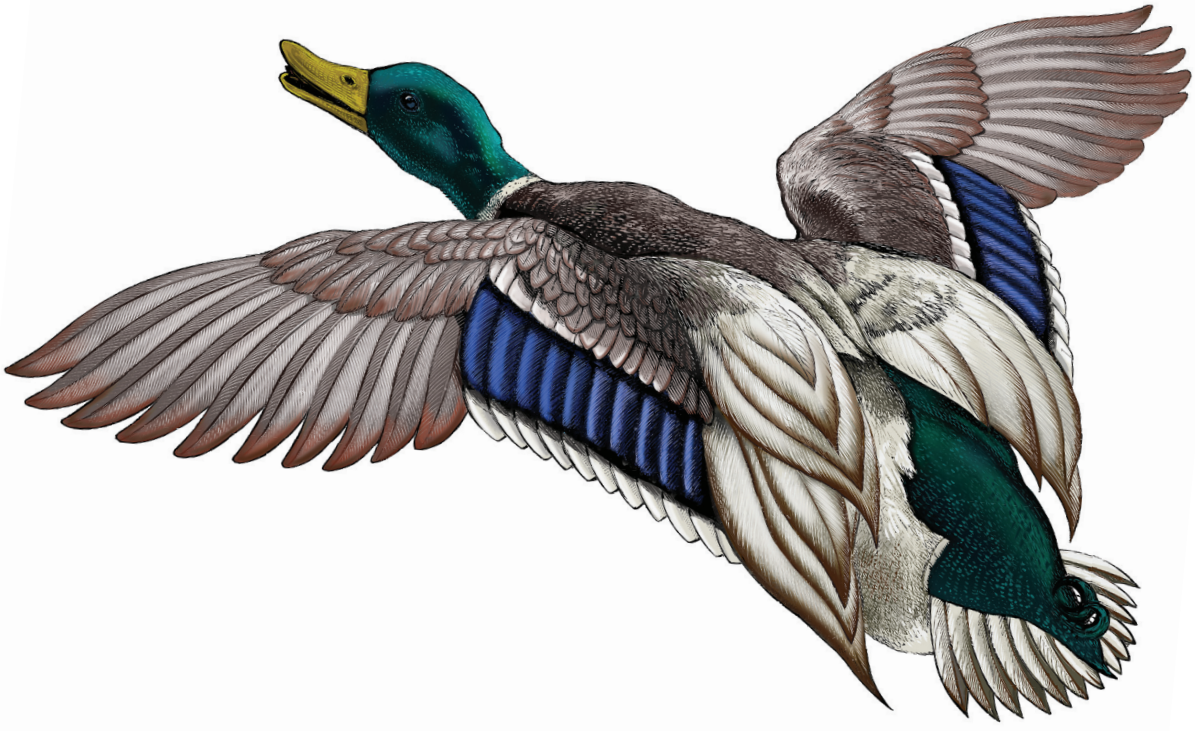
O'REILLY®

# Kubernetes Best Practices

## Blueprints for Building Successful Applications on Kubernetes



Brendan Burns, Eddie Villalba,
Dave Strebel & Lachlan Evenson

O'REILLY®

# Kubernetes Best Practices

In this practical guide, four Kubernetes professionals with deep experience in distributed systems, enterprise application development, and open source will guide you through the process of building applications with this container orchestration system. Based on the experiences of companies that are running Kubernetes in production successfully, many of the methods are also backed by concrete code examples.

This book is ideal for those already familiar with basic Kubernetes concepts who want to learn common best practices. You'll learn exactly what you need to know to build your best app with Kubernetes the first time.

- Set up and develop applications in Kubernetes
- Learn patterns for monitoring, securing your systems, and managing upgrades, rollouts, and rollbacks
- Understand Kubernetes networking policies and where service mesh fits in
- Integrate services and legacy applications and develop higher-level platforms on top of Kubernetes
- Run machine learning workloads in Kubernetes

**Brendan Burns** is a distinguished engineer at Microsoft Azure and cofounder of the Kubernetes open source project.

**Eddie Villalba** is a software engineer with Microsoft's Commercial Software Engineering division, focusing on open source cloud and Kubernetes.

**Dave Strebel** is a global cloud native architect at Microsoft Azure focusing on open source cloud and Kubernetes.

**Lachlan Evenson** is a principal program manager on the container compute team at Microsoft Azure.

"Practical hands-on guidance by these experienced professionals will give you deep insights into the fast-moving Kubernetes ecosystem."

—**Bridget Kromhout**
Principal Program Manager, Microsoft

"The reader will get invaluable insights in operating Kubernetes at different scales, topologies, domains, and use cases straight from the source."

—**Bilgin Ibryam**
Coauthor of *Kubernetes Patterns*,
Principal Architect at Red Hat

"Highly recommended set of practical recipes that provide solutions to a wide variety of real-world Kubernetes challenges."

—**Roland Huß**
Coauthor of *Kubernetes Patterns*,
Principal Software Engineer at Red Hat

CLOUD COMPUTING

US $59.99 CAN $75.99
ISBN: 978-1-492-05647-8


Twitter: @oreillymedia
facebook.com/oreilly

# Kubernetes Best Practices

*Blueprints for Building Successful Applications on Kubernetes*

*Brendan Burns, Eddie Villalba, Dave Strebel, and Lachlan Evenson*

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

**Kubernetes Best Practices**
by Brendan Burns, Eddie Villalba, Dave Strebel, and Lachlan Evenson



Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.



**Acquisitions Editor:** John Devins
**Development Editor:** Virginia Wilson
**Production Editor:** Elizabeth Kelly
**Copyeditor:** Charles Roumeliotis
**Proofreader:** Sonia Saruba

**Indexer:** WordCo Indexing Services, Inc.
**Interior Designer:** David Futato
**Cover Designer:** Karen Montgomery
**Illustrator:** Rebecca Demarest

November 2019: First Edition

**Revision History for the First Release**
2019-11-12: First Release
2020-07-10: Second Release

See *https://www.oreilly.com/catalog/errata.csp?isbn=0636920273219* for release details.



978-1-492-05647-8

[LSI]

# Table of Contents

# Preface

## Who Should Read This Book

Kubernetes is the de facto standard for cloud native development. It is a powerful tool that can make your next application easier to develop, faster to deploy, and more reliable to operate. However, unlocking the power of Kubernetes requires using it correctly. This book is intended for anyone who is deploying real-world applications to Kubernetes and is interested in learning patterns and practices they can apply to the applications that they build on top of Kubernetes.

Importantly, this book is not an introduction to Kubernetes. We assume that you have a basic familiarity with the Kubernetes API and tools, and that you know how to create and interact with a Kubernetes cluster. If you are looking to learn Kubernetes, there are numerous great resources out there, such as *Kubernetes: Up and Running* (O'Reilly) that can give you an introduction.

Instead, this book is a resource for anyone who wants to dive deep on how to deploy specific applications and workloads on Kubernetes. It should be useful to you whether you are about to deploy your first application onto Kubernetes or you've been using Kubernetes for years.

## Why We Wrote This Book

Between the four of us, we have significant experience helping a wide variety of users deploy their applications onto Kubernetes. Through this experience, we have seen where people struggle, and we have helped them find their way to success. When sitting down to write this book, we attempted to capture these experiences so that many more people could learn by reading the lessons that we learned from these real-world experiences. It's our hope that by committing our experiences to writing, we can scale our knowledge and allow you to be successful deploying and managing your application on Kubernetes on your own.

## Navigating This Book

Although you might read this book from cover to cover in a single sitting, that is not really how we intended you to use it. Instead, we designed this book to be a collection of standalone chapters. Each chapter gives a complete overview of a particular task that you might need to accomplish with Kubernetes. We expect people to dive into the book to learn about a specific topic or interest, and then leave the book alone, only to return when a new topic comes up.

Despite this standalone approach, there are some themes that span the book. There are several chapters on developing applications on Kubernetes. Chapter 2 covers developer workflows. Chapter 5 discusses Continuous Integration and testing. Chapter 15 covers building higher-level platforms on top of Kubernetes, and Chapter 16 discusses managing state and stateful applications. In addition to developing applications, there are several chapters on operating services in Kubernetes. Chapter 1 covers the setup of a basic service, and Chapter 3 covers monitoring and metrics. Chapter 4 covers configuration management, while Chapter 6 covers versioning and releases. Chapter 7 covers deploying your application around the world.

There are also several chapters on cluster management, including Chapter 8 on resource management, Chapter 9 on networking, Chapter 10 on pod security, Chapter 11 on policy and governance, Chapter 12 on managing multiple clusters, and Chapter 17 on admission control and authorization. Finally there are several chapters that are truly independent; these cover machine learning (Chapter 14) and integrating with external services (Chapter 13).

Though it can be useful to read all of the chapters before you actually attempt the topic in the real world, our primary hope is that you will treat this book as a reference. It is intended as a guide as you put these topics to practice in the real world.

## Conventions Used in This Book

The following typographical conventions are used in this book:

*Italic*
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

**`Constant width bold`**
: Shows commands or other text that should be typed literally by the user.

*`Constant width italic`*
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

## Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://oreil.ly/KBPsample*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Kubernetes Best Practices* by Brendan Burns, Eddie Villalba, Dave Strebel, and Lachlan Evenson (O'Reilly). Copyright 2020 Brendan Burns, Eddie Villalba, Dave Strebel, and Lachlan Evenson, 978-1-492-05647-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

## O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

## How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/KubBP*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

# Acknowledgments

Brendan would like to thank his wonderful family, Robin, Julia, and Ethan, for the love and support of everything he does; the Kubernetes community, without whom none of this would be possible; and his fabulous coauthors, without whom this book would not exist.

Dave would like to thank his beautiful wife, Jen, and their three children, Max, Maddie, and Mason, for all of their support. He would also like to thank the Kubernetes community for all the advice and help they have provided over the years. Finally, he would like to thank his coauthors in making this adventure a reality.

Lachlan would like to thank his wife and three children for their love and support. He would also like to thank everyone in the Kubernetes community, including the wonderful individuals who have taken the time to teach him over the years. He also would like to send a special thanks to Joseph Sandoval for his mentorship. And, finally, he would like to thank his fantastic coauthors for making this book possible.

Eddie would like to thank his wife, Sandra, for her moral support and for letting him disappear for hours on end to write while she was in the final trimester of their first pregnancy. He would also like to thank his new daughter, Giavanna, for giving him the drive to push forward. Finally, he would like to thank the Kubernetes community and his coauthors who have always been guideposts in his journey to be cloud native.

We would all like to thank Virginia Wilson for her work in developing the manuscript and helping us bring all of our ideas together, and Bridget Kromhout, Bilgin Ibryam, Roland Huß, and Justin Domingus for their attention to the finishing touches.

CHAPTER 1

# Setting Up a Basic Service

This chapter describes the practices for setting up a simple multitier application in Kubernetes. The application consists of a simple web application and a database. Though this might not be the most complicated application, it is a good place to start to orient to managing an application in Kubernetes.

## Application Overview

The application that we will use for our sample isn't particularly complex. It's a simple journal service that stores its data in a Redis backend. It has a separate static file server using NGINX. It presents two web paths on a single URL. The paths are one for the journal's RESTful application programming interface (API), *https://my-host.io/api*, and a file server on the main URL, *https://my-host.io*. It uses the Let's Encrypt service for managing Secure Sockets Layer (SSL) certificates. Figure 1-1 presents a diagram of the application. Throughout this chapter, we build up this application, first using YAML configuration files and then Helm charts.

