new versions of the services as they become available.

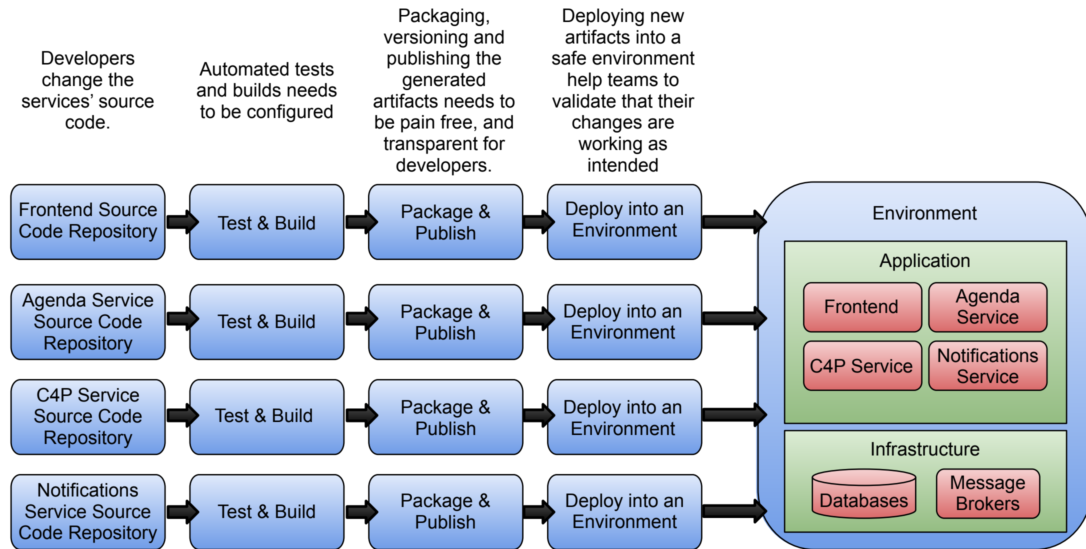


Figure 1.33 Developers can focus on building features, but the platform team needs to automate the entire process after changes are made.

Summary

- Choosing between local and remote Kubernetes clusters requires serious considerations:
  - You can use Kubernetes KinD to bootstrap a local Kubernetes cluster to develop your application. The main drawback is that your cluster is limited by your local resources (CPU and memory) and is not a real cluster of machines.
  - You can have an account in a cloud provider and do all development against a remote cluster. The main drawback of this approach is that most developers are not used to working remotely all the time and that someone needs to pay for the remote resources.
- Package managers, like Helm, help you to package, distribute, and install your Kubernetes applications. In this chapter, you installed an application into a Kubernetes cluster with a single command line.
- Understanding which Kubernetes resources are created by your application gives you an idea about how the application will behave when things go wrong and what extra considerations are needed in real-life scenarios.
- Even with very simple applications, you will face challenges that you will have to tackle one at a time. Knowing these challenges ahead of time helps you to plan and architect your services with the right mindset.
- Having a walking skeleton helps you to try different scenarios and technologies in a controlled environment. In this chapter, you have experimented with:

- Scaling up and down your services to see first-hand how the application behaves when things go wrong.
- Keeping state is hard, and we will need dedicated components to do this efficiently.
- Having at least two replicas for our services minimizes downtime. Making sure that the user-facing components are always up and running guarantees that even when things go wrong, the user will be able to interact with parts of the application.
- Having fallbacks and built-in mechanisms to deal with problems when they arise makes your application more resilient.

- If you have followed the linked step-by-step tutorial, you now have hands-on experience creating a local Kubernetes cluster, installing an application, scaling up and down services, and, most importantly, checking that the application is running as expected.

2 Multi-cloud (app) infrastructure

This chapter covers

- Defining and managing the infrastructure for your cloud-native applications
- Identifying the challenges of managing infrastructure components
- Learning how Crossplane is the Kubernetes way to deal with infrastructure

In previous chapters, we installed a walking skeleton, and we learned how to build each separate component using service pipelines and then how to deploy them into different environments using environment pipelines. We now face a big challenge: dealing with our application infrastructure, meaning running and maintaining not only our application services but also the components that our services need to run. These services expect other components to work correctly, such as databases, message brokers, identity management solutions, email servers, etc. While several tools exist to automate the installation (for on-premises setups) or provisioning of these components in different cloud providers, this chapter will focus on just one that does it in a Kubernetes way. This chapter has three main sections:

- The challenges of dealing with infrastructure
- How to deal with infrastructure using 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, 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 are classified as application infrastructure, which usually involves third-party components not 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 allow your development teams to focus on building the core features of applications, which brings value to the business.

It is essential to distinguish between application infrastructure and hardware infrastructure, because this book is not concerned with hardware provisioning, the reminder of content focus on the application space. 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 automation mechanisms, and you are creating a strong dependency between your business and the 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 requires different expertise to be configured (database administrators for databases, message broker experts, machine learning experts, etc.) 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:* Databases and message brokers are constantly released and patched to improve performance and security. This constant change pushes the operations teams to ensure they can upgrade to newer versions and keep all the data safe without bringing down the entire application. All

this complexity requires a lot of coordination and impact analysis between the teams providing and consuming these components.

- *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 ensure we understand the tradeoffs. The following 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 in the previous chapter, you installed the walking skeleton application containing four services, two databases (Redis and PostgreSQL), and a message broker (Kafka) with a single command.

For our walking skeleton, we are provisioning an instance of a Redis NoSQL database for the Agenda service, an instance of a PostgreSQL database for the Call for Proposals (C4P) service, and an instance of a Kafka cluster, all using Helm Charts. The number of Helm charts available today is impressive, and it is pretty easy to think that installing a Helm Chart will be the way to go. The Helm charts used in the example application can all be found in the Bitnami Helm Chart repositories at https://bitnami.com/stacks/helm.

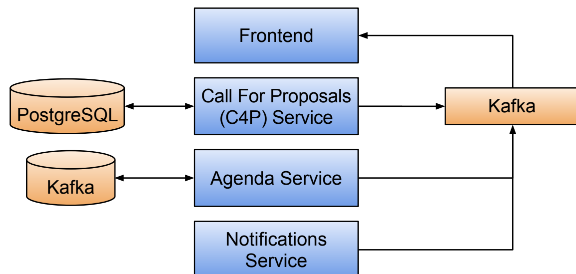


Figure 2.1 Services and their dependencies on application infrastructure components

As discussed in chapter 1, if we want to scale our services that keep state, we must provision specialized components such as databases. Application

developers will define which kind of database will suit them best depending on the data they need to store and how that data will be structured. Figure 2.1 shows the dependency of the application services on some of the application infrastructure components that we have identified for our walking skeleton.

The process of setting up these (PostgreSQL, Redis, and Kafka) components inside your Kubernetes cluster involves the following steps:

- Finding or creating a suitable Helm Chart for the component you want to bootstrap. For the walking skeleton, PostgreSQL (https://bitnami.com/stack/postgresql/helm), Redis (https://bitnami.com/stack/redis/helm), and Kafka (https://bitnami.com/stack/kafka/helm) can be found in the Bitnami Helm Chart repository. If you cannot find a Helm Chart but have a Docker container for the component 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 must set up to accommodate your requirements. Each chart exposes a set of parameters that you can tune for different use cases. Check the chart website to understand what is available. Include your operations teams and DBAs to check the optimal database configurations for your use case; this is not something that a developer can do. This analysis also requires Kubernetes expertise to ensure 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 these components need to be deployed. Helm will then proceed to apply these YAML files to your cluster. For our Conference application Helm Chart that we installed in chapter 1 (section 1.1.3), all the application infrastructure components are added as a dependency to the chart.
- Configure your service to connect to the newly provisioned components. You can achieve this by giving the service the new provisioned instance URL and credentials to connect. 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 shows the steps involved in installing and wiring up these application infrastructure components to our application's services.

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

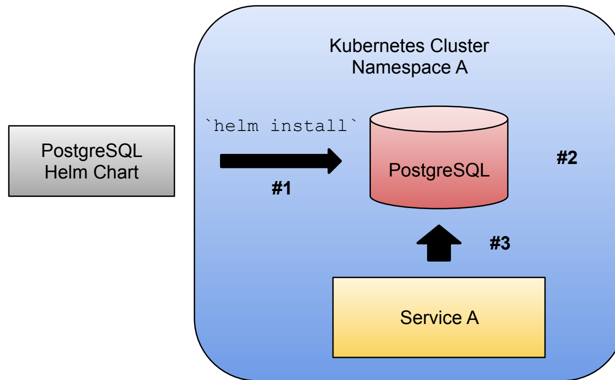


Figure 2.2 Provisioning a new PostgreSQL instance using the PostgreSQL Helm Chart. #1 Install a helm chart into a Namespace inside a Kubernetes Cluster; #2 The chart creates Kubernetes resources such as StatefulSets and Deployments to provision a PostgreSQL instance; #3 A Service needs to connect to the newly created instance, this can be done manually or by referencing a Kubernetes Secret that contains the credentials and details on how to connect.

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, allowing you to upgrade a chart when a new version is available.

- Most charts have 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 (CPU and memory) that might not be available if you use Kubernetes KinD or Minikube on your laptop. Once again, chart documentation usually includes special configurations for development and resource-constrained environments.
- If you are installing a database inside your Kubernetes cluster, each database container (pod) must have access to storage from the underlying Kubernetes node. For databases, you might need a special kind of storage to enable the database to scale elastically, which might require advanced configurations outside of Kubernetes.

For our walking skeleton, for example, we set up the Redis chart to use the `architecture` parameter to `standalone`, (as you can see in the environment pipeline configurations and 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, PostgreSQL, or Kafka instances. We need to provide the services the configurations need to connect while also being conscious of the time needed by these components, such as databases, to start.

Figure 2.3 shows how the wiring usually happens, as most charts automatically create a Kubernetes secret hosting all the details that application's services need to connect.

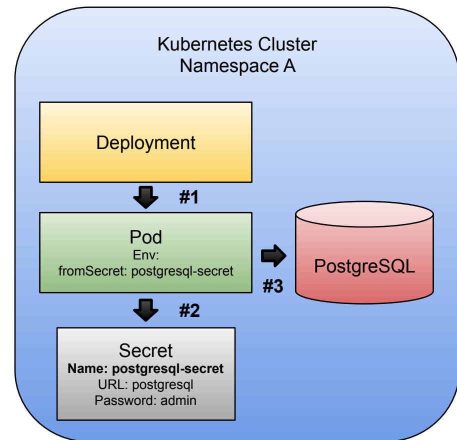


Figure 2.3 Connecting a service to a provisioned resource using secrets. #1 A Kubernetes deployment is created to run one of your services, and the pod template contains the environment variables to configure the pods that this deployment will create; #2 The pod is created using the template specified in the deployment resource, which points to a secret that contains the details to connect to the db instance; #3 The container, which is running inside the pod needs to be prepared to consume the environment variables to connect to the db instance.

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 creates 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 run.

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-c4p-service-deployment`) has the right environment variables (listing 2.1).

Listing 2.1 Environment variables to connect to application infrastructure (PostgreSQL)

```
- name: KAFKA_URL
  value: <KAFKA SERVICE URL>
- name: POSTGRES_HOST
  valueFrom:
    secretKeyRef:
      name: <POSTGRESQL SECRET NAME>
      key: postgres-url
- name: POSTGRES_PASSWORD
  valueFrom:
    secretKeyRef:
      name: <POSTGRESQL SECRET NAME>
      key: postgres-password
```

The bold highlights how we can consume the dynamically generated password when we install the chart and the DB endpoint URL, which is the PostgreSQL Kubernetes

service, also created by the chart. The DB endpoint will be different if you used a different chart release name.

A similar configuration applies to the Agenda service (`conference-agenda-service-deployment`) and Redis (listing 2.2).

Listing 2.2 Environment variables to connect to application infrastructure (Redis)

```
- name: KAFKA_URL
  value: <KAFKA SERVICE URL>
- name: REDIS_HOST
  valueFrom:
    secretKeyRef:
      name: <REDIS SECRET NAME>
      key: redis-url
- name: REDIS_PASSWORD
  valueFrom:
    secretKeyRef:
      name: <REDIS SECRET NAME>
      key: redis-password
```

As before, we extract the password from a Kubernetes secret that will be generated when installing the Redis Helm Chart. The secret name will be derived from the name of the Helm Chart release that we use. The `REDIS_HOST` is obtained from the name of the Kubernetes service that is created by the chart, which depends on the `helm release` name that you used. For all the services of the application we will need to set up the `KAFKA_URL` environment variable so that the services can connect to Kafka. Configuring 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, two databases, and a message broker inside your Kubernetes cluster. Believe it or not, now you are in charge of seven 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 or message brokers.

You might need help with these databases and message brokers depending on how demanding the services are. Imagine 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 shows a typical case of high demand, where we are tempted to increase the number of replicas of our application's services, without checking or changing the configuration of our PostgreSQL instance. In these scenarios, even if the application's

services can scale, the PostgreSQL instance will be the bottleneck if not configured accordingly (to support 200+ concurrent connections).

If you are installing your 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 at https://github.com/zalando/postgres-operator. While these operators are focused on allowing you to provision new instances of PostgreSQL databases, they also implement other features focused on maintenance, for example:

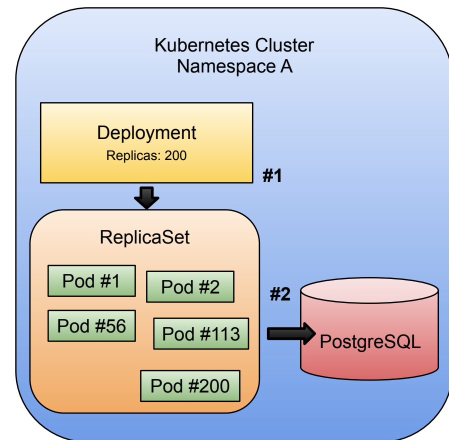


Figure 2.4 Application infrastructure needs to be configured according to how our services will be scaled. #1 If you noticed a surge in demand for one of your services, you might be tempted to increase the number of replicas, and the deployment using the ReplicaSet will not complain about it. If the cluster has enough resources, the replicas will be created; #2 If the application infrastructure is not correctly configured, you might encounter a lot of issues, such as exhausting the database connection pool or overloading the database pods, as they are not scaled when you scale up your deployments.

- 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
- Supporting 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. Take this into consideration when researching and deciding which operator to use.

Regarding the application infrastructure 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 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 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 that are outside of the realm of Kubernetes.

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

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

To use Crossplane, you must first install its control plane in a Kubernetes cluster. You can follow the 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), you will install and configure one or 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 Crossplane providers (https://docs.crossplane.io/v1.12/concepts/providers/) in charge of understanding and interacting with cloud provider-specific services to provision cloud resources on our behalf. Figure 2.5 shows how by installing the GCP provider and the AWS provider, our Crossplane installation can provision resources on both clouds.

By installing Crossplane providers, you are extending the Kubernetes API's 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 that cover the major cloud providers such as GCP, AWS, and Azure. You can find these Crossplane providers in the Crossplane GitHub's organization: https://docs.crossplane.io/latest/concepts/providers/.

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.

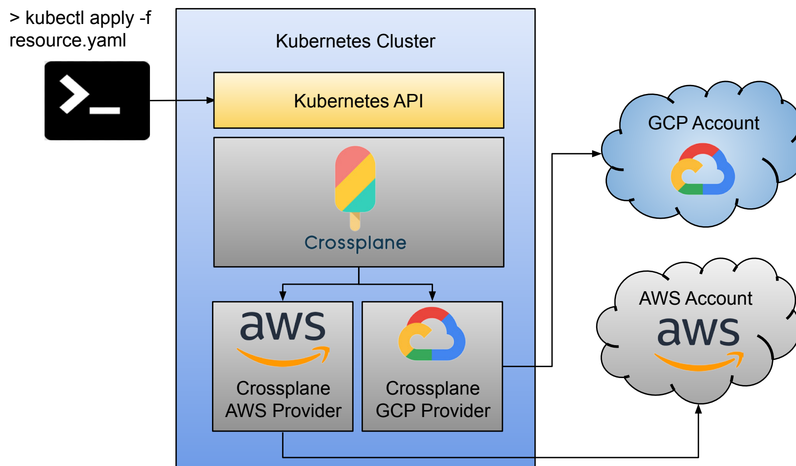


Figure 2.5 Crossplane installed with GCP and AWS providers

For example, creating a bucket in Google Cloud using the Crossplane GCP provider looks like listing 2.4.

Listing 2.4 Google Cloud Platform bucket resource definition

```
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 types 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 need extra network and security configurations, user credentials, and other cloud provider-specific configurations to connect to these provisioned resources. Welcome Crossplane compositions!

2.2.2 Crossplane compositions

Crossplane aims to serve two different personas: *platform teams* and *application teams*. While *platform* teams are cloud provider experts who 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 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 together several components is a very convenient feature, and to achieve these abstractions and simplified interfaces, Crossplane introduced two concepts, *Composite Resource Definitions (XRDs)* and *Composite Resources (XRs)*.

Figure 2.6 shows how you can use Crossplane XRD to define abstractions for different cloud providers. The platform team might be very knowledgeable in Google Cloud or Azure, so they will be in charge of defining which Resources they want to wire up together for a specific application. The application team has a simple resource interface to request the resource they are interested in. But as usual, abstractions are complicated and good to show who is responsible for what, but let's look at a concrete example to understand the power of Crossplane compositions.

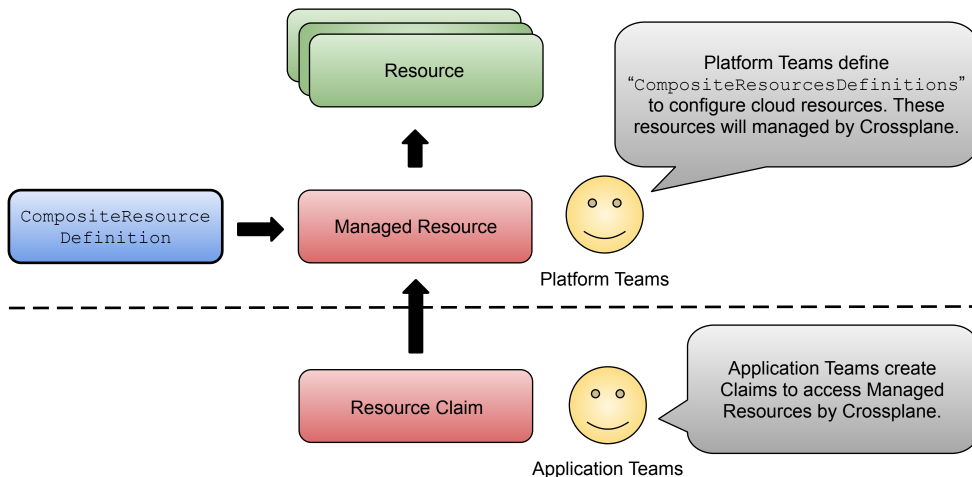


Figure 2.6 Resource composition abstractions by Crossplane composite resources

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

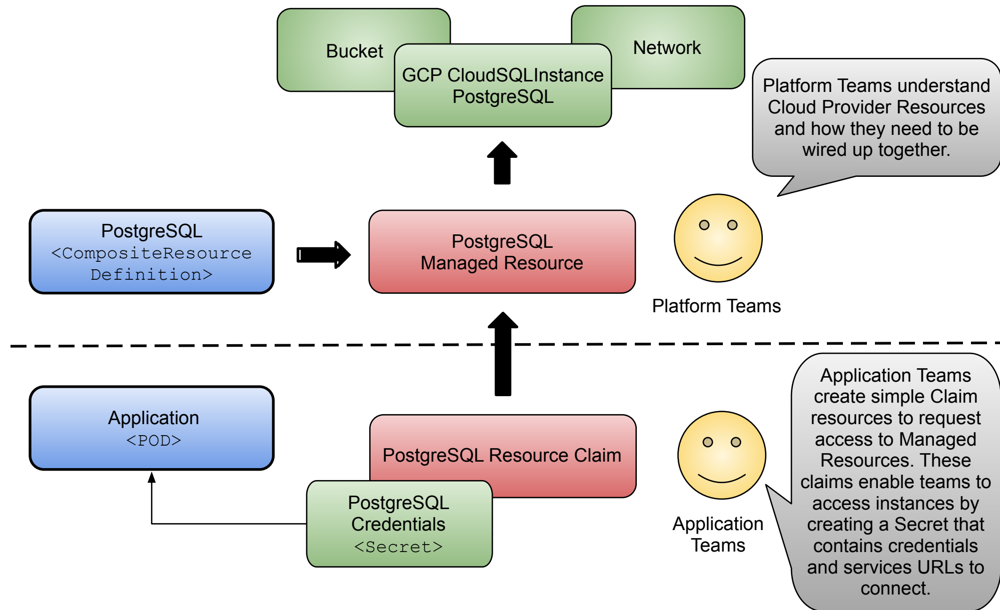


Figure 2.7 Provisioning a PostgreSQL instance in Google Cloud with Crossplane compositions

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 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.

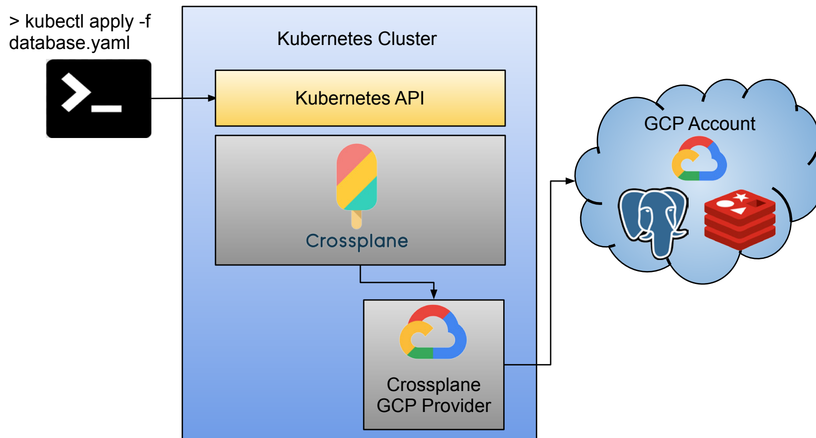


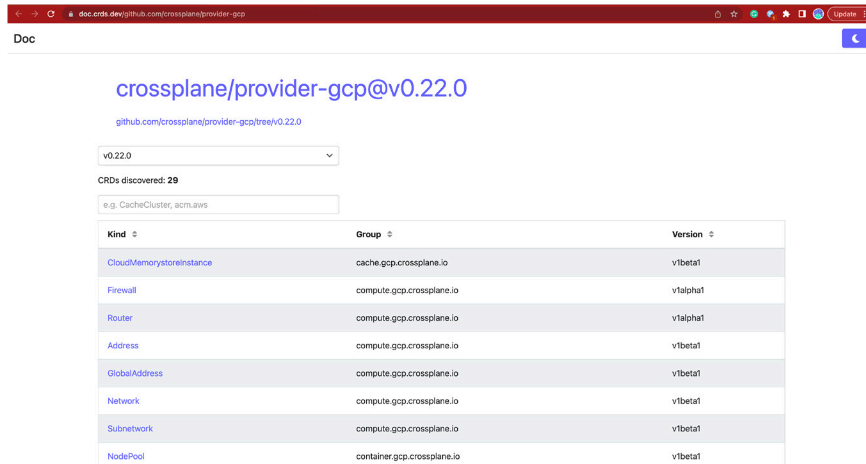
Figure 2.8 Crossplane in Google Cloud Platform

Figure 2.8 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:

- For creating 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. 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).



Kind	Group	Version
CloudMemorystoreInstance	cache.gcp.crossplane.io	v1beta1
Firewall	compute.gcp.crossplane.io	v1alpha1
Router	compute.gcp.crossplane.io	v1alpha1
Address	compute.gcp.crossplane.io	v1beta1
GlobalAddress	compute.gcp.crossplane.io	v1beta1
Network	compute.gcp.crossplane.io	v1beta1
Subnetwork	compute.gcp.crossplane.io	v1beta1
NodePool	container.gcp.crossplane.io	v1beta1

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 access to the list, create, and modify these resources.

In section 2.3, we will look at how to provision cloud or local resources for our applications 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 observe the created managed resources for us. These managed resources offer some advantages compared with just installed resources using Helm. Managed resources have very well-defined behaviors. Here is a summary of what to expect from a Crossplane managed resource:

- *Visible as any other Kubernetes resource:* Crossplane managed resources are just Kubernetes resources. This means that we can use any Kubernetes tool to monitor and query the state of these resources.
- *Continuous reconciliation:* When a managed resource is created, the provider will continuously monitor the resource to ensure 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 Crossplane 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 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.

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 created resource is a Crossplane managed resource, let's imagine a CloudSQLInstance resource the GCP Crossplane provider will pick up and manage.
3. The first step to execute when managing a resource will be checking if it exists in the infrastructure (that is, in the configured GCP account). If it doesn't exist, the provider will request that the resource be created in the infrastructure. The appropriate SQL database will be provisioned depending on the properties set on the resource, such as which kind of SQL database is required. Imagine that we have chosen a PostgreSQL database for the sake of the example.

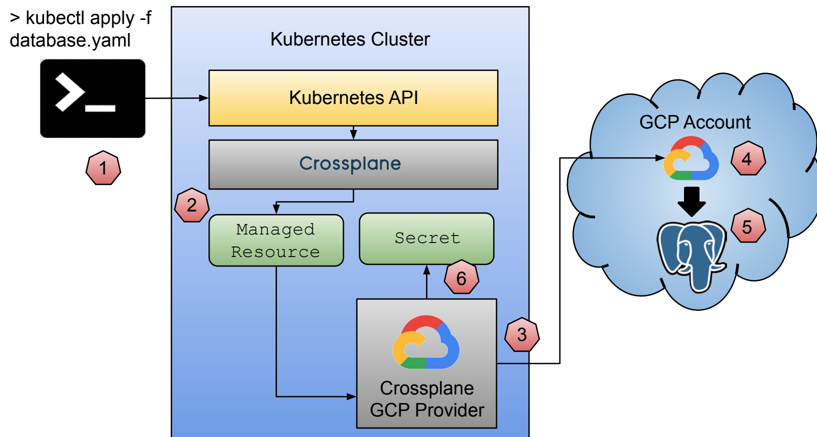


Figure 2.10 Lifecycle of managed resources with Crossplane

4. The cloud provider, after receiving the request, if the resources are enabled, will create a new PostgreSQL instance with the configured parameters in the managed resource.
5. The status of the PostgreSQL will be reported back to the managed resource, which means that we can use `kubectl` or any other tool to monitor the status of the provisioned resources. Crossplane providers will keep these in sync.
6. When the database is up and running, the Crossplane provider will create a secret to store the credentials and properties that our applications will need to connect to the newly created instance.

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 using Crossplane—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 Redis, PostgreSQL, and Kafka instances. From the application perspective, as soon as these three components are available, we can connect to them, and we are good to go. How these components are configured is the responsibility of the operations teams.

The conference application helm chart that we installed in chapter 1 included the installation of Redis, PostgreSQL, and Kafka as Helm dependencies using a conditional value that can be set at installation time. Let's take a quick look at how this was wired up for our Helm Chart: https://github.com/salaboy/platforms-on-k8s/blob/main/conference-application/helm/conference-app/Chart.yaml#L13.

The Conference Helm Chart includes the Redis, PostgreSQL, and Kafka charts dependencies, as shown in listing 2.5.

Listing 2.5 Conference application with Helm Chart dependencies

```
apiVersion: v2
description: A Helm chart for the Conference App
name: conference-app
version: v1.0.0
type: application
icon: https://www.salaboy.com/content/images/2023/06/avatar-new.png
appVersion: v1.0.0
home: http://github.com/salaboy/platforms-on-k8s
dependencies:
- name: redis
  version: 17.11.3
  repository: https://charts.bitnami.com/bitnami
  condition: install.infrastructure
- name: postgresql
  version: 12.5.7
  repository: https://charts.bitnami.com/bitnami
  condition: install.infrastructure
- name: kafka
  version: 22.1.5
  repository: https://charts.bitnami.com/bitnami
  condition: install.infrastructure
```

You can include any number of dependencies to your Helm Charts. This allows complex compositions.

Each dependency requires the chart name, the repository where it is hosted (notice that you can use oci:// references here too), and the version of the chart that you want to install.

Custom conditions can be defined to decide if this dependency is injected when we install the chart.

For this example, all the application infrastructure dependencies are defined at the application level (dependencies section in the Chart.yaml file), but there is nothing stopping you from having one Helm Chart per service, which internally defines its own dependencies.

This kind of chart dependency works for development teams that want to install the entire application with all the components needed with a single command. Still, we want to decouple all the application infrastructural concerns from application services for larger scenarios. Luckily, the Conference application Helm Chart allows us to turn

off these component dependencies, allowing us to plug in Redis, PostgreSQL, and Kafka instances hosted and managed by different teams (figure 2.11).

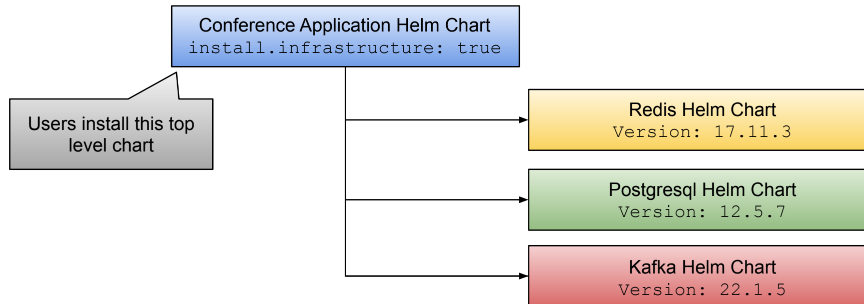


Figure 2.11 Using Helm Chart dependencies for application infrastructure

By separating who requests and who provisions the 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, which 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 request can be provisioned locally (using the Crossplane Helm provider) or remotely using Crossplane cloud providers. Let's see what this would look like. You can follow a step-by-step tutorial to install, configure, and create your Crossplane compositions: https://github.com/salaboy/platforms-on-k8s/tree/main/chapter-5.

In this example, we will create a KinD cluster and configure Crossplane to allow teams to request application infrastructure on demand using the Crossplane Helm provider for development purposes. In production the same requests will be satisfied via scalable cloud resources. More specifically, we enable teams to request Redis, PostgreSQL, and Kafka instances this way using a simplified interface.

For our Conference application example, the platform team decided to create two different concepts:

- *Databases:* NoSQL and SQL databases such as Redis and PostgreSQL.
- *Message brokers:* For managed and unmanaged message brokers such as Kafka.

After having Crossplane and the Crossplane Helm provider installed, the platform team needs to define two Kubernetes resources:

- *Crossplane Composite Resource Definitions (XRDs):* Defines the resources we want to expose to our teams—in this example, Database and MessageBroker. These Composite Resource Definitions define an interface that multiple Compositions can implement.

- *Crossplane composition:* The Crossplane composition allows us to define a set of resource manifests. We can link a composition to a Composite Resource Definition and implement that XRD. By doing so, when the user requests new resources from the XRD–defined resource, all the composed resource manifests in the composition will be created in the cluster. We can provide multiple compositions (for example for different cloud providers), all implementing the same XRD, and then use labels in our resources to choose which composition should kick in.

I know this might sound confusing at first, so let's see these concepts in action. Let's look at the database Crossplane Composite Resource Definition (https://github.com/salaboy/platforms-on-k8s/blob/main/chapter-5/resources/app-database-resource.yaml) in listing 2.6.

Listing 2.6 Database 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
    name: v1alpha1
    served: true
    referenceable: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              parameters:
                type: object
                properties:
                  size:
```

As with every Kubernetes resource, the CompositeResourceDefinition needs a unique name. (→ `name: databases.salaboy.com`)

This CompositeResourceDefinition defines a new type of resource that needs to have a group and a kind. (→ `group: salaboy.com`)

Our new resource type that users can request is Database, because we want to enable them to request new databases. (→ `kind: Database`)

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. (→ `parameters:`)

```
          type: string
        mockData:
          type: boolean
      required:
      - size
  required:
  - parameters
```

Because the Kubernetes API server can validate all resources, we can define which parameters are required and their types and other validations. The Kubernetes API server will reject our resource request if these parameters are not provided or invalid.