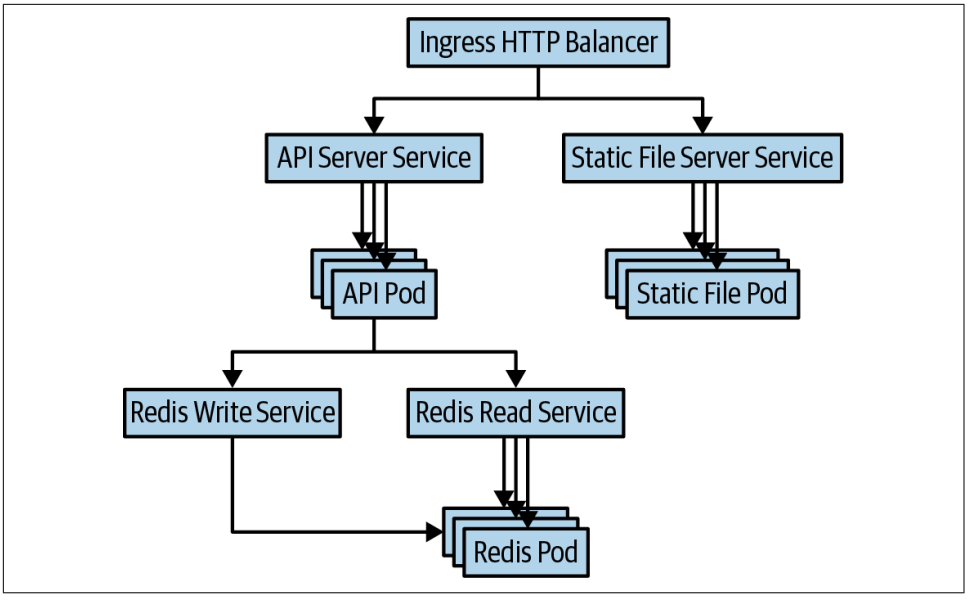



*Figure 1-1. An application diagram*

## Managing Configuration Files

Before we get into the details of how to construct this application in Kubernetes, it is worth discussing how we manage the configurations themselves. With Kubernetes, everything is represented *declaratively*. This means that you write down the desired state of the application in the cluster (generally in YAML or JSON files), and these declared desired states define all of the pieces of your application. This declarative approach is far preferable to an *imperative* approach in which the state of your cluster is the sum of a series of changes to the cluster. If a cluster is configured imperatively, it is very difficult to understand and replicate how the cluster came to be in that state. This makes it very challenging to understand or recover from problems with your application.

When declaring the state of your application, people typically prefer YAML to JSON, though Kubernetes supports them both. This is because YAML is somewhat less verbose and more human editable than JSON. However, it's worth noting that YAML is indentation sensitive; often errors in Kubernetes configurations can be traced to incorrect indentation in YAML. If things aren't behaving as expected, indentation is a good thing to check.

Because the declarative state contained in these YAML files serves as the source of truth for your application, correct management of this state is critical to the success of your application. When modifying your application's desired state, you will want to be able to manage changes, validate that they are correct, audit who made changes,

and possibly roll things back if they fail. Fortunately, in the context of software engineering, we have already developed the tools necessary to manage both changes to the declarative state as well as audit and rollback. Namely, the best practices around both version control and code review directly apply to the task of managing the declarative state of your application.

These days most people store their Kubernetes configurations in Git. Though the specific details of the version control system are unimportant, many tools in the Kubernetes ecosystem expect files in a Git repository. For code review there is much more heterogeneity, though clearly GitHub is quite popular, others use on-premises code review tools or services. Regardless of how you implement code review for your application configuration, you should treat it with the same diligence and focus that you apply to source control.

When it comes to laying out the filesystem for your application, it's generally worthwhile to use the directory organization that comes with the filesystem to organize your components. Typically, a single directory is used to encompass an *Application Service* for whatever definition of Application Service is useful for your team. Within that directory, subdirectories are used for subcomponents of the application.

For our application, we lay out the files as follows:

```
journal/
  frontend/
  redis/
  fileserver/
```

Within each directory are the concrete YAML files needed to define the service. As you'll see later on, as we begin to deploy our application to multiple different regions or clusters, this file layout will become more complicated.

## Creating a Replicated Service Using Deployments

To describe our application, we'll begin at the frontend and work downward. The frontend application for the journal is a Node.js application implemented in TypeScript. The complete application is slightly too large to include in the book. The application exposes an HTTP service on port 8080 that serves requests to the */api/** path and uses the Redis backend to add, delete, or return the current journal entries. This application can be built into a container image using the included Dockerfile and pushed to your own image repository. Then, substitute this image name in the YAML examples that follow.

## Best Practices for Image Management

Though in general, building and maintaining container images is beyond the scope of this book, it's worthwhile to identify some general best practices for building and naming images. In general, the image build process can be vulnerable to "supply-chain attacks." In such attacks, a malicious user injects code or binaries into some dependency from a trusted source that is then built into your application. Because of the risk of such attacks, it is critical that when you build your images you base them on only well-known and trusted image providers. Alternately, you can build all your images from scratch. Building from scratch is easy for some languages (e.g., Go) that can build static binaries, but it is significantly more complicated for interpreted languages like Python, JavaScript, or Ruby.

The other best practices for images relate to naming. Though the version of a container image in an image registry is theoretically mutable, you should treat the version tag as immutable. In particular, some combination of the semantic version and the SHA hash of the commit where the image was built is a good practice for naming images (e.g., *v1.0.1-bfeda01f*). If you don't specify an image version, `latest` is used by default. Although this can be convenient in development, it is a bad idea for production usage because `latest` is clearly being mutated every time a new image is built.

## Creating a Replicated Application

Our frontend application is *stateless*; it relies entirely on the Redis backend for its state. As a result, we can replicate it arbitrarily without affecting traffic. Though our application is unlikely to sustain large-scale usage, it's still a good idea to run with at least two replicas so that you can handle an unexpected crash or roll out a new version of the application without downtime.

Though in Kubernetes, a ReplicaSet is the resource that manages replicating a containerized application, so it is not a best practice to use it directly. Instead, you use the Deployment resource. A Deployment combines the replication capabilities of ReplicaSet with versioning and the ability to perform a staged rollout. By using a Deployment you can use Kubernetes' built-in tooling to move from one version of the application to the next.

The Kubernetes Deployment resource for our application looks as follows:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: frontend
  name: frontend
  namespace: default
spec:
```

```
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - image: my-repo/journal-server:v1-abcde
        imagePullPolicy: IfNotPresent
        name: frontend
        resources:
          request:
            cpu: "1.0"
            memory: "1G"
          limits:
            cpu: "1.0"
            memory: "1G"
```

There are several things to note in this Deployment. First is that we are using Labels to identify the Deployment as well as the ReplicaSets and the pods that the Deployment creates. We've added the `app: frontend` label to all of these resources so that we can examine all resources for a particular layer in a single request. You'll see that as we add other resources, we'll follow the same practice.

Additionally, we've added comments in a number of places in the YAML. Although these comments don't make it into the Kubernetes resource stored on the server, just like comments in code, they serve to help guide people who are looking at this configuration for the first time.

You should also note that for the containers in the Deployment we have specified both Request and Limit resource requests, and we've set Request equal to Limit. When running an application, the Request is the reservation that is guaranteed on the host machine where it runs. The Limit is the maximum resource usage that the container will be allowed. When you are starting out, setting Request equal to Limit will lead to the most predictable behavior of your application. This predictability comes at the expense of resource utilization. Because setting Request equal to Limit prevents your applications from overscheduling or consuming excess idle resources, you will not be able to drive maximal utilization unless you tune Request and Limit very, very carefully. As you become more advanced in your understanding of the Kubernetes resource model, you might consider modifying Request and Limit for your application independently, but in general most users find that the stability from predictability is worth the reduced utilization.

Now that we have the Deployment resource defined, we'll check it into version control, and deploy it to Kubernetes:

```
git add frontend/deployment.yaml
git commit -m "Added deployment" frontend/deployment.yaml
kubectl apply -f frontend/deployment.yaml
```

It is also a best practice to ensure that the contents of your cluster exactly match the contents of your source control. The best pattern to ensure this is to adopt a GitOps approach and deploy to production only from a specific branch of your source control, using Continuous Integration (CI)/Continuous Delivery (CD) automation. In this way you're guaranteed that source control and production match. Though a full CI/CD pipeline might seem excessive for a simple application, the automation by itself, independent of the reliability it provides, is usually worth the time taken to set it up. And CI/CD is extremely difficult to retrofit into an existing, imperatively deployed application.

There are also some pieces of this application description YAML (e.g., the ConfigMap and secret volumes) as well as pod Quality of Service that we examine in later sections.

## Setting Up an External Ingress for HTTP Traffic

The containers for our application are now deployed, but it's not currently possible for anyone to access the application. By default, cluster resources are available only within the cluster itself. To expose our application to the world, we need to create a Service and load balancer to provide an external IP address and to bring traffic to our containers. For the external exposure we are actually going to use two Kubernetes resources. The first is a Service that load-balances Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) traffic. In our case, we're using the TCP protocol. And the second is an Ingress resource, which provides HTTP(S) load balancing with intelligent routing of requests based on HTTP paths and hosts. With a simple application like this, you might wonder why we choose to use the more complex Ingress, but as you'll see in later sections, even this simple application will be serving HTTP requests from two different services. Furthermore, having an Ingress at the edge enables flexibility for future expansion of our service.

Before the Ingress resource can be defined, there needs to be a Kubernetes Service for the Ingress to point to. We'll use Labels to direct the Service to the pods that we created in the previous section. The Service is significantly simpler to define than the Deployment and looks as follows:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: frontend
  name: frontend
  namespace: default
```

```
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    app: frontend
  type: ClusterIP
```

After you've defined the Service, you can define an Ingress resource. Unlike Service resources, Ingress requires an Ingress controller container to be running in the cluster. There are a number of different implementations you can choose from, either provided by your cloud provider, or implemented using open source servers. If you choose to install an open source ingress provider, it's a good idea to use the Helm package manager to install and maintain it. The `nginx` or `haproxy` Ingress providers are popular choices:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: frontend-ingress
spec:
  rules:
  - http:
      paths:
      - path: /api
        backend:
          serviceName: frontend
          servicePort: 8080
```

# Configuring an Application with ConfigMaps

Every application needs a degree of configuration. This could be the number of journal entries to display per page, the color of a particular background, a special holiday display, or many other types of configuration. Typically, separating such configuration information from the application itself is a best practice to follow.

There are a couple of different reasons for this separation. The first is that you might want to configure the same application binary with different configurations depending on the setting. In Europe you might want to light up an Easter special, whereas in China you might want to display a special for Chinese New Year. In addition to this environmental specialization, there are agility reasons for the separation. Usually a binary release contains multiple different new features; if you turn on these features via code, the only way to modify the active features is to build and release a new binary, which can be an expensive and slow process.

The use of configuration to activate a set of features means that you can quickly (and even dynamically) activate and deactivate features in response to user needs or

application code failures. Features can be rolled out and rolled back on a per-feature basis. This flexibility ensures that you are continually making forward progress with most features even if some need to be rolled back to address performance or correctness problems.

In Kubernetes this sort of configuration is represented by a resource called a ConfigMap. A ConfigMap contains multiple key/value pairs representing configuration information or a file. This configuration information can be presented to a container in a pod via either files or environment variables. Imagine that you want to configure your online journal application to display a configurable number of journal entries per page. To achieve this, you can define a ConfigMap as follows:

```
kubectl create configmap frontend-config --from-literal=journalEntries=10
```

To configure your application, you expose the configuration information as an environment variable in the application itself. To do that, you can add the following to the `container` resource in the Deployment that you defined earlier:

```
...
# The containers array in the PodTemplate inside the Deployment
containers:
  - name: frontend
    ...
    env:
    - name: JOURNAL_ENTRIES
      valueFrom:
        configMapKeyRef:
          name: frontend-config
          key: journalEntries
...
```

Although this demonstrates how you can use a ConfigMap to configure your application, in the real world of Deployments, you'll want to roll out regular changes to this configuration with weekly rollouts or even more frequently. It might be tempting to roll this out by simply changing the ConfigMap itself, but this isn't really a best practice. There are several reasons for this: the first is that changing the configuration doesn't actually trigger an update to existing pods. Only when the pod is restarted is the configuration applied. Because of this, the rollout isn't health based and can be ad hoc or random.

A better approach is to put a version number in the name of the ConfigMap itself. Instead of calling it `frontend-config`, call it `frontend-config-v1`. When you want to make a change, instead of updating the ConfigMap in place, you create a new `v2` ConfigMap, and then update the Deployment resource to use that configuration. When you do this, a Deployment rollout is automatically triggered, using the appropriate health checking and pauses between changes. Furthermore, if you ever need to rollback, the `v1` configuration is sitting in the cluster and rollback is as simple as updating the Deployment again.

## Managing Authentication with Secrets

So far, we haven't really discussed the Redis service to which our frontend is connecting. But in any real application we need to secure connections between our services. In part this is to ensure the security of users and their data, and in addition, it is essential to prevent mistakes like connecting a development frontend with a production database.

The Redis database is authenticated using a simple password. It might be convenient to think that you would store this password in the source code of your application, or in a file in your image, but these are both bad ideas for a variety of reasons. The first is that you have leaked your secret (the password) into an environment where you aren't necessarily thinking about access control. If you put a password into your source control, you are aligning access to your source with access to all secrets. This is probably not correct. You probably will have a broader set of users who can access your source code than should really have access to your Redis instance. Likewise, someone who has access to your container image shouldn't necessarily have access to your production database.

In addition to concerns about access control, another reason to avoid binding secrets to source control and/or images is parameterization. You want to be able to use the same source code and images in a variety of environments (e.g., development, canary, and production). If the secrets are tightly bound in source code or image, you need a different image (or different code) for each environment.

Having seen ConfigMaps in the previous section, you might immediately think that the password could be stored as a configuration and then populated into the application as an application-specific configuration. You're absolutely correct to believe that the separation of configuration from application is the same as the separation of secrets from application. But the truth is that a secret is an important concept by itself. You likely want to handle access control, handling, and updates of secrets in a different way than a configuration. More important, you want your developers *thinking* differently when they are accessing secrets than when they are accessing configuration. For these reasons, Kubernetes has a built-in Secret resource for managing secret data.

You can create a secret password for your Redis database as follows:

```
kubectl create secret generic redis-passwd --from-literal=passwd=${RANDOM}
```

Obviously, you might want to use something other than a random number for your password. Additionally, you likely want to use a secret/key management service, either via your cloud provider, like Microsoft Azure Key Vault, or an open source project, like HashiCorp's Vault. When you are using a key management service, they generally have tighter integration with Kubernetes secrets.

Secrets in Kubernetes are stored unencrypted by default. If you want to store secrets encrypted, you can integrate with a key provider to give you a key that Kubernetes will use to encrypt all of the secrets in the cluster. Note that although this secures the keys against direct attacks to the `etcd` database, you still need to ensure that access via the Kubernetes API server is properly secured.

After you have stored the Redis password as a secret in Kubernetes, you then need to *bind* that secret to the running application when deployed to Kubernetes. To do this, you can use a Kubernetes Volume. A Volume is effectively a file or directory that can be mounted into a running container at a user-specified location. In the case of secrets, the Volume is created as a tmpfs RAM-backed filesystem and then mounted into the container. This ensures that even if the machine is physically compromised (quite unlikely in the cloud, but possible in the datacenter), the secrets are much more difficult to obtain by the attacker.

To add a secret volume to a Deployment, you need to specify two new entries in the YAML for the Deployment. The first is a `volume` entry for the pod that adds the volume to the pod:

```
...
  volumes:
  - name: passwd-volume
    secret:
    secretName: redis-passwd
```

With the volume in the pod, you need to mount it into a specific container. You do this via the `volumeMounts` field in the container description:

```
...
  volumeMounts:
  - name: passwd-volume
    readOnly: true
    mountPath: "/etc/redis-passwd"
...
```

This mounts the secret volume into the `redis-passwd` directory for access from the client code. Putting this all together, you have the complete Deployment as follows:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: frontend
  name: frontend
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
```

```
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - image: my-repo/journal-server:v1-abcde
        imagePullPolicy: IfNotPresent
        name: frontend
        volumeMounts:
        - name: passwd-volume
          readOnly: true
          mountPath: "/etc/redis-passwd"
        resources:
          requests:
            cpu: "1.0"
            memory: "1G"
          limits:
            cpu: "1.0"
            memory: "1G"
      volumes:
        - name: passwd-volume
          secret:
            secretName: redis-passwd
```

At this point we have configured the client application to have a secret available to authenticate to the Redis service. Configuring Redis to use this password is similar; we mount it into the Redis pod and load the password from the file.

# Deploying a Simple Stateful Database

Although conceptually deploying a stateful application is similar to deploying a client like our frontend, state brings with it more complications. The first is that in Kubernetes a pod can be rescheduled for a number of reasons, such as node health, an upgrade, or rebalancing. When this happens, the pod might move to a different machine. If the data associated with the Redis instance is located on any particular machine or within the container itself, that data will be lost when the container migrates or restarts. To prevent this, when running stateful workloads in Kubernetes its important to use remote *PersistentVolume*s to manage the state associated with the application.

There is a wide variety of different implementations of PersistentVolumes in Kubernetes, but they all share common characteristics. Like secret volumes described earlier, they are associated with a pod and mounted into a container at a particular location. Unlike secrets, PersistentVolumes are generally remote storage mounted through some sort of network protocol, either file based, such as Network File System (NFS) or Server Message Block (SMB), or block based (iSCSI, cloud-based disks,

etc.). Generally, for applications such as databases, block-based disks are preferable because they generally offer better performance, but if performance is less of a consideration, file-based disks can sometimes offer greater flexibility.

Managing state in general is complicated, and Kubernetes is no exception. If you are running in an environment that supports stateful services (e.g., MySQL as a service, Redis as a service), it is generally a good idea to use those stateful services. Initially, the cost premium of a stateful Software as a Service (SaaS) might seem expensive, but when you factor in all the operational requirements of state (backup, data locality, redundancy, etc.), and the fact that the presence of state in a Kubernetes cluster makes it difficult to move applications between clusters, it becomes clear that, in most cases, storage SaaS is worth the price premium. In on-premises environments where storage SaaS isn't available, having a dedicated team provide storage as a service to the entire organization is definitely a better practice than allowing each team to roll its own.

To deploy our Redis service, we use a StatefulSet resource. Added after the initial Kubernetes release as a complement to ReplicaSet resources, a StatefulSet gives slightly stronger guarantees such as consistent names (no random hashes!) and a defined order for scale-up and scale-down. When you are deploying a singleton, this is somewhat less important, but when you want to deploy replicated state, these attributes are very convenient.

To obtain a PersistentVolume for our Redis, we use a PersistentVolumeClaim. You can think of a claim as a "request for resources." Our Redis declares abstractly that it wants 50 GB of storage, and the Kubernetes cluster determines how to provision an appropriate PersistentVolume. There are two reasons for this. The first is so that we can write a StatefulSet that is portable between different clouds and on-premises, where the details of disks might be different. The other reason is that although many PersistentVolume types can be mounted to only a single pod, we can use volume claims to write a template that can be replicated and yet have each pod assigned its own specific PersistentVolume.

The following example shows a Redis StatefulSet with PersistentVolumes:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  serviceName: "redis"
  replicas: 1
  selector:
    matchLabels:
```

```
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:5-alpine
        ports:
        - containerPort: 6379
          name: redis
        volumeMounts:
        - name: data
          mountPath: /data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 10Gi
```

This deploys a single instance of your Redis service, but suppose you want to replicate the Redis cluster for scale-out of reads and resiliency to failures. To do this you need to obviously increase the number of replicas to three, but you also need to ensure that the two new replicas connect to the write master for Redis.

When you create the headless Service for the Redis StatefulSet, it creates a DNS entry `redis-0.redis`; this is the IP address of the first replica. You can use this to create a simple script that can launch in all of the containters:

```
#!/bin/sh

PASSWORD=$(cat /etc/redis-passwd/passwd)

if [[ "${HOSTNAME}" == "redis-0" ]]; then
  redis-server --requirepass ${PASSWORD}
else
  redis-server --slaveof redis-0.redis 6379 --masterauth ${PASSWORD} --
requirepass ${PASSWORD}
fi
```

You can create this script as a ConfigMap:

```
kubectl create configmap redis-config --from-file=./launch.sh
```

You then add this ConfigMap to your StatefulSet and use it as the command for the container. Let's also add in the password for authentication that we created earlier in the chapter.

The complete three-replica Redis looks as follows:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  serviceName: "redis"
  replicas: 3
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:5-alpine
        ports:
        - containerPort: 6379
          name: redis
        volumeMounts:
        - name: data
          mountPath: /data
        - name: script
          mountPath: /script/launch.sh
          subPath: launch.sh
        - name: passwd-volume
          mountPath: /etc/redis-passwd
        command:
        - sh
        - -c
        - /script/launch.sh
      volumes:
      - name: script
        configMap:
          name: redis-config
          defaultMode: 0777
      - name: passwd-volume
        secret:
          secretName: redis-passwd
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 10Gi
```

## Creating a TCP Load Balancer by Using Services

Now that we've deployed the stateful Redis service, we need to make it available to our frontend. To do this, we create two different Kubernetes Services. The first is the Service for reading data from Redis. Because Redis is replicating the data to all three members of the StatefulSet, we don't care which read our request goes to. Consequently, we use a basic Service for the reads:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: redis
  name: redis
  namespace: default
spec:
  ports:
  - port: 6379
    protocol: TCP
    targetPort: 6379
  selector:
    app: redis
  sessionAffinity: None
  type: ClusterIP
```

To enable writes, you need to target the Redis master (replica #0). To do this, create a *headless* Service. A headless Service doesn't have a cluster IP address; instead, it programs a DNS entry for every pod in the StatefulSet. This means that we can access our master via the `redis-0.redis` DNS name:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: redis-write
  name: redis-write
spec:
  clusterIP: None
  ports:
  - port: 6379
  selector:
    app: redis
```

Thus, when we want to connect to Redis for writes or transactional read/write pairs, we can build a separate write client connected to the `redis-0.redis-write` server.

## Using Ingress to Route Traffic to a Static File Server

The final component in our application is a *static file server*. The static file server is responsible for serving HTML, CSS, JavaScript, and image files. It's both more efficient and more focused for us to separate static file serving from our API serving frontend described earlier. We can easily use a high-performance static off-the-shelf file server like NGINX to serve files while we allow our development teams to focus on the code needed to implement our API.