We have defined a new type of resource called a Database, which contains two parameters that we can set, `size` and `mockData`. Users can define how many resources are allocated for that instance by setting up the `size` parameter. Instead of worrying about 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 see what the Crossplane composition looks like that will implement this XRD, in listing 2.7.

Listing 2.7 Key/value Database Crossplane composition

```
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:
          rollbackLimit: 3
          forProvider:
            namespace: default
            chart:
              name: redis
```

The composition resource also needs a unique name.

For each composition, we can also define labels. We will then use these to match compositions with the requested Database resources.

Using the compositeTypeRef property, we are linking 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.

We need to provide the values 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.

```
          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
```

Using the providerConfigRef, we can target different Crossplane Helm provider configurations. This means 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, we can patch resources to configure them to work together or to apply the parameters of the requested resource. 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 multiple resources, you, as the person defining the composition, will need to define what this condition is.

With this composition, we link our `Database` claim 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 shows two user requests for the same database type.

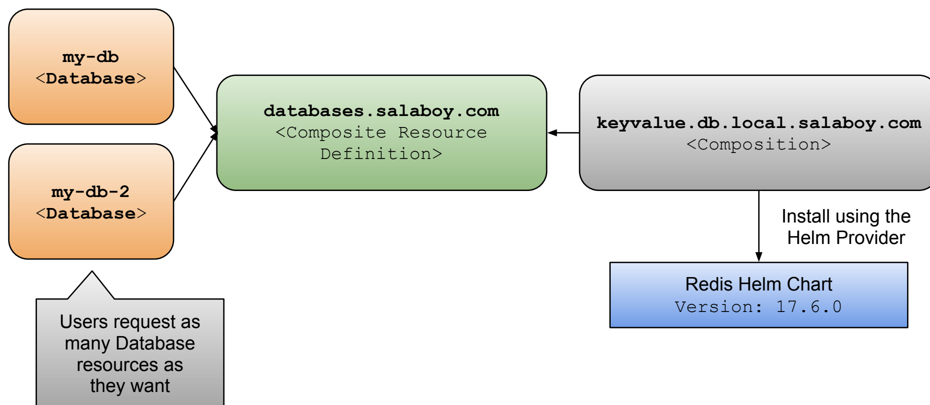


Figure 2.12 Crossplane composition and Composite Resource Definition working together

It is important to note 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 to a completely different cluster.

In the step-by-step tutorial (https://github.com/salaboy/platforms-on-k8s/tree/main/chapter-5), you will install the three Composite Resource Definitions and three compositions. Once these are installed, as shown in figure 3.12, you can request new databases and message brokers, and for every request, all the resources defined in the composition will be provisioned. For the sake of simplicity, the key-value database composition just installs Redis, but there are no limits on how many resources you can create (except for the available hardware or quotas you have).

A `Database` resource is just another Kubernetes resource that now our cluster understands, and it looks like listing 2.8.

Listing 2.8 Teams create database resources to request new database instances

```
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
```

- **The unique name for the resource** (metadata name)
- **We use matchLabels to select the appropriate composition.** (matchLabels)
- **We need to set the parameters that are required by our Database resource claim.** (parameters)

The schema for this Database resource is defined inside the Crossplane `CompositeResourceDefinition`. 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 yet 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, as shown in listing 2.9. The PostgreSQL composition will look similar to the one for Redis, but it will install the PostgreSQL helm chart instead.

Listing 2.9 SQL Database Crossplane Composition

```
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
    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
          …
```

- `name: sql.db.local.salaboy.com` — We need a unique name for our composition, so we can differentiate it from the keyvalue composition that we used for Redis.
- `kind: sql` — We use a different label to describe this composition, notice that the provider is the same as before.
- `chart:` — 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, as shown in listing 2.10.

Listing 2.10 Database resource with kind: `sql` label to select implementation

```
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
```

- `name: my-db-sql` — The unique name used for the PostgreSQL database.
- `kind: sql` — 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 select the right composition based on the `kind` label value.

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.

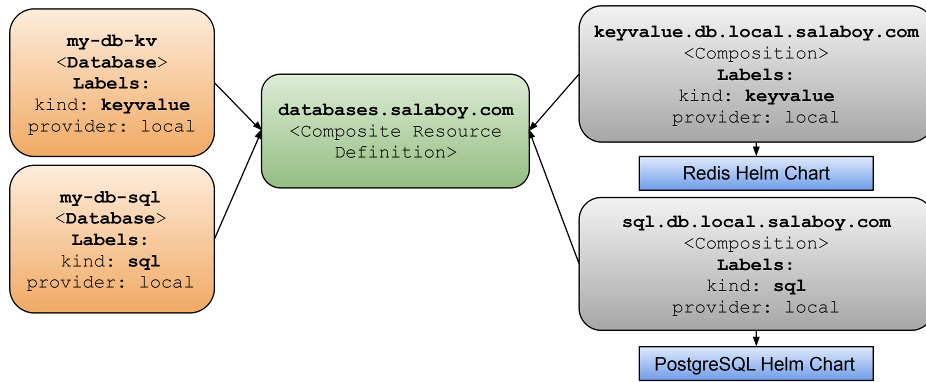


Figure 2.13 Selecting compositions using labels

If we use the Google Cloud Platform (GCP) as an example, for compositions that use cloud resources from GCP, you will need to install the Crossplane GCP provider and configure it accordingly, as explained in the official Crossplane documentation: https://docs.crossplane.io/latest/getting-started/provider-gcp/.

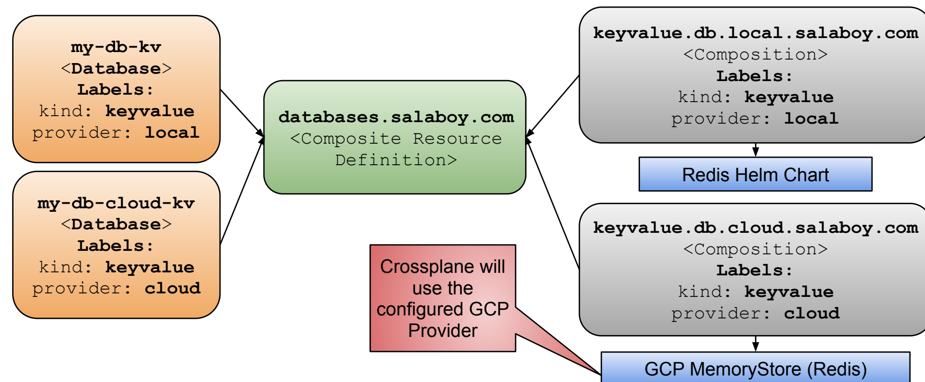


Figure 2.14 Selecting compositions using different providers, still using labels

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

NOTE Check the community–contributed AWS compositions for this example using the Crossplane AWS Provider at https://github.com/salaboy/platforms-on-k8s/tree/main/chapter-5/aws.

Then, creating new database resources that will be provisioned in the Google Cloud Platform will look like listing 2.11.

Listing 2.11 Requesting a new SQL database

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

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

The provider label selects the composition labeled with provider: gcp. In other words, with this label, we select where the database will be provisioned.

The kind label allows us to select which kind of database we want to provision.

No matter where our databases or other application infrastructure components are provisioned, we can connect our application's services by following some conventions. We can use the resource name (for example, `my-db-cloud-sql`) to know which Kubernetes service will be used for service discovery. We can also use the created secret to obtain the credentials that we will need to connect.

The step-by-step tutorial also provides a `CompositeResourceDefinition` for message brokers and a composition that installs the Kafka Helm chart that you can find at https://github.com/salaboy/platforms-on-k8s/blob/main/chapter-5/resources/app-messagebroker-kafka.yaml.

One really important thing to consider for this example is that Google Cloud Platform doesn't provide a managed Kafka service. This pushes your team to decide to replace Kafka when the application is going to be deployed on Google Cloud Platform, install and manage Kafka on Google Cloud compute or hire a third-party service. In the AWS example, we have a Kafka–managed service that we can use, so there is no need to change our application code. But still, wouldn't it be nice to abstract away how we connect to these infrastructure services?

Figure 2.15 shows how easy it is to provide a Composite Resource Definition for key/value databases that can be provisioned locally using Helm or managed by a cloud provider. But in the case of Kafka, it gets a bit trickier because you might need to integrate with a third-party service or take the lead in having a team to manage the Kafka instance(s).

Besides Kafka and Google Cloud Platform, your teams will need a strategy to deal with infrastructure across cloud providers, or at least make conscious choices about how to deal with situations like this. From the application's services perspective, would you maintain two copies of the same service if you decide to swap Kafka and use Google PubSub instead? One includes the Kafka dependencies, and the other includes the

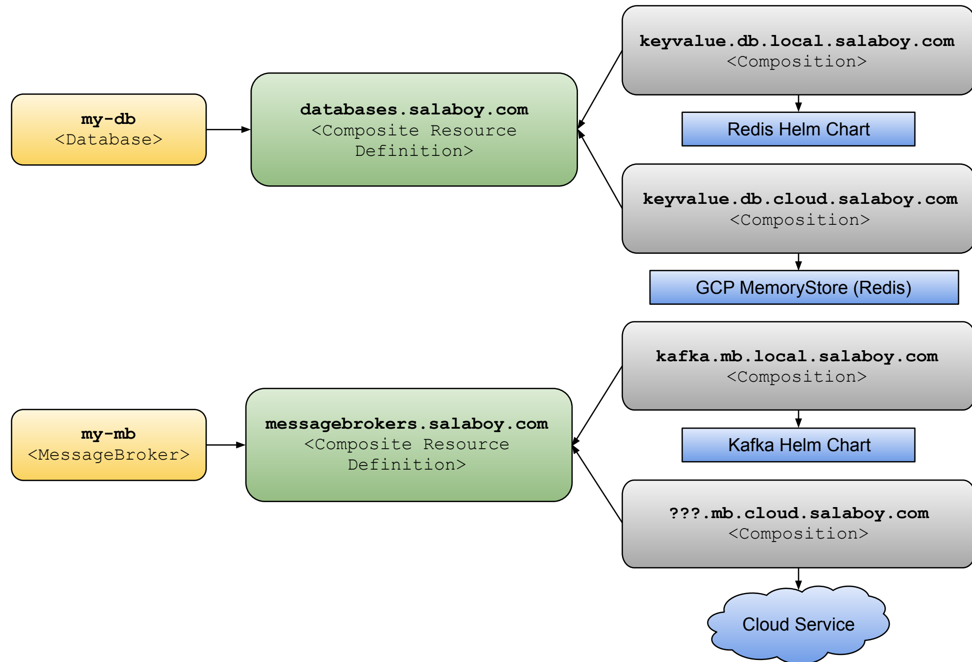


Figure 2.15 Compositions push teams to define which Cloud services are available for our applications to use.

Google GCP SDKs to connect to Google PubSub. If you only use Google PubSub, you lose the ability to run the application outside Google Cloud.

2.3.1 Connecting our services with the new provisioned infrastructure

When we create new database or message broker 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 ensuring that the desired configurations are applied. This means that Crossplane will make sure that our databases and message brokers 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, Redis, and Kafka. As we have seen in chapter 1, this can be disabled by setting one flag: `install.infrastructure=false`:

```
> helm install conference oci://docker.io/salaboy/conference-app
➥--version v1.0.0 --set install.infrastructure=false
```

I strongly recommend you check out the step-by-step tutorial that you can find at https://github.com/salaboy/platforms-on-k8s/tree/main/chapter-5 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, PostgreSQL, or Kafka) will be provisioned by Helm. Still, the application's services will not know where to connect to the Redis, PostgreSQL, and Kafka instances we created using our Crossplane compositions. We need to add more parameters to the application chart, so the services know where to connect. First, check which databases you have available in your cluster as in listing 2.12.

Listing 2.12 Listing all database resources

```
> kubectl get dbs
NAME                SIZE     KIND        SYNCED    READY    COMPOSITION
my-db-keyavalue     small    keyvalue    True      True     keyvalue.db.local.salaboy.com
my-db-sql           small    sql         True      True     sql.db.local.salaboy.com
```

The tutorial also guides you to create a MessageBroker and checks that you have one instance of that too, as in listing 2.13.

Listing 2.13 Listing all MessageBroker resources

```
> kubectl get mbs
NAME          SIZE    KIND    SYNCED    READY    COMPOSITION
my-mb-kafka   small   kafka   True      True     kafka.mb.local.salaboy.com
```

Listing 2.14 shows the Kubernetes pods for our database instances and our message broker.

Listing 2.14 Pods for our application infrastructure

```
> kubectl get pods
NAME                                 READY    STATUS     RESTARTS    AGE
my-db-keyavalue-redis-master-0       1/1      Running    0           25m
my-db-sql-postgresql-0               1/1      Running    0           25m
my-mb-kafka-0                        1/1      Running    0           25m
```

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 (see listing 2.15).

Listing 2.15 Kubernetes secrets containing credentials to connect to our Databases

```
> kubectl get secret
NAME                                      TYPE                  DATA    AGE
my-db-keyavalue-redis                     Opaque                1       26m
my-db-sql-postgresql                      Opaque                1       25m
sh.helm.release.v1.my-db-keyavalue.v1     helm.sh/release.v1    1       26m
sh.helm.release.v1.my-db-sql.v1           helm.sh/release.v1    1       25m
sh.helm.release.v1.my-mb-kafka.v1         helm.sh/release.v1    1       25m
```

Take a look at the services available in the default namespace after we provisioned our databases, see listing 2.16:

Listing 2.16 Custom values.yaml file to connect with new infrastructure

```
> kubectl get services
NAME                             TYPE        CLUSTER-IP      PORT(S)
kubernetes                       ClusterIP   10.96.0.1       443/TCP
my-db-keyavalue-redis-headless   ClusterIP   None            6379/TCP
my-db-keyavalue-redis-master     ClusterIP   10.96.49.121    6379/TCP
my-db-sql-postgresql             ClusterIP   10.96.129.115   5432/TCP
my-db-sql-postgresql-hl          ClusterIP   None            5432/TCP
my-mb-kafka                      ClusterIP   10.96.239.45    9092/TCP
my-mb-kafka-headless             ClusterIP   None            9092/TCP
```

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

```
> helm install conference oci://docker.io/salaboy/conference-app
➥--version v1.0.0 -f app-values.yaml
```

Instead of setting all the parameters in the command, we are using a file for the values to be applied to the chart. For this example, the app-values.yaml file looks like listing 2.17.

Listing 2.17 Helm Chart customized values.yaml file

```
install:
  infrastructure: false
frontend:
  kafka:
    url: my-mb-kafka.default.svc.cluster.local
agenda:
  kafka:
    url: my-mb-kafka.default.svc.cluster.local
  redis:
    host: my-db-keyavalue-redis-master.default.svc.cluster.local
    secretName: my-db-keyavalue-redis
c4p:
  kafka:
    url: my-mb-kafka.default.svc.cluster.local
  postgresql:
    host: my-db-sql-postgresql.default.svc.cluster.local
    secretName: my-db-sql-postgresql
notifications:
  kafka:
    url: my-mb-kafka.default.svc.cluster.local
```

We disable the Redis, PostgreSQL, and Kafka Helm dependencies. These components will not be installed when we install the application.

We use the Kubernetes service created for our Kafka cluster to connect all the application services to the created instance. The same approach is used for Redis and PostgreSQL.

For both Redis and PostgreSQL, a Kubernetes secret is created by the composite resource. Our Helm Chart understands how to get the credentials from the secret, so we only need to specify the secret name.

In this app-values.yaml file, we are not only turning off the Helm dependencies for PostgreSQL, Redis, and Kafka, but we are also configuring the variables needed for the services to connect to our newly provisioned databases. Notice that if the databases were created in a different namespace or with a different name, the `kafka.url`, `postgresql.host` and `redis.host` should contain the appropriate namespace in the fully qualified name of the service, for example, `my-db-sql-postgresql.default.svc.cluster.local` (where `default` is the namespace).

Figure 2.16 shows the Conference application services connecting to the application infrastructure that was created with Crossplane. The boundaries between the developer realm and the platform team become more defined now, as developers interested in getting the infrastructure that they need have a set of options that are carefully selected by the platform team and exposed to developers using simpler interfaces.

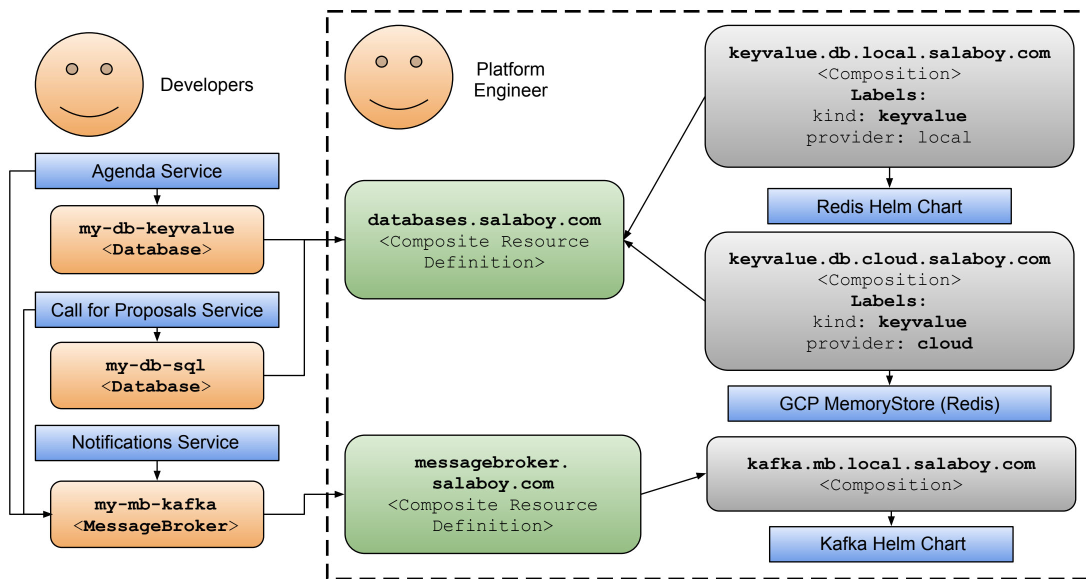


Figure 2.16 Enabling different teams to work together and focus on their tasks at hand

All this effort enables us to split the responsibility of defining, configuring, and running all the application infrastructure to another team not responsible for working on the application's services. Services can be released independently without worrying about which databases are being used or when they need to be upgraded. Developers shouldn't be worrying about cloud provider accounts or if they have access to create different resources. Hence, another team with a completely different set of skills 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 setups for experimentation and testing or to set up new instances of the application quickly. This is a major shift in how we (as developers) were used to doing things, because before this, cloud providers and most companies must have access to a database and a ticketing system to request another team to provision that resource for you, which 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 message brokers such as Kafka and all the configurations needed to access these new instances.
- We exposed a simplified interface for the application teams that is cloud-provider independent because it relies on the Kubernetes API.
- Finally, we connected our application service to the newly provisioned instances by relying on Kubernetes Secrets created by Crossplane, containing all the details required to connect to the newly created instances.

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

NOTE You can follow a step-by-step tutorial that covers all the steps described in this section at https://github.com/salaboy/platforms-on-k8s/tree/main/chapter-5.

2.4 Linking back to platform engineering

We need to be cautious. We cannot expect every developer to understand or be willing to use tools like the ones we have discussed (Crossplane, ArgoCD, Tekton, etc.). We need a way to reduce the complexity these tools introduce. Platforms are meant to reduce the cognitive load of their. For GCP and other platforms, the users interacting with the platform don't need to understand what is going on under the covers, what tools are being used, or the design of the entire platform to use it.

Crossplane was created to serve both platform teams and development teams (or consumers), which have different priorities, interests, and skills. By creating the right abstractions (XRDs) the platform team can expose simple resources that development teams can configure according to their needs, while behind the covers, a complex composition is being set up to create and wire together a group of cloud resources. We have also seen how by using labels and selectors, we can choose between different compositions, enabling the creation of infrastructure in different cloud providers but keeping the same user experience for the teams creating the requests. Crossplane, by extending the Kubernetes APIs, unifies how we manage our workloads and how we can manage application infrastructure across cloud providers. In other words, if we install

Crossplane into a Kubernetes cluster, we cannot only deploy and run our clusters but also provision and manage cloud resources by using the same tooling that we use for our workloads.

With all the goodies that Crossplane brings, you must also be ready for some drawbacks and challenges. Platform teams looking into Crossplane have other more popular options available to provision cloud resources, such as Hashicorp's Terraform and Pulumi. Crossplane is much more recent than Terraform, and because Crossplane is focused on Kubernetes, it requires platform teams to be fully invested in Kubernetes. Teams not used to managing Kubernetes clusters will find tools like Crossplane challenging at first, so you need to level up your Kubernetes skills to run and maintain a tool like Crossplane.

Platform teams will be forced to make a decision about using Crossplane or tools like Terraform, and my recommendation is to think about how much you want to align the tools that you are using with the Kubernetes APIs. Being able to manage infrastructure (cloud resources) in the same way that we manage our applications makes a lot of sense in theory. Still, it also needs to make sense to the teams managing and maintaining these components up and running. In the last couple of years, there has been a huge increase in maturity around observability, security, and operations in the cloud-native space. More and more teams are feeling comfortable with managing and operating Kubernetes at scale. For those teams, Crossplane can be a great addition, because it is going to work with all their existing Kubernetes observability stacks, policy enforcers, and dashboards.

When having tools as flexible as Crossplane, you open the door to new possibilities that can span across cloud providers. Platform teams now have more options available, and that can be counterproductive, but one thing is clear. If you use the right abstractions, the platform can be flexible, as consumer interfaces will not change. At the same time, the platform team can iterate on their previous decisions and provide new implementations behind the covers.

Figure 2.17 shows how by using Crossplane, we can provide self-service abstractions for development teams to consume. They can request databases, message brokers, identity services, and any other internal or external services they might need for their applications. But what do they need from an application perspective? Think about the Kafka example provided before. What needs to change in your applications if you move from Kafka to Google PubSub?

We have covered a lot of ground so far, from installing a simple application into a cluster to building services and deploying them using a GitOps approach and now provisioning application infrastructure declaratively. Figure 2.18 shows how using a GitOps approach we can define not only which services/applications should be running inside an environment, but also which cloud resources need to be provisioned and wired to our application services.

It is time to put everything together into a platform, because it doesn't make too much sense to have our applications running in the same cluster where our pipelines

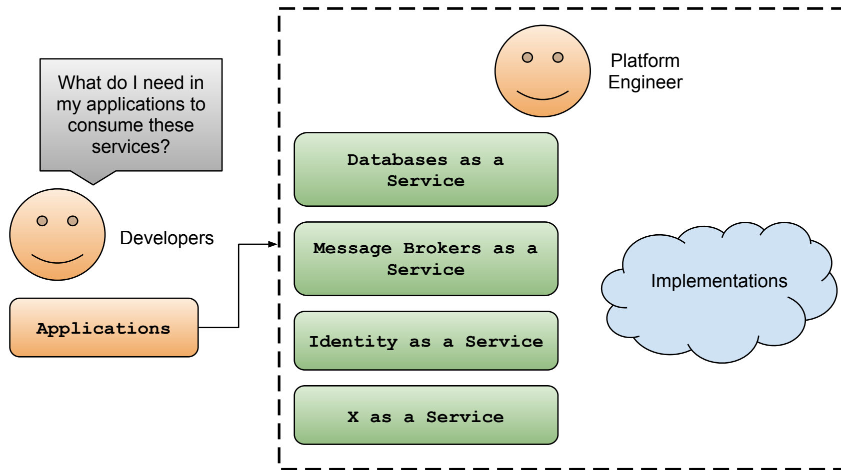


Figure 2.17 What do developers need to consume all these Platform services?

and other tools are running. What would a platform on top of Kubernetes look like? What are the main challenges that your teams will face when trying to build one? There is only one way to find out: Let's build a platform on top of Kubernetes!

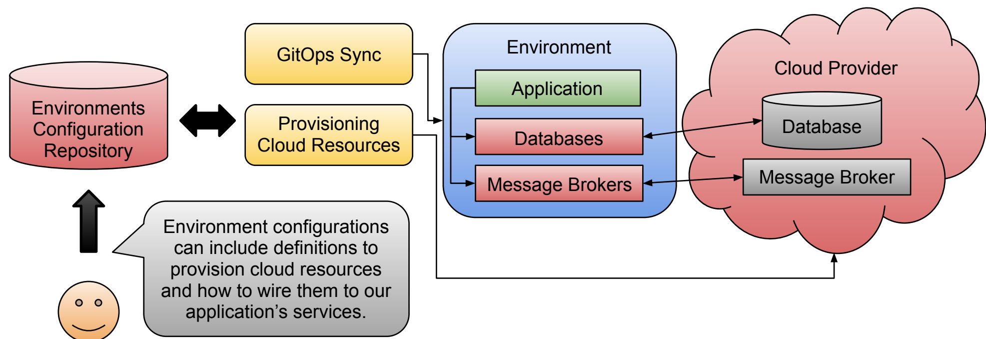


Figure 2.18 Provisioning application infrastructure using a declarative GitOps approach

Summary

- Cloud-native applications depend on application infrastructure to run, as each service might require different persistent storages, a message broker to send messages, and other components to work.
- Creating application infrastructure inside cloud providers is easy and can save us a lot of time, but then we rely on their tools and ecosystem.

- Provisioning infrastructure in a cloud-agnostic way can be achieved by relying on the Kubernetes API and tools like Crossplane, which abstracts the underlying cloud provider and lets us define which resources must 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 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.
- If you followed the step-by-step tutorial, you got hands-on experience on how to provision application infrastructure using a multi-cloud approach using Crossplane.

3 Platform capabilities: Shared application concerns

This chapter covers

- Learning requirements of 95% of cloud-native applications
- Reducing friction between application and infrastructure
- Addressing shared concerns with standard APIs and components

In chapter 2 we created abstractions such as databases and message brokers to provision and configure all the components required by our application's services. This platform enables teams to request new development environments that not only create isolated environments but also install an instance of the Conference application (and all the components required by the application) so teams can do their work. By going through the process of building a platform, we defined the responsibilities of a platform team and where each tool belongs and why.

So far, we have given developers Kubernetes clusters with an instance of their application running. This chapter looks at mechanisms to provide developers with capabilities closer to their application needs. Most of these capabilities will be

accessed by APIs that abstract away the application's infrastructure needs, allowing the platform team to evolve (update, reconfigure, change) infrastructural components without updating any application code. At the same time, developers will interact with these platform capabilities without knowing how they are implemented and without bloating their applications with a load of dependencies. This chapter is divided into three sections:

- What are most applications doing 95% of the time?
- Standard APIs and abstractions to separate application code from infrastructure.
- Updating our Conference application with Dapr (Distributed Application Runtime), a CNCF and open-source project created to provide solutions to distributed application challenges.

Let's start by analyzing what most applications are doing. Don't worry; we will also cover edge cases.

3.1 What are most applications doing 95% of the time?

We have learned how to run it on top of Kubernetes and how to connect the services to databases, key-value stores, and message brokers. There was a good reason to go over those steps and include those behaviors in the walking skeleton. Most applications, like the Conference application, will need the following functionality:

- *Call other services to send or receive information:* Application services don't exist on their own. They need to call and be called by other services. Services can be local or remote, and you can use different protocols, most commonly HTTP and GRPC. We use HTTP calls between services for the conference application walking skeleton.
- *Store and read data from persistent storage:* This can be a database, a key-value store, a blob store like S3 buckets, or even writing and reading from files. For the conference application, we are using Redis and PostgreSQL.
- *Emit and consume events or messages asynchronously:* Using asynchronous messaging for communicating systems implementing an event-driven architecture is a common practice in distributed systems. Using tools like Kafka, RabbitMQ, or even cloud-provider messaging systems is common. Each service in the Conference application is emitting or consuming events using Kafka.
- *Accessing credentials to connect to services:* When connecting to an application's infrastructure components, whether local or remote, most services will need credentials to authenticate to other systems. In this book I've only mentioned tools like external-secrets (https://github.com/external-secrets/external-secrets) or HashiCorp's Vault (https://www.vaultproject.io/), but we haven't dug deeper into it.

Whether we are building business applications or machine learning tools, most applications will benefit from having these capabilities easily available to consume. And

while complex applications require much more than that, there is always a way to separate the complex part from the generic parts.

Figure 3.1 shows several example service interactions with each other and available infrastructure. Service A is calling Service B using HTTP (for this topic, GRPC would fit similarly). Service B stores and reads data from a database and will need the right credentials to connect. Service A also connects to a message broker and places messages into it. Service C can pick messages from the message broker and, using some credentials, connect to a Bucket to store some calculations based on the messages it receives.

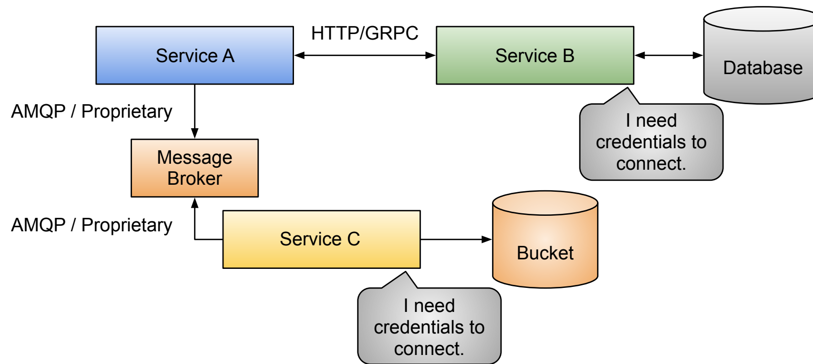


Figure 3.1 Common communication patterns in distributed applications

No matter what logic these services are implementing, we can extract some constant behaviors and enable development teams to consume without the hassle of dealing with the low-level details or pushing them to make decisions around cross-cutting concerns that can be solved at the platform level.

To understand how this can work, we must look closely at what is happening inside these services. As you might know already, the devil is in the details. While from a high-level perspective, we are used to dealing with services doing what is described in figure 3.1, if we want to unlock an increased velocity in our software delivery pipelines, we need to go one level down to understand the intricate relationships between the components of our applications. Let's take a quick look at the challenges the application teams face when trying to change different services and infrastructure that our services require.

3.1.1 The challenges of coupling application and infrastructure

Fortunately, this is not a programming language competition, independent of your programming language of choice. If you want to connect to a database or message broker, you must add some dependencies to your application code. While this is a common practice in the software development industry, it is also one of the reasons why delivery speed is slower than it should be.

Coordination between different teams is the reason behind most blockers when releasing software. We have created architectures and adopted Kubernetes because we want to go faster. By using containers, we have adopted an easier and more standard way to run our applications. No matter in which language the application is written or which tech stack is used, if you give me a container with the application inside, I can run it. We have removed the application dependencies on the operating system and the software that we need to have installed in a machine (or virtual machine) to run your application, which is now encapsulated inside a container.

Unfortunately, we haven't tackled the relationships and integration points between containers (our application's services). We also haven't solved how these containers will interact with application infrastructure components that can be local (self-hosted) or managed by a cloud provider.

Let's take a closer look at where these applications heavily rely on other services and can block teams from making changes, pushing them for complicated coordination that can end up causing downtime to our users. We will start by splitting up the previous example into the specifics of each interaction.

3.1.2 Service-to-service interaction challenges

To send data from one service to another, you must know where the other service is running and which protocol it uses to receive information. Because we are dealing with distributed systems, we also need to ensure that the requests between services arrive at the other service and have mechanisms to deal with unexpected network problems or situations where the other services might fail. In other words, we need to build resilience in our services. We cannot always trust the network or other services to behave as expected.

Let's use Service A and Service B as examples to go deeper into the details. In figure 3.2, Service A needs to send a request to Service B.

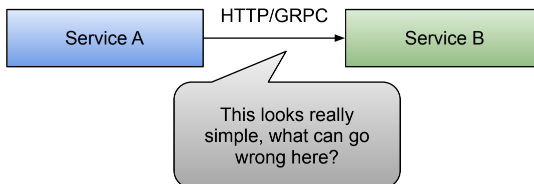


Figure 3.2 Service-to-service interaction challenges

But let's dig deeper into the mechanisms services can use internally. Suppose we leave the fact that Service A depends on the Service B contract (API) to be stable and not change for this to work on the side. What else can go wrong here? As mentioned, development teams should add a resiliency layer inside their services to ensure that Service A requests reach Service B. One way to do this is to use a framework to retry the request if it fails automatically. Frameworks implementing this functionality are available for

all programming languages. Tools like `go-retryablehttp` (https://github.com/hashicorp/go-retryablehttp) or Spring Retry for Spring Boot (https://github.com/spring-projects/spring-retry) add resiliency to your service-to-service interactions. Some of these mechanisms also include exponential backoff functionality to avoid overloading services and the network when things are going wrong.

Unfortunately, there is no standard library shared across tech stacks that can provide the same behavior and functionality for all your applications, so even if you configure both Spring Retry and `go-retryablehttp` with similar parameters, it is quite hard to guarantee that they will behave in the same way when services start failing.

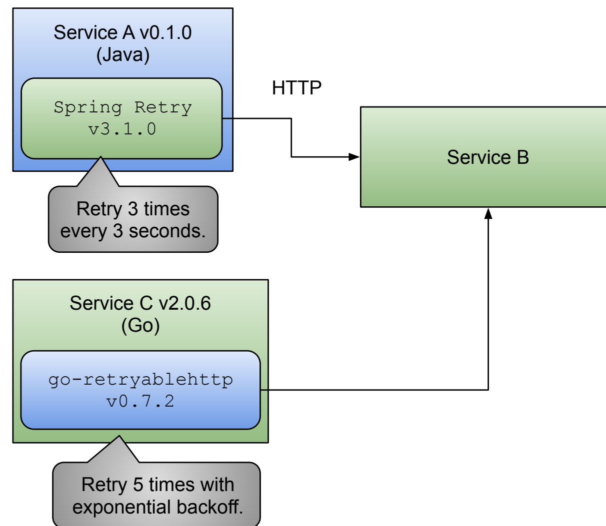


Figure 3.3 Service-to-service interactions retry mechanisms

Figure 3.3 shows Service A written in Java using the Spring Retry library to retry three times with a wait time of 3 seconds between each request when the request fails to be acknowledged by Service B. Service C, written in Go using the `go-retryablehttp` library, is configured to retry five times but using an exponential backoff (the retry period between requests is not fixed; this can provide time for the other service to recover and not be flooded with retries) mechanism when things go wrong.

Even if the applications are written in the same language and using the same frameworks, both services (A and B) must have compatible versions of their dependencies and configurations. If we push both Service A and Service B to have the versions of the frameworks, we are coupling them together, meaning we will need to coordinate the update of the other service whenever any of these internal dependency versions change. This can cause even more slowdowns and increase the complexity of coordination efforts.

NOTE In this section, I've used retrying mechanisms as an example, but think about other cross-cutting concerns that you might want to include for these service-to-service interactions, like circuit breakers (also for resiliency) rate limiting and observability. Consider the frameworks and libraries you will need to add to instrument your application code to get metrics from it.

On the other hand, using different frameworks (and versions) for each service will complicate troubleshooting these services for our operations teams. Wouldn't it be great to have a way to add resiliency to our applications without modifying them? Before answering this question, what else can go wrong?

Something that developers often overlook relates to the security aspect of these communications. Service A and Service B don't live in a vacuum, meaning other services surround them. If any of these services is compromised by a bad actor, having a free-for-all service-to-service invocation between all the services makes our entire system insecure. This is where having service identity and the right security mechanisms to ensure that, for this example, Service A can only call Service B is extremely important, as shown in figure 3.4.

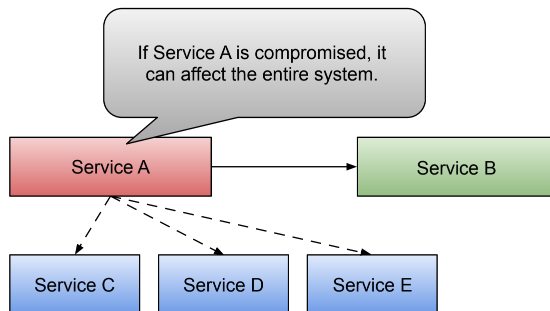


Figure 3.4 If a service is compromised, it can affect the entire system.

Having a mechanism that allows us to define our service's identity, we can define which service-to-service invocations are allowed and which protocols and ports are allowed for the communications to happen. Figure 3.5 shows how we can reduce the blast radius (how many services are affected if a security breach happens) by defining rules that enforce which services are allowed in our system and how they are supposed to interact.

Having the right mechanisms to define and validate these rules cannot be easily built inside each service. Hence developers tend to assume that an external mechanism will be in charge of performing these checks.

As we will see in the following sections, service identity is something that we need across the board and not only for service-to-service interactions. Wouldn't it be great to have a simple way to add service identity to our system without changing our application's services?

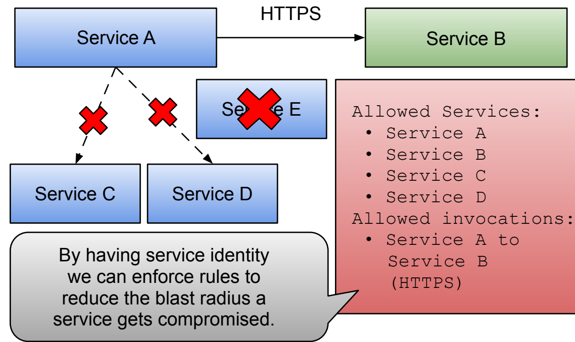


Figure 3.5 Reducing the blast radius by defining system-level rules

Before answering this question, let's look at other challenges teams face when architecting distributed applications. Let's talk about storing and reading state, which most applications do.

3.1.3 Storing/reading state challenges

Our application needs to store or read state from persistent storage. That is quite a common requirement, right? You need data to do some calculations, then store the results somewhere so they don't get lost if your application goes down. In our example, figure 3.6, Service B needed to connect to a database or persistent storage to read and write data.

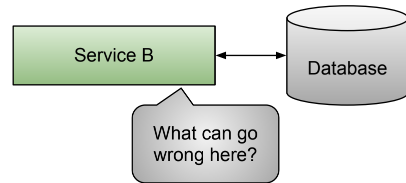


Figure 3.6 Storing/reading state challenges

What can go wrong here? Developers are used to connecting to different kinds of databases (relational, NoSQL, files, buckets) and interacting with them. But two main friction points slow teams from moving their services forward: dependencies and credentials.

Let's start by looking at dependencies. What kind of dependencies does Service B need to connect to a database? Figure 3.7 shows Service B connecting to both a relational database and a NoSQL database. To achieve these connections, Service B needs to include a driver and a client library, plus the configuration needed to fine-tune how the application will connect to these two databases. These configurations define the size of the connection pool (how many application threads can connect concurrently to the database), buffers, health checks, and other important details that can change how the application behaves.

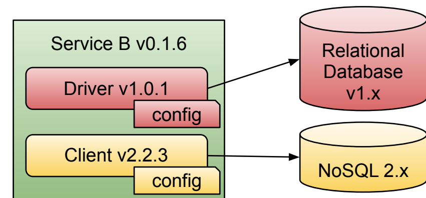


Figure 3.7 Databases dependencies and client versions

Besides the configuration of the driver and the client, their versions need to be compatible with the version of the databases we are running, and this is where the challenges begin.

> **NOTE** It is important to notice that each driver/client is specific to the database (relational or NoSQL) that you are connecting to. This section assumes you used a specific database because it meets your application's requirements. Each database vendor has unique features optimized for different use cases. In this chapter, we are more interested in 95% of the cases that do not use vendor-specific features.

Once the application's service is connected to the database using the client APIs, it should be fairly easy to interact with it. Whether by sending SQL queries or commands to fetch data or using a key-value API to read keys and values from the database instance, developers should know the basics to start reading and writing data.

Do you have more than one service interacting with the same database instance? Are they both using the same library and the same version? Are these services written using the same programming language and frameworks? Even if you manage to control all these dependencies, there is still a coupling that will slow you down. Whenever the operations teams decide to upgrade the database version, each service connecting to this instance might or might not need to upgrade its dependencies and configuration parameters. Would you upgrade the database first or the dependencies?

For credentials, we face a similar problem. It is quite common to consume credentials from a credential store like HashiCorp's Vault (https://www.vaultproject.io/). If not provided by the platform and not managed in Kubernetes, application services can include a dependency to consume credentials from their application's code easily. Figure 3.8 shows Service B connecting to a credential store, using a specific client library, to get a token to connect to a database.

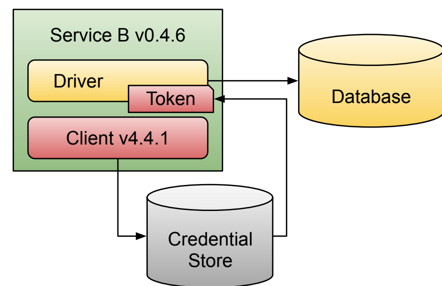


Figure 3.8 Credentials store dependencies

In chapters 1 and 2, we connected the Conference services to different components using Kubernetes Secrets. By using Kubernetes Secrets, we were removing the need for application developers to worry about where to get these credentials from.

Otherwise, if your service connects to other services or components that might require dependencies in this way, the service will need to be upgraded for any change in any of the components. This coupling between the service code and dependencies creates the need for complex coordination between application development teams, the platform team, and the operations teams in charge of keeping these components up and running.

Can we get rid of some of these dependencies? Can we push some of these concerns down to the platform team, so we remove the hassle of keeping them updated from developers? If we decouple these services with a clean interface, then the infrastructure and applications can be updated independently.

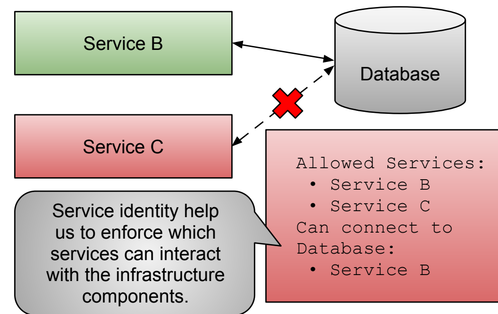


Figure 3.9 Enforcing rules based on service identity

Before jumping into the next topic, I wanted to briefly talk about why having service identity at this level can also help reduce security problems when interacting with application infrastructure components. Figure 3.9 shows how similar service identity rules can be applied to validate who can interact with infrastructure components. Once again, the system will limit the blast radius if a service is compromised.

But what about asynchronous interactions? Let's look at how these challenges relate to asynchronous messaging before jumping into the solutions space.

3.1.4 Asynchronous messaging challenges

With asynchronous messaging, you want to decouple the producer from the consumer. When using HTTP or GRPC, Service A needs to know about Service B, and both services need to be up to exchange information. When using asynchronous messaging, Service A doesn't know anything about Service C. You can take it even further, where Service C might not even be running when Service A places a message into the message broker. Figure 3.10 shows Service A placing a message into the message broker; at a later point in time, Service C can connect to the message broker and fetch messages from it.

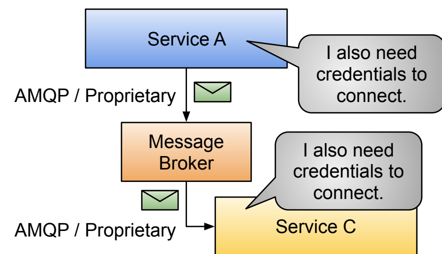


Figure 3.10 Asynchronous messaging interactions

Similar to HTTP/GRPC service-to-service interactions, when using a message broker, we need to know where the message broker is to send messages to or to subscribe to get messages from. Message brokers also provide isolation to enable applications to group messages together using the concept of topics. Services can be connected to the same message broker instance but send and consume messages from different topics.

When using message brokers, we face the same problems described with databases. We need to add a dependency to our applications depending on which message broker we decide to use, its version, and the programming language that we have chosen.

Message brokers will use different protocols to receive and send information. A standard increasingly adopted in this space is the CloudEvent specification (https://cloudevents.io/) from the CNCF. While CloudEvents is a great step forward, it doesn't save your application developers from adding dependencies to connect and interact with your message brokers.

Figure 3.11 shows Service A, which includes the Kafka client library to connect to Kafka and send messages. Besides the URL, port, and credentials to connect to the Kafka instance, the Kafka client also receives configurations on how the client will behave when connecting to the broker, similar to databases. Service C uses the same client, but with different versions, to connect to the same broker.

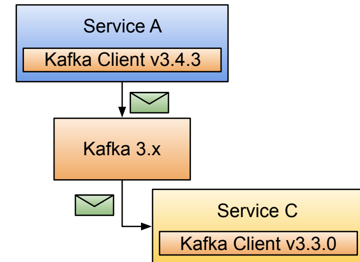


Figure 3.11 Dependencies and API challenges

Message brokers face the same problem as with databases and persistent storage. But unfortunately, with message brokers, developers will need to learn specific APIs that might not be that easy initially. Sending and consuming messages using different programming languages present more challenges and cognitive load on teams without experience with the specifics of the message broker at hand.

Same as with databases, if you have chosen Kafka, for example, it means that Kafka fits your application requirements. You might want to use advanced Kafka features that other message brokers don't provide. However, let me repeat it here: we are interested in 95% of the cases where application services want to exchange messages to externalize the state and let other interested parties know. For those cases, we want to remove the cognitive load from our application teams and let them emit and consume messages without the hassle of learning all the specifics of the selected message broker. By reducing the cognitive load required on developers to learn specific technologies, you can onboard less experienced developers and let experts take care of the details. Similar to databases, we can use service identity to control which services can connect, read, and write messages from a message broker. The same principles apply.

3.1.5 Dealing with edge cases (the remaining 5%)

There is always more than one good reason to add libraries to your application's services. Sometimes these libraries will give you the ultimate control over how to connect to vendor-specific components and functionalities. Other times, we add libraries because it is the easiest way to get started or because we are instructed to do so. Someone in the organization decided to use PostgreSQL, and the fastest way to connect and use it is to add the PostgreSQL driver to our application code. We usually don't realize that we are coupling our application to that specific PostgreSQL version. For edge cases, or to be more specific, scenarios where you need to use some vendor-specific functionality, consider wrapping up that specific functionality as a separate unit from all the generic functionality you might consume from a database or message broker.

I've chosen to use async messaging as an example in figure 3.12, but the same applies to databases and credential stores. If we can decouple 95% of our services to use generic capabilities to do their work and encapsulate edge cases as separate units, we reduce the coupling and the cognitive load on new team members tasked to modify these services. Service A in figure 3.12 is consuming a message API provided by the platform team to consume and emit messages asynchronously. We will look deeper into this approach in the next section. But more importantly, the edge cases, where we need to use some Kafka-specific features, for example, are extracted into a separate service that Service A can still interact with using HTTP or GRPC. Notice that the messaging API also uses Kafka to move information around. Still, for Service A, that is no longer relevant, because a simplified API is exposed as a platform capability.

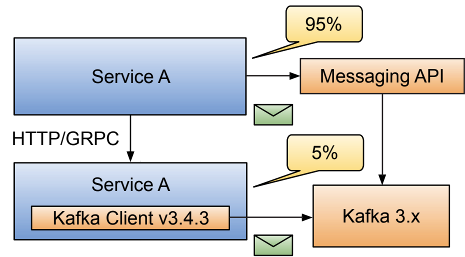


Figure 3.12 Common vs. edge cases encapsulation

When we need to change these services, 95% of the time, we don't need team members to worry about Kafka. The messaging API removes that concern from our application development teams. For modifying Service Y, you will need Kafka experts, and the Service Y code will need to be upgraded if Kafka is upgraded because it directly depends on the Kafka client. For this book, platform engineering teams should focus on trying to reduce the cognitive load on teams for the most common cases while at the same time allowing teams to choose the appropriate tool for edge cases and specific scenarios that don't fit the common solutions.

The following section will look at some approaches to address some of the challenges we have been discussing. However, keep in mind that these are generic solutions, and further steps may be required within your own specific context.

3.2 Standard APIs to separate applications from infrastructure

What about if we encapsulate all these common functionalities (storing and reading data, messaging, credential stores, resiliency policies) into APIs that developers can use from within their applications to solve common challenges while, at the same time, enabling the platform team to wire infrastructure in a way that doesn't require the application's code to change? In figure 3.13 we can see the same services, but instead of adding dependencies to interact with infrastructure, they use HTTP/GRPC requests.

Suppose we expose a set of HTTP/GRPC APIs that our applications services can consume. In that case, we can remove vendor-specific dependencies from our application code and consume these services using standard HTTP or GRPC calls.

This separation between application services and platform capabilities enables separate teams to handle different responsibilities. The platform can evolve independently

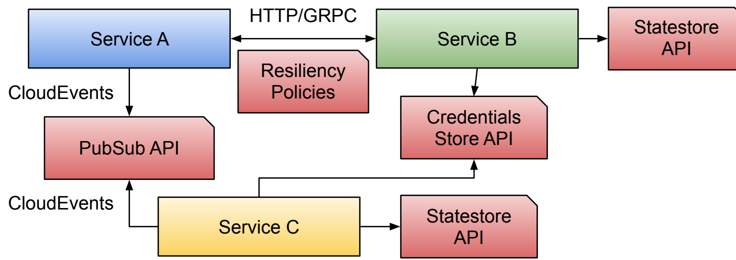


Figure 3.13 Platform capabilities as APIs

from applications, and application code will now only depend on the platform capabilities interfaces but not the version of the components running under the hood. Figure 3.14 shows the separation between application code (our three services) managed by application development teams and platform capabilities that are managed by the platform team.

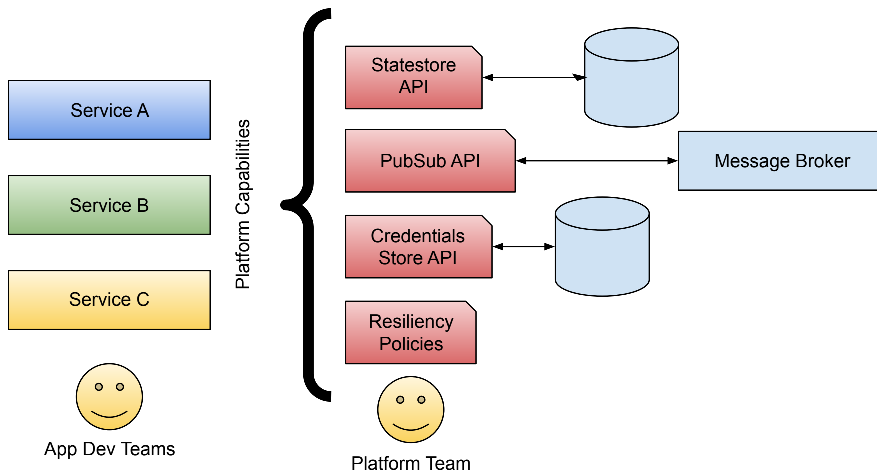


Figure 3.14 Decoupling responsibilities from app dev teams and platform capabilities

When using an approach like the one suggested here, the platform team can expand the platform capabilities, introducing new services for application development teams. More importantly, they can do so without affecting the existing applications or forcing them to release new versions. This enables teams to decide when to release new versions of their services based on their features and the capabilities that they want to consume.

By following this approach, the platform team can make new capabilities available for services to use and promote best practices. Because these platform capabilities are accessible to all services, they can promote standardization and implement best

practices behind the covers. Each team can decide which capabilities are needed to solve their specific problems based on the available ones. If capabilities are correctly versioned, teams can decide how and when to upgrade to the latest version, allowing teams to move at their own pace without the platform pushing every team to upgrade whenever a new version is available.

For the sake of argument, imagine that the platform team decides to expose a consistent feature flagging capability to all the services. Using this capability, all services can consistently define and use feature flags without adding anything to their code except the feature flag conditional checks. Teams then can manage, visualize, and toggle on and off all their flags consistently. A capability like feature flags introduced and managed by the platform team directly affects developers' performance, because they don't need to worry about defining how feature flags will be handled under the hood (persistence, refresh, consistency, etc.), and they know for sure that they are doing things aligned with other services.

Figure 3.15 shows how the platform team can add extra capabilities, like, for example, feature flags, directly enabling teams to use this new capability uniformly in all the services. No new dependencies are needed.

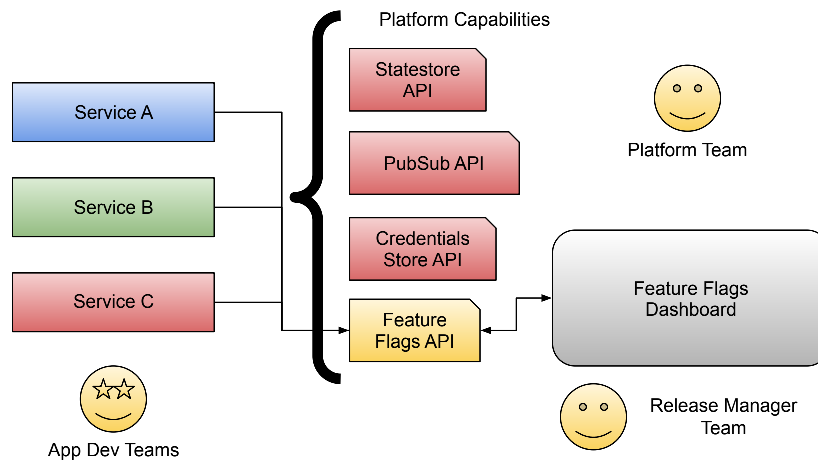


Figure 3.15 Enabling teams by providing consistent and unified capabilities such as feature flags

Before moving forward, here's a word of caution. Let's look at some challenges that you will face when externalizing capabilities like APIs, as suggested in the previous figure.

3.2.1 Exposing platform capabilities challenges

Externalizing APIs for teams to use will require, first of all, stable (and versioned) contracts that application teams can trust. When these APIs change, all applications

consuming those APIs will break and must be updated. Platform teams can adopt a non-breaking changes policy that guarantees backward compatibility to teams and their applications. Adopting such policies makes your platform easier to consume, because the platform APIs and contracts are reliable for teams to use.

One of the main advantages of adding dependencies to your application code and, for example, using containers is that for local development, you can always start a PostgreSQL instance using Docker or Docker Compose and connect your application locally to it. If you move toward platform-provided capabilities, you must ensure that you can provide a local development experience for your teams unless your organization is mature enough to always work against remote services.

Another big difference is that the connection between your services and the platform provided APIs will introduce latency and require security by default. Before, calling the PostgreSQL driver APIs was a local call in the same process as your application. HTTPS, or a secure protocol, established the connection to the database itself, but setting that secure channel between your application and the database was the responsibility of the operations team.

It is also essential to recognize all the edge cases we can find when applying this approach to real-life projects. If you want to build these platform capabilities and push for your teams to consume them, you need to make sure that there is always a door open for edge cases so that teams (or even the platform team) aren't forced to make common cases more complex to account for an obscure feature that will be used only 1% of the time. Figure 3.16 shows Services A, B, and C using the capabilities exposed by the platform via the capabilities APIs. Service Y, on the other hand, has very specific requirements for how to connect to the database, and the team maintaining the service has decided to bypass the platform capabilities APIs to connect directly to the database using the database client.

Treating edge cases separately allows Services A, B, and C to evolve separately from the platform components (database, message brokers, credential stores), while Service Y is now heavily dependent on the database that is connecting to and requires a specific version of the client. While this sounds bad, in practice, it is acceptable and should be considered a platform feature. Teams that cannot solve their business problems with the exposed APIs will hate the platform and silently find workarounds. Good platforms (and platform teams) will promote APIs that cover a wide range of use cases, solving and facilitating the implementation of common functionality for application developers. If these APIs are not enough for all teams, documenting and deeply understanding the edge cases leads to new APIs and platform features that the platform team can implement in future versions.

The following section will examine a couple of CNCF initiatives that took these ideas forward and helped us implement the platform capabilities that most of our applications require.

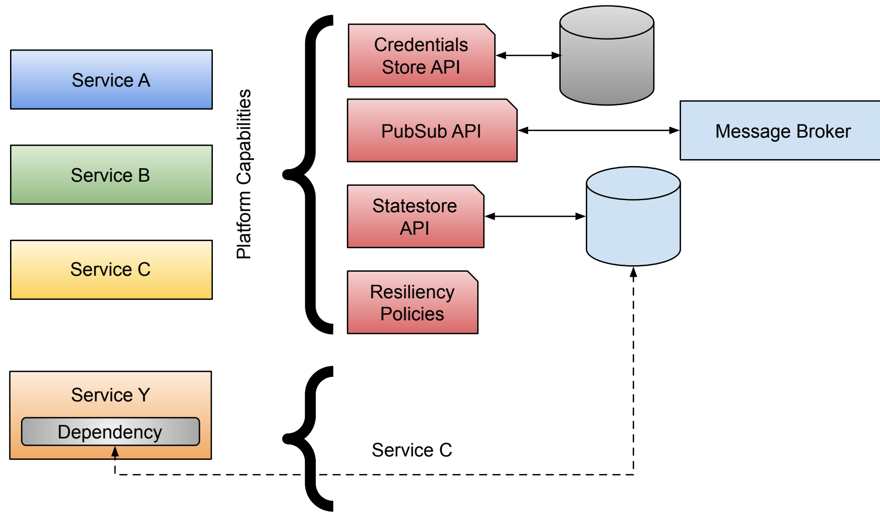


Figure 3.16 Handling edge cases; do not ignore them

3.3 Providing application-level platform capabilities

In this section, we will look at two projects that can save development teams time in standardizing these generic APIs that most of our applications will need. We will start by looking at the Dapr project (https://dapr.io/), what it is, how it works, and what it can do for our development and platform teams. Then we will look into OpenFeature (https://openfeature.dev/), a CNCF initiative that provides our applications with the right abstractions to define and use feature flags without being tied to a specific feature flag provider.

Once we get a bit of an understanding of how these two projects work and complement each other by helping us to provide application-level platform capabilities, we will look into how these projects can be applied to our Conference application, what changes are needed, the advantages of following this approach, and some examples showing edge cases. Let's start with Dapr, our Distributed Application Runtime.

3.3.1 Dapr in action

Dapr provides a set of consistent APIs to solve common and recurrent distributed application challenges. The Dapr project has spent the last four years implementing a set of APIs (called Building Block APIs) to abstract away common challenges and best practices that distributed applications will need 95% of the time. Created by Microsoft in 2019 and donated to the CNCF in 2021, the Dapr project has a large community contributing with extensions and improvements to the project APIs, making it the 10th fastest-growing project in the CNCF of 2023.

Dapr defines a set of building blocks that provide concrete APIs to solve distributed application challenges and swappable implementations that the platform team can configure. If you visit the https://dapr.io website, you will see the list of Building Block APIs, including Service Invocation, State Management, Publish & Subscribe, Secrets Store, Input/Output Bindings, Actors, Configurations Management, and, more recently Workflows. Figure 3.17 shows the Dapr official website describing the current Dapr Building Block APIs that teams can use to build their distributed applications. Check the Dapr Overview page at https://docs.dapr.io/concepts/overview/ for more information about the Dapr project.

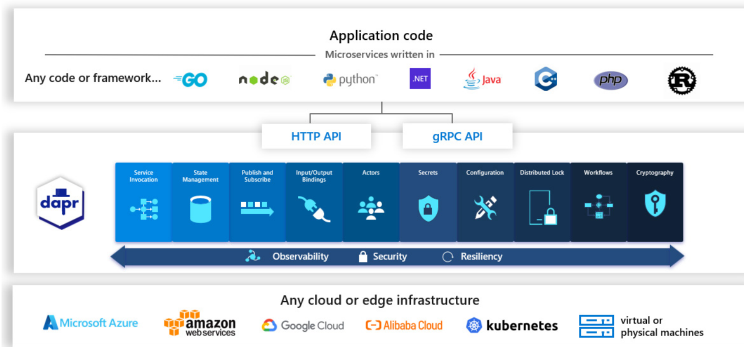


Figure 3.17 Dapr components for building distributed applications

While Dapr does much more than just expose APIs, in this chapter, I wanted to focus on the APIs provided by the project and the mechanisms used by the project to enable applications/services to consume these APIs.

Because this is a Kubernetes book, we will look at Dapr in the context of Kubernetes, but the project can also be used outside of Kubernetes clusters, making Dapr a generic tool to build distributed applications no matter where you are running them. As a side note, Dapr is currently part of Azure Container Apps service (https://azure.microsoft.com/en-us/products/container-apps), where it is configured with another CNCF project KEDA (https://keda.sh/) for autoscaling your distributed applications.

3.3.2 Dapr in Kubernetes

Dapr works as a Kubernetes extension or add-on. You must install a set of Dapr controllers (a Dapr control plane) on your Kubernetes clusters. Figure 3.15 shows Service A deployed in a Kubernetes cluster with Dapr installed. Service A needs to be annotated with two annotations: `dapr.io/enabled: "true"` for the Dapr control plane to be aware of the application and `dapr.io/appid: "service-a"` to use Dapr service identity features.

Once Dapr is installed in your clusters, your applications deployed in the cluster can start using the Dapr APIs by adding a set of annotations to your deployments. This enables the Dapr control plane services to understand that your application wants to use the Dapr APIs, as shown in figure 3.18.

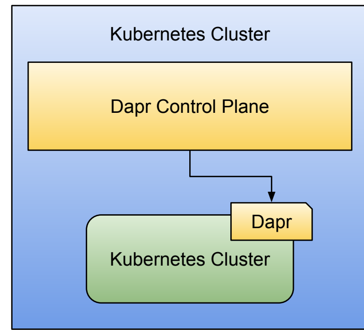


Figure 3.18 The Dapr control plane monitor for applications with Dapr annotations

By default, Dapr will make all the Dapr APIs available to your applications/services as a sidecar (`daprd` is the container that will run beside your applications/services) that runs beside your application's containers. Using the sidecar pattern, we enable our application to interact with a co-located (localhost) API that runs very close to the application's container and avoids network round trips. Figure 3.19 shows how the Dapr control plane injects the `daprd` sidecar into the application annotated with the Dapr annotations. This enables the application to access the configured Dapr components.

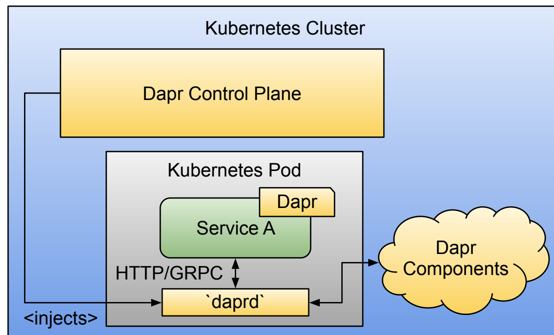


Figure 3.19 Dapr sidecars (daprd) give your applications local access to Dapr components.

Once the Dapr sidecar is running beside your applications/service container, it can use the Dapr APIs by sending requests (using HTTP or GRPC) to `localhost`, because the `daprd` sidecar runs inside the same pod as the application, sharing the same networking space.

Now for the Dapr APIs to be of some use, the platform team needs to configure the implementation (or backing mechanisms named Dapr components) for these APIs to work. For example, if you want to use the Statestore Dapr APIs (https://docs.dapr.io/operations/components/setup-state-store/) from your applications/services, you must define and configure a Statestore component.

When working with Dapr on Kubernetes, you configure a Dapr component specification using a Kubernetes resource. For example, you can configure a Statestore

Dapr component to use Redis. See listing 3.1 for an example Dapr component resource definition.

Listing 3.1 Dapr Statestore component definition

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: statestore
spec:
  type: state.redis
  version: v1
  metadata:
  - name: keyPrefix
    value: name
  - name: redisHost
    value: redis-master:6379
  - name: redisPassword
    secretKeyRef:
      name: redis
      key: redis-password
auth:
  secretStore: kubernetes
```

The Statestore component APIs support different implementations that you can find at https://docs.dapr.io/reference/components-reference/supported-state-stores/. For this example, we are setting up the state.redis implementation. (annotates `type: state.redis`)

By setting the redisHost, the platform team can define where the Redis instance is located. There is no need for this instance to be inside the Kubernetes cluster; it can be any accessible Redis instance. (annotates `- name: redisHost`)

The redisPassword property (required by the state.redis implementation) can use, as shown in this example, a Kubernetes Secret reference to fetch the password. (annotates `- name: redisPassword`)

If the component resource is available in the Kubernetes cluster, the `daprd` sidecar can read its configurations and connect to the Redis instance for this example. From the application perspective, there is no need to know if Redis is being used or if any other implementation for the Statestore component. Figure 3.20 shows how Dapr components are wired so Service A can use the Statestore component APIs. For this example, by calling a local API, Service A will be able to store and read data from the Redis instance.

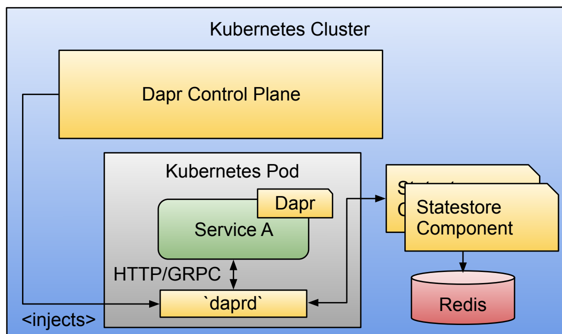


Figure 3.20 Dapr sidecars use component configurations to connect to the component's infrastructure.

Dapr makes it easy to build your application using a local/self-hosted Redis instance but then move it to the cloud where a managed Redis service can be used. No code or dependencies changes are needed, just a different Dapr component configuration.

Do you want to emit and consume messages between different applications? You just need to configure a Dapr PubSub component (https://docs.dapr.io/operations/components/setup-pubsub/) and its implementation. Now your service can use a local API to emit asynchronous messages. Do you want to make all service interactions (including infrastructure) calls resilient? You can use Dapr resiliency policies (https://docs.dapr.io/operations/resiliency/policies/) to avoid writing custom logic inside your application code.

Figure 3.21 shows how Service A and Service B can send requests to each other using the Service Invocation APIs, in contrast to calling the other service directly. Using these APIs (that send traffic through the `daprd` sidecar) enables the platform team to configure resiliency policies at the platform level, uniformly without adding any dependencies or changing the application code.

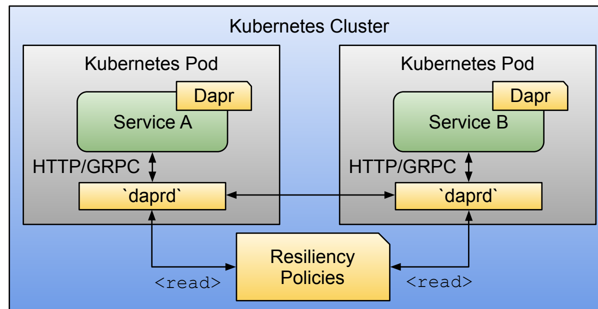


Figure 3.21 Dapr-enabled services can use service-to-service communications and resiliency policies.

OK, so the Dapr control plane will inject the Dapr sidecars (`daprd`) to the applications that are interested in using Dapr components. But how does this look from the application point of view?

3.3.3 Dapr and your applications

If we go back to the example introduced in the previous section where Service A wants to use the Statestore component to store/read some data from persistent storage like Redis, the application code is straightforward. No matter which programming language you use, as soon as you know how to create HTTP or GRPC requests, you have all you need to work with Dapr.

For example, to store data using the Statestore APIs your application code needs to send an HTTP/GRPC request to the following endpoint:

```
http://localhost:<DAPR_HTTP_PORT>/v1.0/state/<STATESTORE_NAME>
```

Using `curl`, the request will look like this, where `-d` shows the data we want to persist and `3500` is the default `DAPR_HTTP_PORT` and our Statestore component is called `statestore`:

```
> curl -X POST -H "Content-Type: application/json"
➥-d '[{ "key": "name", "value": "Bruce Wayne"}]'
➥http://localhost:3500/v1.0/state/statestore
```

To read the data that we have persisted, instead of sending a POST request, we just write a GET request. With `curl`, it would look like this:

```
curl http://localhost:3500/v1.0/state/statestore/name
```

Usually, you will not be using `curl` from inside your applications. You will use your programming language tools to write these requests. So, if you use Python, Go, Java, .NET, or JavaScript, you can find tutorials online on using popular libraries or built-in mechanisms to write these requests.

Another option is to use one of the Dapr SDKs (Software Development Kits) available for different programming languages. Adding the Dapr SDK to your application as a dependency allows you to make your developers' lives easier, so they don't need to craft HTTP or GRCP requests manually. It is crucial to notice that while you are now adding a new dependency to your application, this dependency is optional and only used as a helper to speed things up, because this dependency is not tied to any of the infrastructural components that the Dapr APIs are interacting with.

Check the Dapr website for examples of how your code will look if you use the Dapr SDK. For example, for a multi-programming language example on how to use the Statestore component using the SDKs, you can visit https://docs.dapr.io/getting-started/quickstarts/statemanagement-quickstart/.

While I decided to focus on Dapr for API abstractions, Dapr offers much more. By allowing platform teams to swap Dapr components implementations, applications can be moved across cloud providers without needing to change any application code. By default, the entire system is observable (https://docs.dapr.io/operations/observability/), secure (https://docs.dapr.io/operations/security/), and resilient (https://docs.dapr.io/operations/resiliency/), as Dapr sidecars will enforce service identity and the rules specified by the platform team, while at the same time extracting metrics from all the Dapr-enabled applications and components. I recommend platform teams familiarize themselves with the Dapr Project, as the project was built to solve common challenges that teams will face when working with distributed applications. Check section 3.3.5 of this chapter to see how we can make our Conference application Dapr-enabled. Now let's talk a bit about feature flags.

3.3.4 Feature flags in action

Feature flags enable teams to release software that includes new features without making those features available immediately. New features can be hidden behind feature flags that can be enabled later. In other words, feature flags allow teams to keep deploying new versions of their services or applications, and once these applications are running, features can be turned on or off based on the company's needs.

Compared to application-level APIs, which directly enabled developers with out-of-the-box behaviors to implement complex features, feature flags can enable other teams that make business-related decisions on when features should be enabled to customers.

While most companies might build mechanisms to implement feature flags, it is a well-recognized pattern to be encapsulated into a specialized service or library. In the Kubernetes world, you can consider using `ConfigMaps` as the simplest way to parameterize your containers. As soon as your container can read environment variables to turn on and off features, you are ready to go. We used this approach in chapter 1 with the `FEATURE_DEBUG_ENABLED=true` environment variable.

Unfortunately, this approach is too simplistic and doesn't work for real-world scenarios. First, one of the main reasons is that your containers will need to be restarted to reread the content of the ConfigMap if it changes. Second, you might need many flags for your different services, so you might need multiple ConfigMaps to manage your feature flags. Third, if you use environment variables, you will need to develop a convention to define each flag's status, default values, and type, because you cannot get away with just defining variables as plain strings.

Because this is a well-understood problem, several companies have come up with tools and managed services like LaunchDarkly (https://launchdarkly.com/) and Split (https://www.split.io/product/feature-flags/), among others, which enable teams to host their feature flags in a remote service that offers simplified access to view and modify feature flags without the need for technical knowledge. For each of these services, to fetch and evaluate complex feature flags, you will need to download and add a dependency to your applications. As each feature flag provider will offer different functionalities, switching between providers would require many changes.

OpenFeature (https://openfeature.dev/) is a CNCF initiative to unify how feature flags can be consumed and evaluated in cloud-native applications. In the same way that Dapr is abstracting how to interact with Statestores (storing and reading state) or PubSub (async message brokers) components, OpenFeature provides a consistent API to consume and evaluate feature flags no matter which features flag provider we use.

In this short section, we will look at a simple example using a ConfigMap to hold a set of feature flag definitions. We will also be using the `flagd` implementation provided by OpenFeature, but the beauty of this approach is that you can then swap the provider where the feature flags are stored without changing any single line of code in your application.

Figure 3.22 shows a simple application including the OpenFeature SDK that is configured to connect to an OpenFeature provider—in this case, `flagd`, which is in charge of hosting our feature flag definitions.

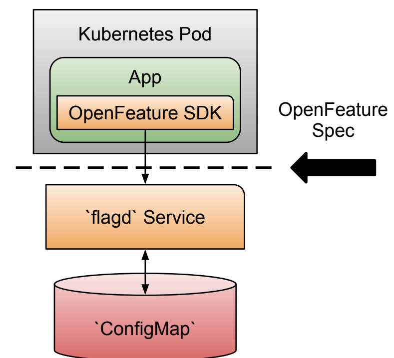


Figure 3.22 Consuming and evaluating feature flags from our application services

For this simple example, our app is written in Go and uses the OpenFeature Go SDK to fetch feature flags from the `flagd` service. The `flagd` service for this example is configured to watch a Kubernetes `ConfigMap` that contains some complex feature flags definitions.

While this is a simple example, it allows us to see how a service like `flagd` can allow us to abstract away all the complexities of the storage and implementation of the mechanisms needed to provide a feature flag capability as part of our platform.

In contrast with Dapr, the OpenFeature SDK is needed because we are not only fetching the feature flag definitions but also performing evaluations that can involve complex feature flags.

You can hook every service in your application to connect to an OpenFeature provider to perform feature flag evaluations. An important difference with just using plain ConfigMaps is that by using OpenFeature, containers don't need to be restarted to fetch values if they change; that is now the responsibility of the OpenFeature flag provider.

In the next section, we look at how to apply both Dapr and OpenFeature to the Conference application walking skeleton.

3.3.5 Updating our Conference application to consume application-level platform capabilities

Conceptually and from a platform perspective, it will be great to consume all these capabilities without the platform leaking which tools are used to implement different behaviors. This would enable the platform team to change/swap implementations and reduce the cognitive load from teams using these capabilities. But as we discussed with Kubernetes, understanding how these tools work, their behaviors, and how their functionalities were designed influences how we architect our applications and services. In this last section of the chapter, I wanted to show how tools like Dapr and OpenFeature can influence your application architecture and, at the same time, show how these tools offer building blocks to create higher-level abstractions to reduce consumers' cognitive load.

For our Conference application, we can use the following Dapr components, so let's focus on these:

- *Dapr Statestore component:* Using the Statestore component APIs enables us to remove the Redis dependency from the Agenda service included in the Conference application. If, for some reason, we want to swap Redis for another persistent store, we will be able to do so without changing any of the application code.
- *Dapr PubSub component:* For emitting events, we can replace the Kafka client from all the services to use the PubSub component APIs, allowing us to test different implementations, such as RabbitMQ or a cloud provider service to exchange asynchronous messages between applications.
- *Dapr service-to-service invocations and Dapr resiliency policies:* If we use the service-to-service invocation APIs, we can configure resiliency policies between the services

without adding a library or custom code to our services code. By default, all services have resiliency policies defined if no custom configuration is provided.

While we can choose to use the Statestore component APIs also to remove the PostgreSQL dependency in our Call for Proposals service, I have chosen not to do so to support the use of SQL and PostgreSQL features that the team needed for this service. When adopting Dapr, you must avoid pushing for an "all or nothing" approach.

Let's look at how the application will change if we decide to use Dapr. Figure 3.23 shows the application services using Dapr components, because all the services are annotated to use Dapr, and the `daprd` sidecar has been injected all services. Once the PubSub and Statestore components have been configured, they can be accessed by the Call for Proposals service, Agenda service, and Notifications service. Finally, a Dapr Subscription pushes events to the Frontend application.

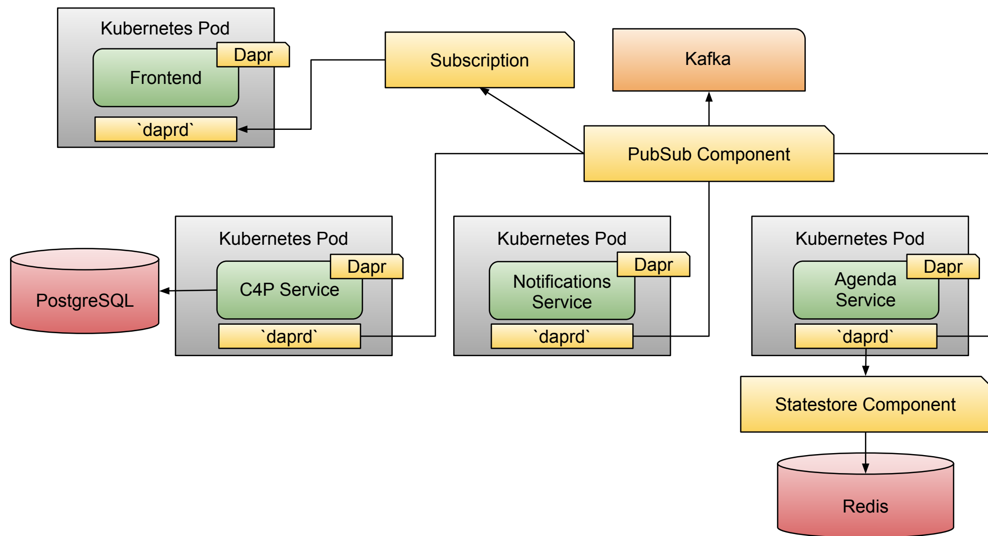


Figure 3.23 Using Dapr components for our walking skeleton / Conference application

Resiliency policies can also be configured and defined for the Call for Proposals service to interact with the agenda and notifications services, as shown in figure 3.24.

Dapr applies default resiliency policies if we don't configure any. These resiliency policies also apply to, for our example, contacting the `statestore` and `pubsub` components. This means that not only our service-to-service invocations are resilient, but every time our application code wants to interact with infrastructure components such as databases, caches and message brokers, the resiliency policies will kick in.

The application code needs to change slightly, because when services want to talk to each other, they need to use the Dapr API to use resiliency policies.

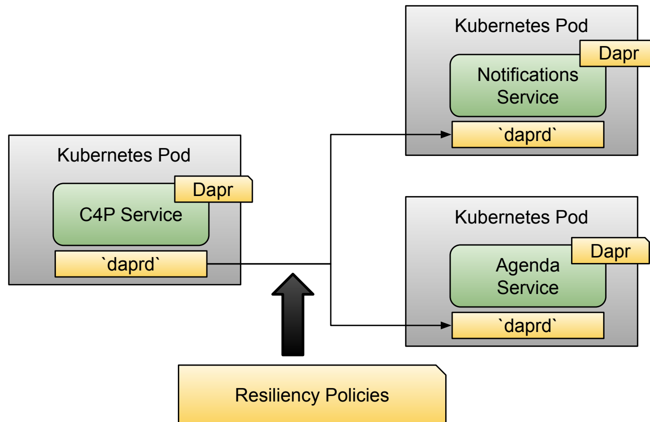


Figure 3.24 Service-to-service interactions can be handled by the `daprd` sidecar, allowing platform teams to define different resiliency policies.

Finally, because we wanted to enable all the services to use feature flags, each service now includes the OpenFeature SDK, which allows the platform team to define which feature flag implementation all services will use.

In figure 3.25 each service has included the OpenFeature SDK library and is configured to point to the `flagd` service that enables the platform team to configure the mechanism used to store, fetch, and manage all the feature flags used by all the services.

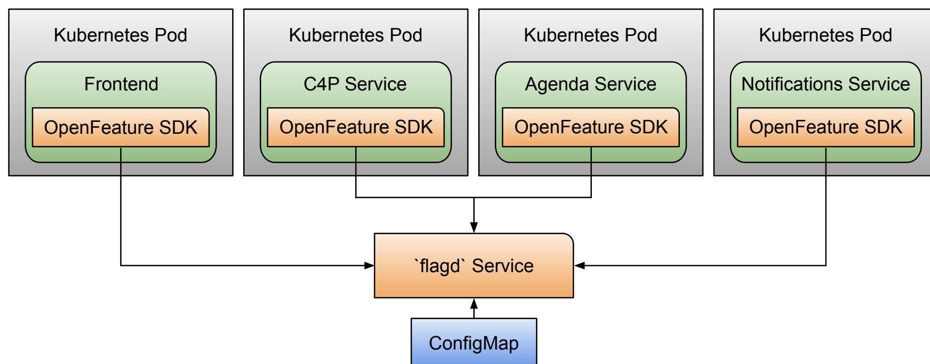


Figure 3.25 Services using the `flagd` feature flag provider.

Using the OpenFeature SDK, we can change the feature flag provider without changing our application code. The OpenFeature SDK now standardizes all the feature flag consumption and evaluation of our service code.

While in Dapr, using the SDK is optional (because you can always craft your HTTP or GRPC requests by hand), in OpenFeature, the scenario is a bit more complicated. because the SDKs provide some of the evaluation logic to understand which type each flag is and if it is on or off.

The step-by-step tutorial (https://github.com/salaboy/platforms-on-k8s/tree/v2.0.0/chapter-7) deploys version v2.0.0 of the conference application that uses Dapr and OpenFeature flags to enable application teams to keep evolving the application services. Version v2.0.0 of the application services doesn't include the Kafka or Redis client to interact with infrastructure. These services can be deployed in different environments (including cloud providers) and wired against different implementations of these standard APIs. Figure 3.26 shows the dependencies that we managed to remove for version v2.0.0 of the application using the Dapr component APIs.

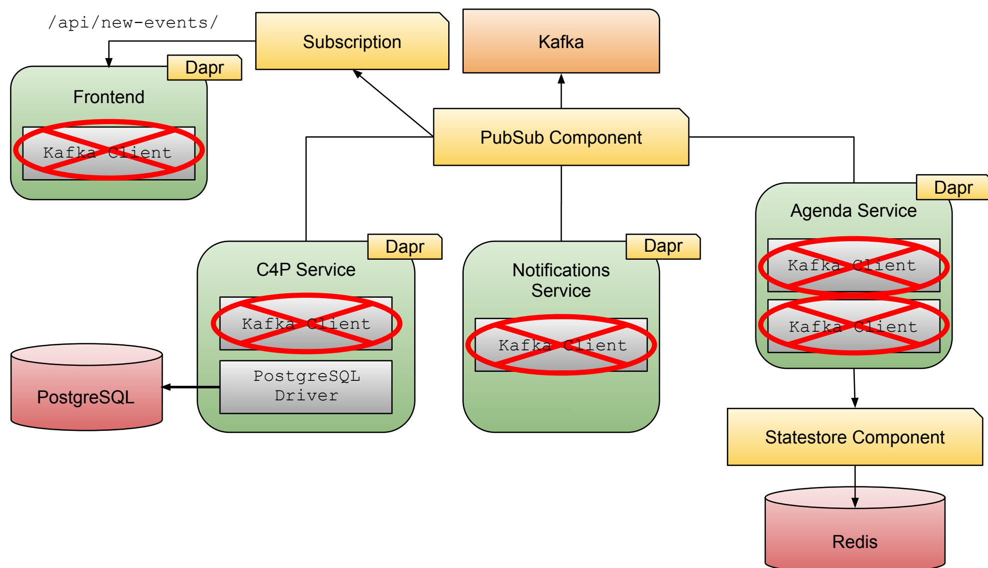


Figure 3.26 Kafka and Redis client removed from services' dependencies.

From a platform perspective, three Kubernetes resources are defined by the Dapr Statestore component, the Dapr PubSub component, and the Dapr Subscription.

We've already seen in section 3.3.1 how a Dapr Statestore component is defined. In listing 3.2, we can see how a PubSub component is defined, in this case selecting the type to be `pubsub.kafka`, which uses the Kafka instance installed using Helm.

Listing 3.2 Dapr PubSub component definition

```
apiVersion: dapr.io/v1alpha1
kind: Component
```

```
metadata:
  name: conference-pubsub
spec:
  type: pubsub.kafka
  version: v1
  metadata:
  - name: brokers
    value:  kafka.default.svc.cluster.local:9092
  - name: authType
    value: "none"
```

We need to specify the Kafka brokers available for the PubSub component to connect to.

By default, the Kafka Helm Chart provided by Bitnami doesn't require authentication.

You can find all the supported PubSub implementations on the official Dapr website (https://docs.dapr.io/reference/components-reference/supported-pubsub/). Finally, the Dapr Subscription resources allow us to declaratively configure subscriptions to PubSub components and route events to the application's endpoints, as shown in listing 3.3.

Listing 3.3 Dapr Subscription definition

```
apiVersion: dapr.io/v1alpha1
kind: Subscription
metadata:
  name: frontend-subscription
spec:
  pubsubname: conference-pubsub
  topic: events-topic
  route: /api/new-events/
scopes:
- frontend
```

The PubSub component where we want to register the subscription

The topic inside the PubSub component that the subscription will listen to

The route where the events received in the topic will be forwarded to by Dapr

scopes allows us to define which Dapr applications are allowed to receive events from this subscription. In this case the only consumer is the frontend application. Scopes heavily relies on service identity to block messages from being forwarded to unauthorized services.

From an application developer perspective, the changes in v2.0.0 use the Dapr Go SDK to call the Dapr components API. For example, to read the state from the Statestore component, the Agenda service performs the call shown in listing 3.4.

Listing 3.4 Getting state from a Statestore using the Dapr SDK

```
s.APIClient.GetState(ctx,
STATESTORE_NAME,
KEY,
nil)
```

To store state, you only need to provide the Statestore component name configured in Dapr.

You also need to provide the key that you want to retrieve from the Statestore.

The APIClient instance here is just a Dapr client that provides helpers to interact with the DAPR HTTP and GRPC APIs. Similarly, to store state, you can use the `SaveState` method; see listing 3.5.

Listing 3.5 Saving state from a Statestore using the Dapr SDK

```
s.APIClient.SaveState(ctx,
STATESTORE_NAME,
KEY,
jsonData,
nil)
```

- STATESTORE_NAME: Same as before, we need to provide the Statestore component name. Notice that applications can have access to multiple Statestore components for different purposes.
- KEY: The KEY will be used to store the payload, so it can then be retrieved by calling GetState method.
- jsonData: The state is sent to the APIs as a JSON payload.

Finally, and following exactly the same approach, applications can publish events to the PubSub component by using the API shown in listing 3.6.

Listing 3.6 Publishing an event using the Dapr SDK

```
s.APIClient.PublishEvent(ctx,
PUBSUB_NAME,
PUBSUB_TOPIC,
eventJson)
```

- PUBSUB_NAME: To publish an event, we need to specify the Dapr PubSub component that we want to use as well as the topic.
- PUBSUB_TOPIC: The topic allows us to divide the PubSub component into different logical buckets that the application can use to exchange events and messages.
- eventJson: The event payload is expressed as JSON.

On the OpenFeature side, the feature flag configurations are defined inside a ConfigMap (https://github.com/salaboy/platforms-on-k8s/blob/v2.0.0/conference-application/helm/conference-app/templates/openfeature.yaml#L49). The tutorial shows three different feature flags added to the Conference application to control frontend and backend features. By modifying the ConfigMap that contains the flag definitions, we can change the application behavior without the need to restart any container. The `eventsEnabled` feature flag in listing 3.7 shows a feature flag of type Object that contains properties for each of the services. By defining different variants, we can codify profiles, allowing us to define complex scenarios.

Listing 3.7 Feature flag definitions, including variants

```
"eventsEnabled": {
"state": "ENABLED",
"variants": {
  "all": {
    "agenda-service": true,
```

```
    "notifications-service": true,
    "c4p-service": true
  },
  "decisions-only": {
     "agenda-service": false,
     "notifications-service": false,
     "c4p-service": true
   },
   "none": {
     "agenda-service": false,
      "notifications-service": false,
      "c4p-service": false
    }
  },
  "defaultVariant": "all"
```

Listing 3.7 shows an Object feature flag that defines three variants: `all`, `decisions-only`, and `none`. By changing the `defaultVariant` property, we can change which profile is selected, in this case to enable and disable which services will emit events. Inside the Agenda service source code, we use the OpenFeature GO SDK to fetch and evaluate the flag, as shown in the following listing.

Listing 3.8 Feature flag evaluation using OpenFeature SDK

```
s.FeatureClient.ObjectValue(ctx, "eventsEnabled",
EventsEnabled{},
openfeature.EvaluationContext{})
```

Listing 3.8 shows using the OpenFeature client to fetch the `eventsEnabled` feature. The `EventsEnabled{}` struct is the default value that should return in case there is a problem fetching the feature flag. Finally, the `EvaluationContext` struct allows you to add extra parameters for OpenFeature to evaluate the flag for more complex scenarios.

You can find the differences between `v1.0.0` and `v2.0.0` by comparing the `main` branch and the `v2.0.0` branch in the application repository at https://github.com/salaboy/platforms-on-k8s/compare/v2.0.0. At the same time, the platform team is free to configure and wire up application infrastructure and define all the backing mechanisms and implementations for feature flags, storage, messaging, configuration, managing credentials, resiliency, and other common challenges they don't want to expose directly to developers.

3.4 Linking back to platform engineering

In this chapter, we have seen how to enable teams with platform-wide capabilities in the form of APIs. We aim to speed up their process of writing and delivering complex software by providing teams with common and standard APIs to solve everyday challenges when creating distributed applications and mechanisms such as feature flags.

By separating application infrastructure from the application's code, we not only remove dependencies from our services, but we also enable the platform team to decide how to configure application infrastructure components and how the services will connect to them. If different environments require different implementations, the platform team can work behind the APIs to provide different configurations for different scenarios.

Figure 3.27 shows how we can reduce friction and dependencies related to the application infrastructure. This allows our application's services to work in various environments the platform team can control. Using projects like Dapr, you also gain portability of your applications across cloud providers, consistent APIs that can be used from any programming language, and you enable teams to bring their applications from a local development environment to production environments, allowing the platform team to wire up the infrastructure that your application needs to work. With feature flags, we enable developers to keep releasing software by masking features behind feature flags that can be turned on and off, enabling other teams closer to customers (like product teams) to decide when these features should be exposed.

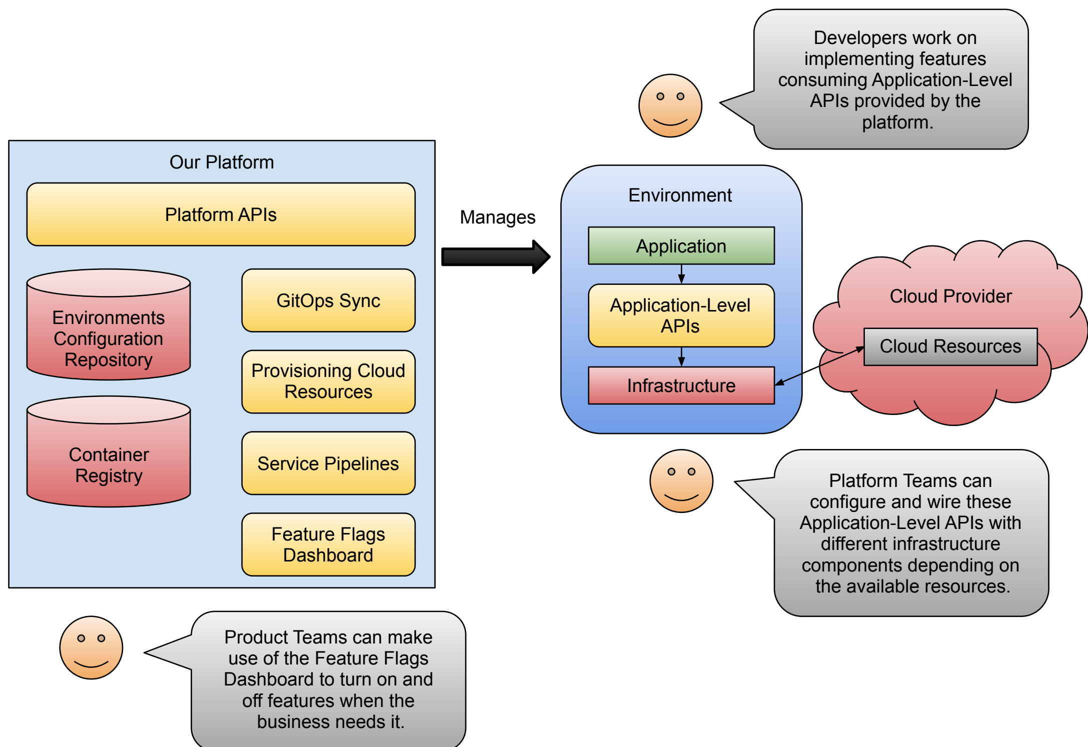


Figure 3.27 Consistent capabilities across environments enable smoother paths to production.

By providing consistent capabilities across environments, we enable easier paths to production, because we can control which features are exposed to customers after releasing the new version to production. Developers can keep building features relying on platform-provided application-level APIs without knowing where the available infrastructure is or which versions of databases and message brokers are used in the production environment.

For the sake of space, topics such as observability, metrics, and logs, service meshes haven't been covered in these sections, because these capabilities are currently more mature and more operations-focused. I've decided to focus on capabilities that build on top of the operation and infrastructure teams to speed up development teams and solve everyday challenges. Platform teams will define which observability stack they will use across environments early and how this data can be available to developers troubleshooting problems. Service meshes and certificate rotation tools for mutual TLS (encryption between services) are often discussed in these conversations because these are topics that development teams will not want to spend time on and should be provided at the platform level. Figure 3.28 shows how our platform is responsible for defining, fetching, and aggregating data from the tools available inside each environment. Our platform should provide a single entry point to understand what is happening in different environments and provide teams with enough information to troubleshoot problems and access the tools the organization needs to deliver software to customers.

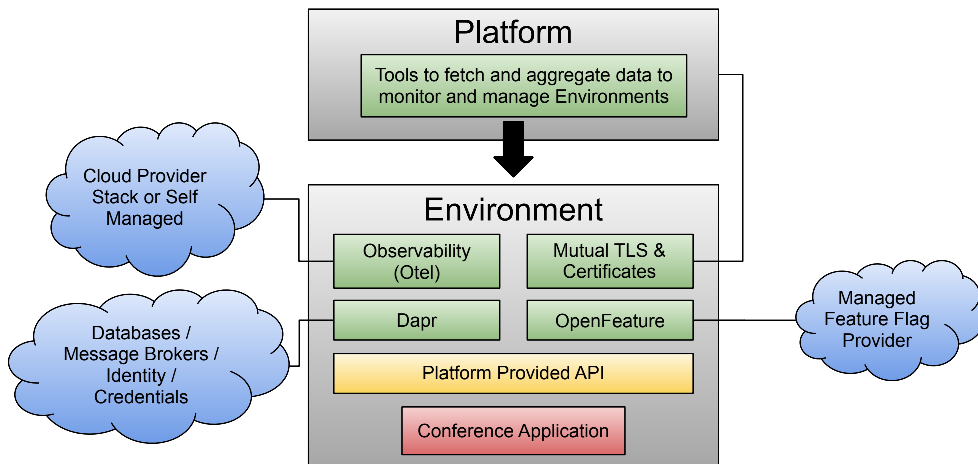


Figure 3.28 The platform that we build needs to define, manage, and monitor the tools available in each environment.

The next chapter will explore tools to enable teams to experiment while releasing software. Along the same lines of using feature flags, we will dig deeper into how to use

different release strategies to catch problems earlier in the release process and enable stakeholders to try different approaches simultaneously.

Summary

- Moving dependencies to application infrastructure enables application code to stay agnostic to platform-wide upgrades. Separating the lifecycle of the applications and the infrastructure enables teams to rely on stable APIs instead of dealing with provider-specific clients and drivers for everyday use cases.
- Treating edge cases separately allows experts to make more conscious cases based on their application requirements. This also allows common scenarios to be handled by less experienced team members, who don't need to understand the specifics of tools like vendor-specific database features or low-level message broker configurations when they only want to store or read data or emit events from their application's code.
- Dapr solves common and shared concerns when building distributed applications. Developers that can write HTTP/GRPC requests can interact with infrastructure that the platform team will wire up.
- Feature flags enable developers to keep releasing software by masking new features behind feature flags that can be turned on and off.
- OpenFeature standardizes the way applications consume and evaluate feature flags. Relying on OpenFeature abstractions allows platform teams to decide where feature flags are stored and how they are managed. Different providers can offer non-technical people dashboards where they can see and manipulate flags.
- If you followed the step-by-step tutorial, you gained hands-on experience in using tools like Dapr and OpenFeature in the context of a cloud-native application composed of four services that interact with SQL and NoSQL databases and a message broker like Kafka. You also modified feature flags on a running application to change its behavior without restarting any of its components.