Fortunately, the Ingress resource makes this source of mini-microservice architecture very easy. Just like the frontend, we can use a Deployment resource to describe a replicated NGINX server. Let's build the static images into the NGINX container and deploy them to each replica. The Deployment resource looks as follows:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: fileserver
  name: fileserver
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: fileserver
  template:
    metadata:
      labels:
        app: fileserver
    spec:
      containers:
      # This image is intended as an example, replace it with your own
      # static files image.
      - image: my-repo/static-files:v1-abcde
        imagePullPolicy: Always
        name: fileserver
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        resources:
          request:
            cpu: "1.0"
            memory: "1G"
          limits:
            cpu: "1.0"
            memory: "1G"
      dnsPolicy: ClusterFirst
      restartPolicy: Always
```

Now that there is a replicated static web server up and running, you will likewise create a Service resource to act as a load balancer:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: fileserver
  name: fileserver
  namespace: default
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: fileserver
  sessionAffinity: None
  type: ClusterIP
```

Now that you have a Service for your static file server, extend the Ingress resource to contain the new path. It's important to note that you must place the `/` path *after* the `/api` path, or else it would subsume `/api` and direct API requests to the static file server. The new Ingress looks like this:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: frontend-ingress
spec:
  rules:
  - http:
      paths:
      - path: /api
        backend:
          serviceName: fileserver
          servicePort: 8080
      # NOTE: this should come after /api or else it will hijack requests
      - path: /
        backend:
          serviceName: fileserver
          servicePort: 80
```

# Parameterizing Your Application by Using Helm

Everything that we have discussed so far focuses on deploying a single instance of our service to a single cluster. However, in reality, nearly every service and every service team is going to need to deploy to multiple different environments (even if they share a cluster). Even if you are a single developer working on a single application, you likely want to have at least a development version and a production version of your

application so that you can iterate and develop without breaking production users. After you factor in integration testing and CI/CD, it's likely that even with a single service and a handful of developers, you'll want to deploy to at least three different environments, and possibly more if you consider handling datacenter-level failures.

An initial failure mode for many teams is to simply copy the files from one cluster to another. Instead of having a single *frontend/* directory, have a *frontend-production/* and *frontend-development/* pair of directories. The reason this is so dangerous is because you are now in charge of ensuring that these files remain synchronized with one another. If they were intended to be entirely identical, this might be easy, but some skew between development and production is expected because you will be developing new features; it's critical that the skew is both intentional, and easily managed.

Another option to achieve this would be to use branches and version control, with the production and development branches leading off from a central repository, and the differences between the branches clearly visible. This can be a viable option for some teams, but the mechanics of moving between branches are challenging when you want to simultaneously deploy software to different environments (e.g., a CI/CD system that deploys to a number of different cloud regions).

Consequently, most people end up with a *templating system*. A templating system combines templates, which form the centralized backbone of the application configuration, with parameters that *specialize* the template to a specific environment configuration. In this way, you can have a generally shared configuration, with intentional (and easily understood) customization as needed. There are a variety of different template systems for Kubernetes, but the most popular by far is a system called Helm.

In Helm, an application is packaged in a collection of files called a *chart* (nautical jokes abound in the world of containers and Kubernetes).

A chart begins with a *chart.yaml* file, which defines the metadata for the chart itself:

```
apiVersion: v1
appVersion: "1.0"
description: A Helm chart for our frontend journal server.
name: frontend
version: 0.1.0
```

This file is placed in the root of the chart directory (e.g., *frontend/*). Within this directory, there is a *templates* directory, which is where the templates are placed. A template is basically a YAML file from the previous examples, with some of the values in the file replaced with parameter references. For example, imagine that you want to parameterize the number of replicas in your frontend. Previously, here's what the Deployment had:

```
...
spec:
  replicas: 2
...
```

In the template file (*frontend-deployment.tmpl*), it instead looks like the following:

```
...
spec:
  replicas: {{ .replicaCount }}
...
```

This means that when you deploy the chart, you'll substitute the value for replicas with the appropriate parameter. The parameters themselves are defined in a *values.yaml* file. There will be one values file per environment where the application should be deployed. The values file for this simple chart would look like this:

```
replicaCount: 2
```

Putting this all together, you can deploy this chart using the `helm` tool, as follows:

```
helm install path/to/chart --values path/to/environment/values.yaml
```

This parameterizes your application and deploys it to Kubernetes. Over time these parameterizations will grow to encompass the variety of different environments for your application.

# Deploying Services Best Practices

Kubernetes is a powerful system that can seem complex. But setting up a basic application for success can be straightforward if you use the following best practices:

- Most services should be deployed as Deployment resources. Deployments create identical replicas for redundancy and scale.
- Deployments can be exposed using a Service, which is effectively a load balancer. A Service can be exposed either within a cluster (the default) or externally. If you want to expose an HTTP application, you can use an Ingress controller to add things like request routing and SSL.
- Eventually you will want to parameterize your application to make its configuration more reusable in different environments. Packaging tools like Helm are the best choice for this kind of parameterization.

# Summary

The application built in this chapter is a simple one, but it contains nearly all of the concepts you'll need to build larger, more complicated applications. Understanding

how the pieces fit together and how to use foundational Kubernetes components is key to successfully working with Kubernetes.

Laying the correct foundation via version control, code review, and continuous delivery of your service ensures that no matter what you build, it is built in a solid manner. As we go through the more advanced topics in subsequent chapters, keep this foundational information in mind.

CHAPTER 2

# Developer Workflows

Kubernetes was built for reliably operating software. It simplifies deploying and managing applications with an application-oriented API, self-healing properties, and useful tools like Deployments for zero downtime rollout of software. Although all of these tools are useful, they don't do much to make it easier to develop applications for Kubernetes. Furthermore, even though many clusters are designed to run production applications and thus are rarely accessed by developer workflows, it is also critical to enable development workflows to target Kubernetes, and this typically means having a cluster or at least part of a cluster that is intended for development. Setting up such a cluster to facilitate easy development of applications for Kubernetes is a critical part of ensuring success with Kubernetes. Clearly if there is no code being built for your cluster, the cluster itself isn't accomplishing much.

## Goals

Before we describe the best practices for building out development clusters, it is worth stating our goals for such clusters. Obviously, the ultimate goal is to enable developers to rapidly and easily build applications on Kubernetes, but what does that really mean in practice and how is that reflected in practical features of the development cluster?

It is useful to identify phases of developer interaction with the cluster.

The first phase is *onboarding*. This is when a new developer joins the team. This phase includes giving the user a login to the cluster as well as getting them oriented to their first deployment. The goal for this phase is to get a developer's feet wet in a minimal amount of time. You should set a key performance indicator (KPI) goal for this process. A reasonable goal would be that a user could go from nothing to the current

application at HEAD running in less than half an hour. Every time someone is new to the team, test how you are doing against this goal.

The second phase is *developing*. This is the day-to-day activity of the developer. The goal for this phase is to ensure rapid iteration and debugging. Developers need to quickly and repeatedly push code to the cluster. They also need to be able to easily test their code and debug it when it isn't operating properly. The KPI for this phase is more challenging to measure, but you can estimate it by measuring the time to get a pull request (PR) or change up and running in the cluster, or with surveys of the user's perceived productivity, or both. You will also be able to measure this in the overall productivity of your teams.

The third phase is *testing*. This phase is interleaved with developing and is used to validate the code before submission and merging. The goals for this phase are two-fold. First, the developer should be able to run all tests for their environment before a PR is submitted. Second, all tests should automatically run before code is merged into the repository. In addition to these goals you should also set a KPI for the length of time the tests take to run. As your project becomes more complex, it's natural for more and more tests to take a longer time. As this happens, it might become valuable to identify a smaller set of smoke tests that a developer can use for initial validation before submitting a PR. You should also have a very strict KPI around *test flakiness*. A flaky test is one that occasionally (or not so occasionally) fails. In any reasonably active project, a flakiness rate of more than one failure per one thousand runs will lead to developer friction. You need to ensure that your cluster environment does not lead to flaky tests. Whereas sometimes flaky tests occur due to problems in the code, they can also occur because of interference in the development environment (e.g., running out of resources and noisy neighbors). You should ensure that your development environment is free of such issues by measuring test flakiness and acting quickly to fix it.

## Building a Development Cluster

When people begin to think about developing on Kubernetes, one of the first choices that occurs is whether to build a single large development cluster or to have one cluster per developer. Note that this choice only makes sense in an environment in which dynamic cluster creation is easy, such as the public cloud. In physical environments, its possible that one large cluster is the only choice.

If you do have a choice you should consider the pros and cons of each option. If you choose to have a development cluster per user, the significant downside of this approach is that it will be more expensive and less efficient, and you will have a large number of different development clusters to manage. The extra costs come from the fact that each cluster is likely to be heavily underutilized. Also, with developers creating different clusters, it becomes more difficult to track and garbage-collect resources

that are no longer in use. The advantage of the cluster-per-user approach is simplicity: each developer can self-service manage their own cluster, and from isolation, it's much more difficult for different developers to step on one another's toes.

On the other hand, a single development cluster will be significantly more efficient; you can likely sustain the same number of developers on a shared cluster for one-third the price (or less). Plus, it's much easier for you to install shared cluster services, for example, monitoring and logging, which makes it significantly easier to produce a developer-friendly cluster. The downside of a shared development cluster is the process of user management and potential interference between developers. Because the process of adding new users and namespaces to the Kubernetes cluster isn't currently streamlined, you will need to activate a process to onboard new developers. Although Kubernetes resource management and Role-Based Access Control (RBAC) can reduce the probability that two developers conflict, it is always possible that a user will *brick* the development cluster by consuming too many resources so that other applications and developers won't schedule. Additionally, you will still need to ensure that developers don't leak and forget about resources they've created. This is somewhat easier, though, than the approach in which developers each create their own clusters.

Even though both approaches are feasible, generally, our recommendation is to have a single large cluster for all developers. Although there are challenges in interference between developers, they can be managed and ultimately the cost efficiency and ability to easily add organization-wide capabilities to the cluster outweigh the risks of interference. But you will need to invest in a process for onboarding developers, resource management, and garbage collection. Our recommendation would be to try a single large cluster as a first option. As your organization grows (or if it is already large), you might consider having a cluster per team or group (10 to 20 people) rather than a giant cluster for hundreds of users. This can make both billing and management easier.

## Setting Up a Shared Cluster for Multiple Developers

When setting up a large cluster, the primary goal is to ensure that multiple users can simultaneously use the cluster without stepping on one another's toes. The obvious way to separate your different developers is with Kubernetes namespaces. Namespaces can serve as scopes for the deployment of services so that one user's frontend service doesn't interfere with another user's frontend service. Namespaces are also scopes for RBAC, ensuring that one developer cannot accidentally delete another developer's work. Thus, in a shared cluster it makes sense to use a namespace as a developer's workspace. The processes for onboarding users and creating and securing a namespace are described in the following sections.

## Onboarding Users

Before you can assign a user to a namespace, you have to onboard that user to the Kubernetes cluster itself. To achieve this, there are two options. You can use certificate-based authentication to create a new certificate for the user and give them a *kubeconfig* file that they can use to log in, or you can configure your cluster to use an external identity system (for example, Microsoft Azure Active Directory or AWS Identity and Access Management [IAM]) for cluster access.

In general, using an external identity system is a best practice because it doesn't require that you maintain two different sources of identity, but in some cases this isn't possible and you need to use certificates. Fortunately, you can use the Kubernetes certificate API for creating and managing such certificates. Here's the process for adding a new user to an existing cluster.

First, you need to generate a certificate signing request to generate a new certificate. Here is a simple Go program to do this:

```
package main

import (
        "crypto/rand"
        "crypto/rsa"
        "crypto/x509"
        "crypto/x509/pkix"
        "encoding/asn1"
        "encoding/pem"
        "os"
)

func main() {
        name := os.Args[1]
        user := os.Args[2]

        key, err := rsa.GenerateKey(rand.Reader, 1024)
        if err != nil {
                panic(err)
        }
        keyDer := x509.MarshalPKCS1PrivateKey(key)
        keyBlock := pem.Block{
                Type:  "RSA PRIVATE KEY",
                Bytes: keyDer,
        }
        keyFile, err := os.Create(name + "-key.pem")
        if err != nil {
                panic(err)
        }
        pem.Encode(keyFile, &keyBlock)
        keyFile.Close()
```

```
        commonName := user
        // You may want to update these too
        emailAddress := "someone@myco.com"

        org := "My Co, Inc."
        orgUnit := "Widget Farmers"
        city := "Seattle"
        state := "WA"
        country := "US"

        subject := pkix.Name{
                CommonName:         commonName,
                Country:            []string{country},
                Locality:           []string{city},
                Organization:       []string{org},
                OrganizationalUnit: []string{orgUnit},
                Province:           []string{state},
        }

        asn1, err := asn1.Marshal(subject.ToRDNSequence())
        if err != nil {
                panic(err)
        }
        csr := x509.CertificateRequest{
                RawSubject:         asn1,
                EmailAddresses:     []string{emailAddress},
                SignatureAlgorithm: x509.SHA256WithRSA,
        }

        bytes, err := x509.CreateCertificateRequest(rand.Reader, &csr, key)
        if err != nil {
                panic(err)
        }
        csrFile, err := os.Create(name + ".csr")
        if err != nil {
                panic(err)
        }

        pem.Encode(csrFile, &pem.Block{Type: "CERTIFICATE REQUEST", Bytes:
bytes})
        csrFile.Close()
}
```

You can run this as follows:

```
go run csr-gen.go client <user-name>;
```

This creates files called *client-key.pem* and *client.csr*. You then can run the following script to create and download a new certificate:

```
#!/bin/bash

csr_name="my-client-csr"
```

```
name="${1:-my-user}"

csr="${2}"


cat <<EOF | kubectl create -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: ${csr_name}
spec:
  groups:
  - system:authenticated
  request: $(cat ${csr} | base64 | tr -d '\n')
  usages:
  - digital signature
  - key encipherment
  - client auth
EOF

echo
echo "Approving signing request."
kubectl certificate approve ${csr_name}

echo
echo "Downloading certificate."
kubectl get csr ${csr_name} -o jsonpath='{.status.certificate}' \
        | base64 --decode > $(basename ${csr} .csr).crt

echo
echo "Cleaning up"
kubectl delete csr ${csr_name}

echo
echo "Add the following to the 'users' list in your kubeconfig file:"
echo "- name: ${name}"
echo "  user:"
echo "    client-certificate: ${PWD}/$(basename ${csr} .csr).crt"
echo "    client-key: ${PWD}/$(basename ${csr} .csr)-key.pem"
echo
echo "Next you may want to add a role-binding for this user."
```

This script prints out the final information that you can add to a *kubeconfig* file to enable that user. Of course, the user has no access privileges, so you will need to apply Kubernetes RBAC for the user in order to grant them privileges to a namespace.

## Creating and Securing a Namespace

The first step in provisioning a namespace is actually just creating it. You can do this using **`kubectl create namespace my-namespace`**.

But the truth is that when you create a namespace, you want to attach a bunch of metadata to that namespace, for example, the contact information for the team that builds the component deployed into the namespace. Generally, this is in the form of annotations; you can either generate the YAML file using some templating, such as Jinja or others, or you can create and then annotate the namespace. A simple script to do this looks like:

```
ns='my-namespace'
kubectl create namespace ${ns}
kubectl annotate namespace ${ns} annotation_key=annotation_value
```

When the namespace is created, you want to secure it by ensuring that you can grant access to the namespace to a specific user. To do this, you can bind a role to a user in the context of that namespace. You do this by creating a `RoleBinding` object within the namespace itself. The `RoleBinding` might look like this:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: example
  namespace: my-namespace
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: edit
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: myuser
```

To create it, you simply run `kubectl create -f role-binding.yaml`. Note that you can reuse this binding as much as you want so long as you update the namespace in the binding to point to the correct namespace. If you ensure that the user doesn't have any other role bindings, you can be assured that this namespace is the only part of the cluster to which the user has access. A reasonable practice is to also grant reader access to the entire cluster; in this way developers can see what others are doing in case it is interfering with their work. Be careful in granting such read access, however, because it will include access to secret resources in the cluster. Generally, in a development cluster this is OK because everyone is in the same organization and the secrets are used only for development; however, if this is a concern, then you can create a more fine-grained role that eliminates the ability to read secrets.

If you want to limit the amount of resources consumed by a particular namespace, you can use the ResourceQuota resource to set a limit to the total number of resources that any particular namespace consumes. For example, the following quota limits the namespace to 10 cores and 100 GB of memory for both Request and Limit for the pods in the namespace:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: limit-compute
  namespace: my-namespace
spec:
  hard:
    requests.cpu: "10"
    requests.memory: 100Gi
    limits.cpu: "10"
    limits.memory: 100Gi
```

## Managing Namespaces

Now that you have seen how to onboard a new user and how to create a namespace to use as a workspace, the question remains how to assign a developer to the namespace. As with many things, there is no single perfect answer; rather, there are two approaches. The first is to give each user their own namespace as part of the onboarding process. This is useful because after a user is onboarded, they always have a dedicated workspace in which they can develop and manage their applications. However, making the developer's namespace too persistent encourages the developer to leave things lying around in the namespace after they are done with them, and garbage-collecting and accounting individual resources is more complicated. An alternate approach is to temporarily create and assign a namespace with a bounded time to live (TTL). This ensures that the developer thinks of the resources in the cluster as transient and that it is easy to build automation around the deletion of entire namespaces when their TTL has expired.

In this model, when the developer wants to begin a new project, they use a tool to allocate a new namespace for the project. When they create the namespace, it has a selection of metadata associated with the namespace for management and accounting. Obviously, this metadata includes the TTL for the namespace, but it also includes the developer to which it is assigned, the resources that should be allocated to the namespace (e.g., CPU and memory), and the team and purpose of the namespace. This metadata ensures that you can both track resource usage and delete the namespace at the right time.

Developing the tooling to allocate namespaces on demand can seem like a challenge, but simple tooling is relatively simple to develop. For example, you can achieve the

allocation of a new namespace with a simple script that creates the namespace and prompts for the relevant metadata to attach to the namespace.

If you want to get more integrated with Kubernetes, you can use custom resource definitions (CRDs) to enable users to dynamically create and allocate new namespaces using the `kubectl` tool. If you have the time and inclination, this is definitely a good practice because it makes namespace management declarative and also enables the use of Kubernetes RBAC.

After you have tooling to enable the allocation of namespaces, you also need to add tooling to reap namespaces when their TTL has expired. Again, you can accomplish this with a simple script that examines the namespaces and deletes those that have an expired TTL.

You can build this script into a container and use a `ScheduledJob` to run it at an interval like once per hour. Combined together, these tools can ensure that developers can easily allocate independent resources for their project as needed, but those resources will also be reaped at the proper interval to ensure that you don't have wasted resources and that old resources don't get in the way of new development.

### Cluster-Level Services

In addition to tooling to allocate and manage namespaces, there are also useful cluster-level services, and it's a good idea to enable them in your development cluster. The first is log aggregation to a central Logging as a Service (LaaS) system. One of the easiest things for a developer to do to understand the operation of their application is to write something to STDOUT. Although you can access these logs via `kubectl logs`, that log is limited in length and is not particularly searchable. If you instead automatically ship those logs to a LaaS system such as a cloud service or an Elasticsearch cluster, developers can easily search through logs for relevant information as well as aggregate logging information across multiple containers in their service.

## Enabling Developer Workflows

Now that we succesfully have a shared cluster setup and we can onboard new application developers to the cluster itself, we need to actually get them developing their application. Remember that one of the key KPIs that we are measuring is the time from onboarding to an initial application running in the cluster. It's clear that via the just-described onboarding scripts we can quickly authenticate a user to a cluster and allocate a namespace, but what about getting started with the application? Unfortunately, even though there are a few techniques that help with this process, it generally requires more convention than automation to get the initial application up and running. In the following sections, we describe one approach to achieving this; it is by no

means the only approach or the only solution. You can optionally apply the approach as is or be inspired by the ideas to arrive at your own solution.

## Initial Setup

One of the main challenges to deploying an application is the installation of all of the dependencies. In many cases, especially in modern microservice architectures, to even get started developing on one of the microservices requires the deployment of multiple dependencies, either databases or other microservices. Although the deployment of the application itself is relatively straightforward, the task of identifying and deploying all of the dependencies to build the complete application is often a frustrating case of trial and error married with incomplete or out-of-date instructions.

To address this issue, it is often valuable to introduce a convention for describing and installing dependencies. This can be seen as the equivalent of something like `npm install`, which installs all of the required JavaScript dependencies. Eventually, there is likely to be a tool similar to `npm` that provides this service for Kubernetes-based applications, but until then, the best practice is to rely on convention within your team.

One such option for a convention is the creation of a *setup.sh* script within the root directory of all project repositories. The responsibility of this script is to create all dependencies within a particular namespace to ensure that all of the application's dependencies are correctly created. For example, a setup script might look like the following:

```
kubectl create my-service/database-stateful-set-yaml
kubectl create my-service/middle-tier.yaml
kubectl create my-service/configs.yaml
```

You then could integrate this script with `npm` by adding the following to your *package.json*:

```
{
    ...
    "scripts": {
        "setup": "./setup.sh",
        ...
    }
}
```

With this setup, a new developer can simply run `npm run setup` and the cluster dependencies will be installed. Obviously, this particular integration is Node.js/npm specific. In other programming languages, it will make more sense to integrate with the language-specific tooling. For example, in Java you might integrate with a Maven *pom.xml* file instead.

## Enabling Active Development

Having set up the developer workspace with required dependencies, the next task is to enable them to iterate on their application quickly. The first prerequisite for this is the ability to build and push a container image. Let's assume that you have this already set up; if not, you can read how to do this in a number of other online resources and books.

After you have built and pushed a container image, the task is to roll it out to the cluster. Unlike traditional rollouts, in the case of developer iteration, maintaining availability is really not a concern. Thus, the easiest way to deploy new code is to simply delete the Deployment object associated with the previous Deployment and then create a new Deployment pointing to the newly built image. It is also possible to update an existing Deployment in place, but this will trigger the rollout logic in the Deployment resource. Although it is possible to configure a Deployment to roll out code quickly, doing so introduces a difference between the development environment and the production environment that can be dangerous or destabilizing. Imagine, for example, that you accidentally push the development configuration of the Deployment into production; you will suddenly and accidentally deploy new versions to production without appropriate testing and delays between phases of the rollout. Because of this risk and because there is an alternative, the best practice is to delete and recreate the Deployment.

Just like installing dependencies, it is also a good practice to make a script for performing this deployment. An example *deploy.sh* script might look like the following:

```
kubectl delete -f ./my-service/deployment.yaml
perl -pi -e 's/${old_version}/${new_version}/' ./my-service/deployment.yaml
kubectl create -f ./my-service/deployment.yaml
```

As before, you can integrate this with existing programming language tooling so that (for example) a developer can simply run `npm run deploy` to deploy their new code into the cluster.

## Enabling Testing and Debugging

After a user has successfully deployed their development version of their application, they need to test it and, if there are problems, debug any issues with the application. This can also be a hurdle when developing in Kubernetes because it is not always clear how to interact with your cluster. The `kubectl` command line is a veritable Swiss army knife of tools to achieve this, from `kubectl logs` to `kubectl exec` and `kubectl port-forward`, but learning how to use all of the different options and achieving familiarity with the tool can take a considerable amount of experience. Furthermore, because the tool runs in the terminal, it often requires the composition of

multiple windows to simultaneously examine both the source code for the application and the running application itself.

To streamline the testing and debugging experience, Kubernetes tooling is increasingly being integrated into development environments, for example, the open source extension for Visual Studio (VS) Code for Kubernetes. The extension is easily installed for free from the VS Code marketplace. When installed, it automatically discovers any clusters that you already have in your *kubeconfig* file, and it provides a tree-view navigation pane for you to see the contents of your cluster at a glance.

In addition to being able to see your cluster state at a glance, the integration allows a developer to use the tools available via `kubectl` in an intuitive, discoverable way. From the tree view, if you right-click a Kubernetes pod, you can immediately use port forwarding to bring a network connection to the pod directly to the local machine. Likewise, you can access the logs for the pod or even get a terminal within the running container.

The integration of these commands with prototypical user interface expectations (e.g., right-click shows a context menu), as well as the integration of these experiences alongside the code for the application itself, enable developers with minimal Kubernetes experience to rapidly become productive in the development cluster.

Of course this VS Code extension isn't the only integration between Kubernetes and a devlopment environment; there are several others that you can install depending on your choice of programming environment and style (`vi`, `emacs`, etc.).

## Setting Up a Development Environment Best Practices

Setting up successful workflows on Kubernetes is key to productivity and happiness. Following these best practices will help to ensure that developers are up and running quickly:

- Think about developer experience in three phases: onboarding, developing, and testing. Make sure that the development environment you build supports all three of these phases.
- When building a development cluster, you can choose between one large cluster and a cluster per developer. There are pros and cons to each, but generally a single large cluster is a better approach.
- When you add users to a cluster, add them with their own identity and access to their own namespace. Use resource limits to restrict how much of the cluster they can use.
- When managing namespaces, think about how you can reap old, unused resources. Developers will have bad hygiene about deleting unused things. Use automation to clean it up for them.

- Think about cluster-level services like logs and monitoring that you can set up for all users. Sometimes, cluster-level dependencies like databases are also useful to set up on behalf of all users using templates like Helm charts.

## Summary

We've reached a place where creating a Kubernetes cluster, especially in the cloud, is a relatively straightforward exercise, but enabling developers to productively use such a cluster is significantly less obvious and easy. When thinking about enabling developers to successfully build applications on Kubernetes, it's important to think about the key goals around onboarding, iterating, testing, and debugging applications. Likewise, it pays to invest in some basic tooling specific to user onboarding, namespace provisioning, and cluster services like basic log aggregation. Viewing a development cluster and your code repositories as an opportunity to standardize and apply best practices will ensure that you have happy and productive developers, successfully building code to deploy to your production Kubernetes clusters.

CHAPTER 3

# Monitoring and Logging in Kubernetes

In this chapter, we discuss best practices for monitoring and logging in Kubernetes. We'll dive into the details of different monitoring patterns, important metrics to collect, and building dashboards from these raw metrics. We then wrap up with examples of implementing monitoring for your Kubernetes cluster.

## Metrics Versus Logs

You first need to understand the difference between log collection and metrics collection. They are complementary to each other but serve different purposes.

*Metrics*
: A series of numbers measured over a period of time

*Logs*
: Used for exploratory analysis of a system

An example of where you would need to use both metrics and logging is when an application is performing poorly. Our first indication of the issue might be an alert of high latency on the pods hosting the application, but the metrics might not give a good indication of the issue. We then can look into our logs to perform an investigation of errors that are being emitted from the application.

## Monitoring Techniques

Black-box monitoring focuses on monitoring from the outside of an application and is what's been used traditionally when monitoring systems for components like CPU, memory, storage, and so on. Black-box monitoring can still be useful for monitoring at the infrastructure level, but it lacks insights and context into how the application is operating. For example, to test whether a cluster is healthy, we might schedule a pod,

and if it's successful, we know that the scheduler and service discovery are healthy within our cluster, so we can assume the cluster components are healthy.

White-box monitoring focuses on the details in the context of the application state, such as total HTTP requests, number of 500 errors, latency of requests, and so on. With white-box monitoring, we can begin to understand the "Why" of our system state. It allows us to ask, "Why did the disk fill up?" and not just, "The disk filled up."

## Monitoring Patterns

You might look at monitoring and say, "How difficult can this be? We've always monitored our systems." Yes, some of your typical monitoring patterns in place today also fit into how you monitor Kubernetes. The difference is that platforms like Kubernetes are much more dynamic and transient, and you'll need to change your thinking about how to monitor these environments. For example, when monitoring a virtual machine (VM) you expect that VM to be up 24/7 and all its state preserved. In Kubernetes, pods can be very dynamic and short-lived, so you need to have monitoring in place that can handle this dynamic and transient nature.

There are a couple of different monitoring patterns to focus on when monitoring distributed systems.

The *USE* method, popularized by Brendan Gregg, focuses on the following:

- U—Utilization
- S—Saturation
- E—Errors

This method is focused on infrastructure monitoring because there are limitations on using it for application-level monitoring. The USE method is described as, "For every resource, check utilization, saturation, and error rates." This method lets you quickly identify resource constraints and error rates of your systems. For example, to check the health of the network for your nodes in the cluster, you will want to monitor the utilization, saturation, and error rate to be able to easily identify any network bottlenecks or errors in the network stack. The USE method is a tool in a larger toolbox and is not the only method you will utilize to monitor your systems.

Another monitoring approach, called the *RED* method, was popularized by Tom Willke. The RED method approach is focused on the following:

- R—Rate
- E—Errors
- D—Duration

The philosophy was taken from Google's *Four Golden Signals*:

- Latency (how long it takes to serve a request)
- Traffic (how much demand is placed on your system)
- Errors (rate of requests that are failing)
- Saturation (how utilized your service is)

As an example, you could use this method to monitor a frontend service running in Kubernetes to calculate the following:

- How many requests is my frontend service processing?
- How many 500 errors are users of the service receiving?
- Is the service overutilized by requests?

As you can see from the previous example, this method is more focused on the experience of the users and their experience with the service.

The USE and RED methods are complementary to each other given that the USE method focuses on the infrastructure components and the RED method focuses on monitoring the end-user experience for the application.

# Kubernetes Metrics Overview

Now that we know the different monitoring techniques and patterns, let's look at what components you should be monitoring in your Kubernetes cluster. A Kubernetes cluster consists of control-plane components and worker-node components. The control-plane components consist of the API Server, etcd, scheduler, and controller manager. The worker nodes consist of the kubelet, container runtime, kube-proxy, kube-dns, and pods. You need to monitor all these components to ensure a healthy cluster and application.

Kubernetes exposes these metrics in a variety of ways, so let's take a look at different components that you can use to collect metrics within your cluster.

## cAdvisor

Container Advisor, or cAdvisor, is an open source project that collects resources and metrics for containers running on a node. cAdvisor is built into the Kubernetes kubelet, which runs on every node in the cluster. It collects memory and CPU metrics through the Linux control group (cgroup) tree. If you are not familiar with cgroups, it's a Linux kernel feature that allows isolation of resources for CPU, disk I/O, or network I/O. cAdvisor will also collect disk metrics through statfs, which is built into the Linux kernel. These are implementation details you don't really need to worry

about, but you should understand how these metrics are exposed and the type of information you can collect. You should consider cAdvisor as the source of truth for all container metrics.

## Metrics Server

The Kubernetes metrics server and Metrics Server API are a replacement for the deprecated Heapster. Heapster had some architectural disadvantages with how it implemented the data sink, which caused a lot of vendored solutions in the core Heapster code base. This issue was solved by implementing a resource and Custom Metrics API as an aggregated API in Kubernetes. This allows implementations to be switched out without changing the API.

There are two aspects to understand in the Metrics Server API and metrics server.

First, the canonical implementation of the Resource Metrics API is the metrics server. The metrics server gathers resource metrics such as CPU and memory. It gathers these metrics from the kubelet's API and then stores them in memory. Kubernetes uses these resource metrics in the scheduler, Horizontal Pod Autoscaler (HPA), and Vertical Pod Autoscaler (VPA).

Second, the Custom Metrics API allows monitoring systems to collect arbitrary metrics. This allows monitoring solutions to build custom adapters that will allow for extending outside the core resource metrics. For example, Prometheus built one of the first custom metrics adapters, which allows you to use the HPA based on a custom metric. This opens up better scaling based on your use case because now you can bring in metrics like queue size and scale based on a metric that might be external to Kubernetes.

Now that there is a standardized Metrics API, this opens up many possibilities to scale outside the plain old CPU and memory metrics.

## kube-state-metrics

kube-state-metrics is a Kubernetes add-on that monitors the object stored in Kubernetes. Where cAdvisor and metrics server are used to provide detailed metrics on resource usage, kube-state-metrics is focused on identifying conditions on Kubernetes objects deployed to your cluster.

Following are some questions that kube-state-metrics can answer for you:

- Pods
  - — How many pods are deployed to the cluster?
  - — How many pods are in a pending state?
  - — Are there enough resources to serve a pods request?

- Deployments
  - How many pods are in a running state versus a desired state?
  - How many replicas are available?
  - What deployments have been updated?
- Nodes
  - What's the status of my worker nodes?
  - What are the allottable CPU cores in my cluster?
  - Are there any nodes that are unschedulable?
- Jobs
  - When did a job start?
  - When did a job complete?
  - How many jobs failed?

As of this writing, there are 22 object types that kube-state-metrics tracks. These are always expanding, and you can find the documentation in the Github repository.

## What Metrics Do I Monitor?

The easy answer is "Everything," but if you try to monitor too much, you can create too much noise that filters out the real signals into which you need to have insight. When we think about monitoring in Kubernetes, we want to take a layered approach that takes into account the following:

- Physical or virtual nodes
- Cluster components
- Cluster add-ons
- End-user applications

Using this layered approach to monitoring allows you to more easily identify the correct signals in your monitoring system. It allows you to approach issues with a more targeted approach. For example, if you have pods going into a pending state, you can start with resource utilization of the nodes, and if all is OK, you can target cluster-level components.

Following are metrics you would want to target in your system:

- Nodes
  - CPU utilization
  - Memory utilization

  - — Network utilization
  - — Disk utilization
- Cluster components
  - — etcd latency
- Cluster add-ons
  - — Cluster Autoscaler
  - — Ingress controller
- Application
  - — Container memory utilization and saturation
  - — Container CPU utilization
  - — Container network utilization and error rate
  - — Application framework-specific metrics

# Monitoring Tools

There are many monitoring tools that can integrate with Kubernetes, and more arriving every day, building on their feature set to have better integration with Kubernetes. Following are a few popular tools that integrate with Kubernetes:

*Prometheus*
: Prometheus is an open source systems monitoring and alerting toolkit originally built at SoundCloud. Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user community. It is now a standalone open source project and maintained independent of any company. To emphasize this, and to clarify the project's governance structure, Prometheus joined the Cloud Native Computing Foundation (CNCF) in 2016 as the second hosted project, after Kubernetes.

*InfluxDB*
: InfluxDB is a time-series database designed to handle high write and query loads. It is an integral component of the TICK (Telegraf, InfluxDB, Chronograf, and Kapacitor) stack. InfluxDB is meant to be used as a backing store for any use case involving large amounts of timestamped data, including DevOps monitoring, application metrics, IoT sensor data, and real-time analytics.

*Datadog*
: Datadog provides a monitoring service for cloud-scale applications, providing monitoring of servers, databases, tools, and services through a SaaS-based data analytics platform.

*Sysdig*
: Sysdig Monitor is a commercial tool that provides Docker monitoring and Kubernetes monitoring for container-native apps. Sysdig also allows you to collect, correlate, and query Prometheus metrics with direct Kubernetes integration.

*Cloud provider tools*

*GCP Stackdriver*
: Stackdriver Kubernetes Engine Monitoring is designed to monitor Google Kubernetes Engine (GKE) clusters. It manages monitoring and logging services together and features an interface that provides a dashboard customized for GKE clusters. Stackdriver Monitoring provides visibility into the performance, uptime, and overall health of cloud-powered applications. It collects metrics, events, and metadata from Google Cloud Platform (GCP), Amazon Web Services (AWS), hosted uptime probes, and application instrumentation.

*Microsoft Azure Monitor for containers*
: Azure Monitor for containers is a feature designed to monitor the performance of container workloads deployed to either Azure Container Instances or managed Kubernetes clusters hosted on Azure Kubernetes Service. Monitoring your containers is critical, especially when you're running a production cluster, at scale, with multiple applications. Azure Monitor for containers gives you performance visibility by collecting memory and processor metrics from controllers, nodes, and containers that are available in Kubernetes through the Metrics API. Container logs are also collected. After you enable monitoring from Kubernetes clusters, metrics and logs are automatically collected for you through a containerized version of the Log Analytics agent for Linux.

*AWS Container Insights*
: If you use Amazon Elastic Container Service (ECS), Amazon Elastic Kubernetes Service, or other Kubernetes platforms on Amazon EC2, you can use CloudWatch Container Insights to collect, aggregate, and summarize metrics and logs from your containerized applications and microservices. The metrics include utilization for resources such as CPU, memory, disk, and network. Container Insights also provides diagnostic information, such as container restart failures, to help you isolate issues and resolve them quickly.

One important aspect when looking at implementing a tool to monitor metrics is to look at how the metrics are stored. Tools that provide a time-series database with key/value pairs will give you a higher degree of attributes for the metric.

Always evaluate monitoring tools you already have, because taking on a new monitoring tool has a learning curve and a cost due to the operational implementation of the tool. Many of the monitoring tools now have integration into Kubernetes, so evaluate which ones you have today and whether they will meet your requirements.

## Monitoring Kubernetes Using Prometheus

In this section we focus on monitoring metrics with Prometheus, which provides good integrations with Kubernetes labeling, service discovery, and metadata. The high-level concepts we implement throughout the chapter will also apply to other monitoring systems.

Prometheus is an open source project that is hosted by the CNCF. It was originally developed at SoundCloud, and a lot of its concepts are based on Google's internal monitoring system, BorgMon. It implements a multidimensional data model with keypairs that work much like how the Kubernetes labeling system works. Prometheus exposes metrics in a human-readable format, as in the following example:

```
# HELP node_cpu_seconds_total Seconds the CPU is spent in each mode.
# TYPE node_cpu_seconds_total counter
node_cpu_seconds_total{cpu="0",mode="idle"} 5144.64
node_cpu_seconds_total{cpu="0",mode="iowait"} 117.98
```

To collect metrics, Prometheus uses a pull model in which it scrapes a metrics endpoint to collect and ingest the metrics into the Prometheus server. Systems like Kubernetes already expose their metrics in a Prometheus format, making it simple to collect metrics. Many other Kubernetes ecosystem projects (NGINX, Traefik, Istio, LinkerD, etc.) also expose their metrics in a Prometheus format. Prometheus also can use exporters, which allow you to take emitted metrics from your service and translate them to Prometheus-formatted metrics.

Prometheus has a very simplified architecure, as depicted in Figure 3-1.

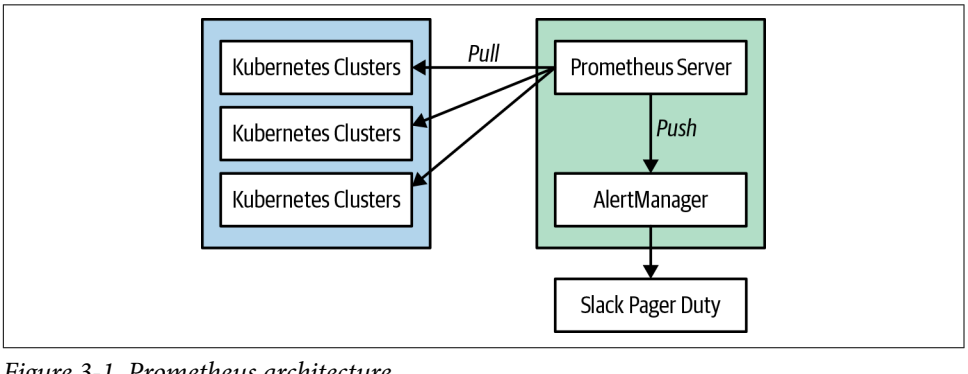



*Figure 3-1. Prometheus architecture*

You can install Prometheus within the cluster or outside the cluster. It's a good practice to monitor your cluster from a "utility cluster" to avoid a production issue also affecting your monitoring system. There are tools like Thanos that provide high availability for Prometheus and allow you to export metrics into an external storage system.

A deep dive into the Prometheus architecture is beyond the scope of this book, and you should refer to another one of the dedicated books on this topic. *Prometheus: Up & Running* (O'Reilly) is a good in-depth book to get you started.

So, let's dive in and get Prometheus set up on our Kubernetes cluster. There are many different ways to do this, and the deployment will depend on your specific implementation. In this chapter we install the Prometheus Operator:

*Prometheus Server*
: Pulls and stores metrics being collected from systems.

*Prometheus Operator*
: Makes the Prometheus configuration Kubernetes native, and manages and operates Prometheus and Alertmanager clusters. Allows you to create, destroy, and configure Prometheus resources through native Kubernetes resource definitions.

*Node Exporter*
: Exports host metrics from Kubernetes nodes in the cluster.

*kube-state-metrics*
: Collects Kubernetes-specific metrics.

*Alertmanager*
: Allows you to configure and forward alerts to external systems.

*Grafana*
: Provides visualization on dashboard capabilities for Prometheus.

```
helm install --name prom stable/prometheus-operator
```

After you've installed the Operator, you should see the following pods deployed to your cluster:

```
$ kubectl get pods -n monitoring
NAME                                  READY   STATUS    RESTARTS   AGE
alertmanager-main-0                   2/2     Running   0          5h39m
alertmanager-main-1                   2/2     Running   0          5h39m
alertmanager-main-2                   2/2     Running   0          5h38m
grafana-5d8f767-ct2ws                 1/1     Running   0          5h39m
kube-state-metrics-7fb8b47448-k6j6g   4/4     Running   0          5h39m
node-exporter-5zk6k                   2/2     Running   0          5h39m
node-exporter-874ss                   2/2     Running   0          5h39m
```

```
node-exporter-9mtgd                     2/2     Running   0          5h39m
node-exporter-w6xwt                     2/2     Running   0          5h39m
prometheus-adapter-66fc7797fd-ddgk5     1/1     Running   0          5h39m
prometheus-k8s-0                        3/3     Running   1          5h39m
prometheus-k8s-1                        3/3     Running   1          5h39m
prometheus-operator-7cb68545c6-gm84j    1/1     Running   0          5h39m
```

Lets take a look at the Prometheus Server to see how you can run some queries to retrieve Kubernetes metrics:

```
kubectl port-forward svc/prom-prometheus-operator-prometheus 9090
```

This creates a tunnel to our localhost on port 9090. Now, we can open a web browser and connect to the Prometheus server on *http://127.0.0.1:9090*.

Figure 3-2 depicts the screen you'll see if you successfully deployed Prometheus to your cluster.

Now that we have Prometheus deployed, let's explore some Kubernetes metrics through the Prometheus PromQL query language. There is a PromQL Basics guide available.

We talked earlier in the chapter about employing the USE method, so let's gather some node metrics on CPU utilization and saturation.

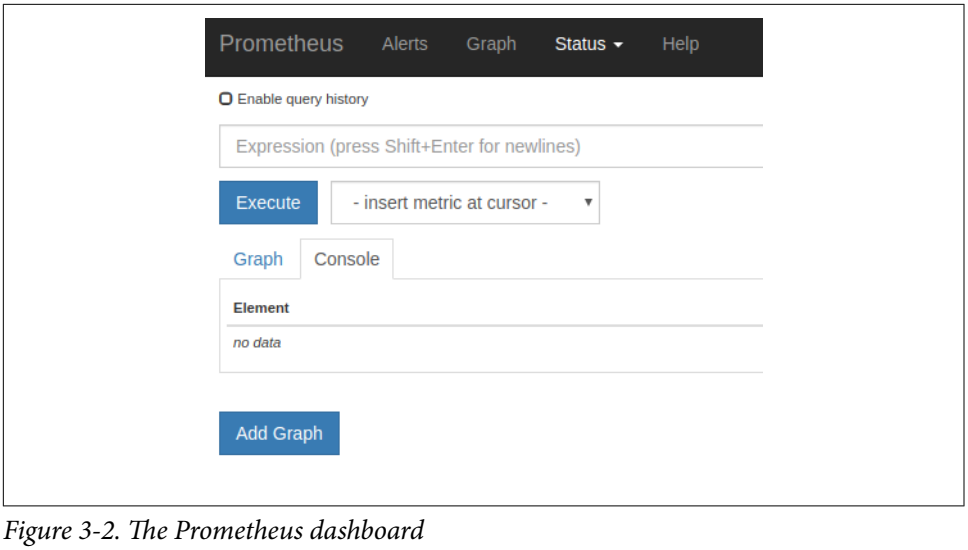



*Figure 3-2. The Prometheus dashboard*

In the Expression input, enter the following query:

```
avg(rate(node_cpu_seconds_total[5m]))
```

This will return the average CPU utilization for the entire cluster.

If we want to get the CPU utilization per node, we can write a query like the following:

```
avg(rate(node_cpu_seconds_total[5m])) by (node_name)
```

This returns average CPU utilization for each node in the cluster.

So, now that you have some experience with running queries within Prometheus, let's take a look at how Grafana can help build dashboard visualization for these common USE method metrics we want to track. The great thing about the Prometheus Operator you installed is that it comes with some prebuilt Grafana dashboards that you can use.

You'll now need to create a port-forward tunnel to the Grafana pod so that you can access it from your local machine:

```
kubectl port-forward svc/prom-grafana 3000:3000
```

Now, point your web browser at *http://localhost:3000* and log in using the following credentials:

- Username: admin
- Password: admin

Under the Grafana dashboard you'll find a dashboard called Kubernetes / USE Method / Cluster. This dashboard gives you a good overview of the utilization and saturation of the Kubernetes cluster, which is at the heart of the USE method. Figure 3-3 presents an example of the dashboard.


*Figure 3-3. A Grafana dashboard*

Go ahead and take some time to explore the different dashboards and metrics that you can visualize in Grafana.

Avoid creating too many dashboards (aka "The Wall of Graphs") because this can be difficult for engineers to reason with in troubleshooting situations. You might think having more information in a dashboard means better monitoring, but the majority of the time it causes more confusion for a user looking at the dashboard. Focus your dashboard design on outcomes and time to resolution.

## Logging Overview

Up to this point, we have discussed a lot about metrics and Kubernetes, but to get the full picture of your environment, you also need to collect and centralize logs from the Kubernetes cluster and the applications deployed to your cluster.

With logging, it might be easy to say, "Let's just log everything," but this can cause two issues:

- There is too much noise to find issues quickly.
- Logs can consume a lot of resources and come with a high cost.

There is no clear-cut answer to what exactly you should log because debug logs become a necessary evil. Over time you'll start to understand your environment better and learn what noise you can tune out from the logging system. Also, to address the ever-increasing amount of logs stored, you will need to implement a retention and archival policy. From an end-user experience, having somewhere between 30 and 45 days worth of historical logs is a good fit. This allows for investigation of problems that manifest over a longer period of time, but also reduces the amount of resources needed to store logs. If you require longer-term storage for compliance reasons, you'll want to archive the logs to more cost-effective resources.

In a Kubernetes cluster, there are multiple components to log. Following is a list of components from which you should be collecting metrics:

- Node logs
- Kubernetes control-plane logs
  - — API server
  - — Controller manager
  - — Scheduler
- Kubernetes audit logs
- Application container logs

With node logs, you want to collect events that happen to essential node services. For example, you will want to collect logs from the Docker daemon running on the

worker nodes. A healthy Docker daemon is essential for running containers on the worker node. Collecting these logs will help you diagnose any issues that you might run into with the Docker daemon, and it will give you information into any underlying issues with the daemon. There are also other essential services that you will want to log from the underlying node.

The Kubernetes control plane consists of several components from which you'll need to collect logs to give you more insight into underlying issues within it. The Kubernetes control plane is core to a healthy cluster, and you'll want to aggregate the logs that it stores on the host in */var/log/kube-APIserver.log*, */var/log/kube-scheduler.log*, and */var/log/kube-controller-manager.log*. The controller manager is responsible for creating objects defined by the end user. As an example, as a user you create a Kubernetes service with type LoadBalancer and it just sits in a pending state; the Kubernetes events might not give all the details to diagnose the issue. If you collect the logs in a centralized system, it will give you more detail into the underlying issue and a quicker way to investigate the issue.

You can think of Kubernetes audit logs as security monitoring because they give you insight into who did what within the system. These logs can be very noisy, so you'll want to tune them for your environment. In many instances these logs can cause a huge spike in your logging system when first initialized, so make sure that you follow the Kubernetes documentation guidance on audit log monitoring.

Application container logs give you insight into the actual logs your application is emitting. You can forward these logs to a central repository in multiple ways. The first and recommended way is to send all application logs to STDOUT because this gives you a uniform way of application logging, and a monitoring daemon set can gather the logs directly from the Docker daemon. The other way is to use a *sidecar* pattern and run a log forwarding container next to the application container in a Kubernetes pod. You might need to use this pattern if your application logs to the filesystem.

There are many options and configurations for managing Kubernetes audit logs. These audit logs can be very noisy and it can be expensive to log all actions. You should consider looking at the audit logging documentation, so that you can fine-tune these logs for your environment.

# Tools for Logging

Like collecting metrics there are numerous tools to collect logs from Kubernetes and applications running in the cluster. You might already have tooling for this, but be aware of how the tool implements logging. The tool should have the capability to run as a Kubernetes DaemonSet and also have a solution to run as a sidecar for

applications that don't send logs to STDOUT. Utilizing an existing tool can be advantageous because you will already have a lot of operational knowledge of the tool.

Some of the more popular tools with Kubernetes integration are:

- Elastic Stack
- Datadog
- Sumo Logic
- Sysdig
- Cloud provider services (GCP Stackdriver, Azure Monitor for containers, and Amazon CloudWatch)

When looking for a tool to centralize logs, hosted solutions can provide a lot of value because they offload a lot of the operational cost. Hosting your own logging solution seems great on day *N*, but as the environment grows, it can be very time consuming to maintain the solution.

## Logging by Using an EFK Stack

For the purposes of this book, we use an Elasticsearch, Fluentd, and Kibana (EFK) stack to set up monitoring for our cluster. Implementing an EFK stack can be a good way to get started, but at some point you'll probably ask yourself, "Is it really worth managing my own logging platform?" Typically it's not worth the effort because self-hosted logging solutions are great on day one, but they become overly complex by day 365. Self-hosted logging solutions become more operationally complex as your environment scales. There is no one correct answer, so evaluate whether your business requirements need you to host your own solution. There are also a number of hosted solutions based on the EFK stack, so you can always move pretty easily if you choose not to host it yourself.

You will deploy the following for your monitoring stack:

- Elasticsearch Operator
- Fluentd (forwards logs from our Kubernetes environment into Elasticsearch)
- Kibana (visualization tool to search, view, and interact with logs stored in Elasticsearch)

Deploy the manifest to your Kubernetes cluster:

```
kubectl create namespace logging

kubectl apply -f https://raw.githubusercontent.com/dstrebel/kbp/master/
elasticsearch-operator.yaml -n logging
```

Deploy the Elasticsearch operator to aggregate all forwarded logs:

```
kubectl apply -f https://raw.githubusercontent.com/dstrebel/kbp/master/efk.yaml
-n logging
```

This deploys Fluentd and Kibana, which will allow us to forward logs to Elasticsearch and visualize the logs using Kibana.

You should see the following pods deployed to your cluster:

```
kubectl get pods -n logging

efk-kibana-854786485-knhl5                1/1     Running   0          4m
elasticsearch-operator-5647dc6cb-tc2st    1/1     Running   0          5m
elasticsearch-operator-sysctl-ktvk9       1/1     Running   0          5m
elasticsearch-operator-sysctl-lf2zs       1/1     Running   0          5m
elasticsearch-operator-sysctl-r8qhb       1/1     Running   0          5m
es-client-efk-cluster-9f4cc859-sdrsl      1/1     Running   0          4m
es-data-efk-cluster-default-0             1/1     Running   0          4m
es-master-efk-cluster-default-0           1/1     Running   0          4m
fluent-bit-4kxdl                          1/1     Running   0          4m
fluent-bit-tmqjb                          1/1     Running   0          4m
fluent-bit-w6fs5                          1/1     Running   0          4m
```

After all pods are "Running," let's go ahead and connect to Kibana through port forwarding to our localhost:

```
export POD_NAME=$(kubectl get pods --namespace logging -l
"app=kibana,release=efk" -o jsonpath="{.items[0].metadata.name}")

kubectl port-forward $POD_NAME 5601:5601
```

Now point your web browser at *http://localhost:5601* to open the Kibana dashboard.

To interact with the logs forwarded from our Kubernetes cluster, you first need to create an index.

The first time you start Kibana, you will need to navigate to the Management tab, and create an index pattern for Kubernetes logs. The system will guide you through the required steps.

After you create an index, you can search through logs using a Lucene query syntax, such as the following:

```
log:(WARN|INFO|ERROR|FATAL)
```

This returns all logs containing the fields warn, info, error, or fatal. You can see an example in Figure 3-4.

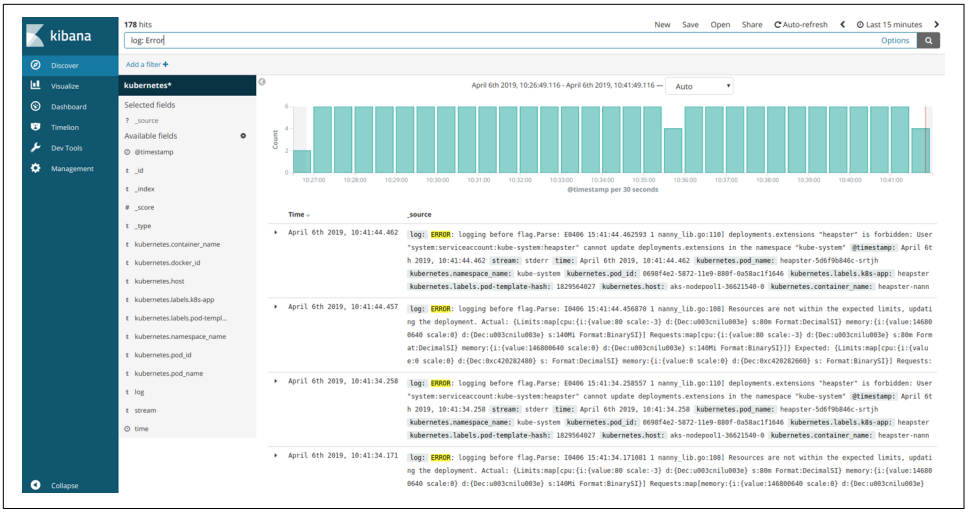



*Figure 3-4. The Kibana dashboard*

In Kibana, you can perform ad hoc queries on the logs, and you can build out dashboards to give you an overview of the environment.

Go ahead and take some time to explore the different logs that you can visualize in Kibana.

# Alerting

Alerting is a double-edged sword, and you need to strike a balance on what you alert on versus what should just be monitored. Alerting on too much causes alert fatigue, and important events will be lost in all the noise. An example would be generating an alert any time a pod fails. You might be asking, "Why wouldn't I want to monitor for a pod failure?" Well, the beauty of Kubernetes is that it provides features to automatically check the health of a container and restart the container automatically. You really want to focus alerting on events that affect your Service-Level Objectives (SLOs). SLOs are specific measurable characteristics such as availability, throughput, frequency, and response time that you agree upon with the end user of your service. Setting SLOs sets expectations with your end users and provides clarity on how the system should behave. Without an SLO, users can form their opinion, which might be an unrealistic expectation of the service. Alerting in a system like Kubernetes needs an entirely new approach from what we are typically accustomed to and needs to focus on how the end user is experiencing the service. For example, if your SLO for a frontend service is a 20-ms response time and you are seeing higher latency than average, you want to be alerted on the problem.

You need to decide what alerts are good and require intervention. In typical monitoring, you might be accustomed to alerting on high CPU usage, memory usage, or processes not responding. These might seem like good alerts, but probably don't indicate an issue that someone needs to take immediate action on and requires notifying an on-call engineer. An alert to an on-call engineer should be an issue that needs immediate human attention and is affecting the UX of the application. If you have ever experienced a "That issue resolved itself" scenario, then that is a good indication that the alert did not need to contact an on-call engineer.

One way to handle alerts that don't need immediate action is to focus on automating the remediation of the cause. For example, when a disk fills up, you could automate the deletion of logs to free up space on the disk. Also, utilizing Kubernetes *liveness probes* in your app deployment can help autoremediate issues with a process that is not responding in the application.

When building alerts, you also need to consider *alert thresholds*; if you set thresholds too short, then you can get a lot of false positives with your alerts. It's generally recommended to set a threshold of at least five minutes to help eliminate false positives. Coming up with standard thresholds can help define a standard and avoid micromanaging many different thresholds. For example, you might want to follow a specific pattern of 5 minutes, 10 minutes, 30 minutes, 1 hour, and so on.

When building notifications for alerts you want to ensure that you provide relevant information in the notification, for example, providing a link to a "playbook" that gives troubleshooting or other helpful information on resolving the issue. You should also include information on the datacenter, region, app owner, and affected system in notifications. Providing all this information will allow engineers to quickly formalize a theory around the issue.

You also need to build notification channels to route alerts that are fired. When thinking about "Who do I notify when an alert is triggered?" you should ensure that notifications are not just sent to a distribution list or team emails. What tends to happen if alerts are sent to larger groups is that they end up getting filtered out because users see these as noise. You should route notifications to the user who is going to take responsibility for the issue.

With alerting, you'll never get it perfect on day one, and we could argue it might never be perfect. You just want to make sure that you incrementally improve on alerting to preclude alert fatigue, which can cause many issues with staff burnout and your systems.

For further insight on how to approach alerting on and managing systems, read "My Philosophy on Alerting" by Rob Ewaschuk, which is based on Rob's observations as a site reliability engineer (SRE) at Google.

# Best Practices for Monitoring, Logging, and Alerting

Following are the best practices that you should adopt regarding monitoring, logging, and alerting.

## Monitoring

- Monitor nodes and all Kubernetes components for utilization, saturation, and error rates, and monitor applications for rate, errors, and duration.
- Use black-box monitoring to monitor for symptoms and not predictive health of a system.
- Use white-box monitoring to inspect the system and its internals with instrumentation.
- Implement time-series-based metrics to gain high-precision metrics that also allow you to gain insight within the behavior of your application.
- Utilize monitoring systems like Prometheus that provide key labeling for high dimensionality; this will give a better signal to symptoms of an impacting issue.
- Use average metrics to visualize subtotals and metrics based on factual data. Utilize sum metrics to visualize the distribution across a specific metric.

## Logging

- You should use logging in combination with metrics monitoring to get the full picture of how your environment is operating.
- Be cautious of storing logs for more than 30 to 45 days and, if needed, use cheaper resources for long-term archiving.
- Limit usage of log forwarders in a sidecar pattern, as they will utilize a lot more resources. Opt for using a DaemonSet for the log forwarder and sending logs to STDOUT.

## Alerting

- Be cautious of alert fatigue because it can lead to bad behaviors in people and processes.
- Always look at incrementally improving upon alerting and accept that it will not always be perfect.
- Alert for symptoms that affect your SLO and customers and not for transient issues that don't need immediate human attention.

## Summary

In this chapter we discussed the patterns, techniques, and tools that can be used for monitoring our systems with metric and log collection. The most important piece to take away from this chapter is that you need to rethink how you perform monitoring and do it from the outset. Too many times we see this implemented after the fact, and it can get you into a very bad place in understanding your system. Monitoring is all about having better insight into a system and being able to provide better resiliency, which in turn provides a better end-user experience for your application. Monitoring distributed applications and distributed systems like Kubernetes requires a lot of work, so you must be ready for it at the beginning of your journey.

CHAPTER 4

# Configuration, Secrets, and RBAC

The composable nature of containers allows us as operators to introduce configuration data into a container at runtime. This makes it possible for us to decouple an application's function from the environment it runs in. By means of the conventions allowed in the container runtime to pass through either environment variables or mount external volumes into a container at runtime, you can effectively change the configuration of the application upon its instantiation. As a developer, it is important to take into consideration the dynamic nature of this behavior and allow for the use of environment variables or the reading of configuration data from a specific path available to the application runtime user.

When moving sensitive data such as secrets into a native Kubernetes API object, it is important to understand how Kubernetes secures access to the API. The most commonly implemented security method in use in Kubernetes is Role-Based Access Control (RBAC) to implement a fine-grained permission structure around actions that can be taken against the API by specific users or groups. This chapter covers some of the best practices regarding RBAC and also provides a small primer.

## Configuration Through ConfigMaps and Secrets

Kubernetes allows you to natively provide configuration information to our applications through ConfigMaps or secret resources. The main differentiator between the two is the way a pod stores the receiving information and how the data is stored in the etcd data store.

### ConfigMaps

It is very common to have applications consume configuration information through some type of mechanism such as command-line arguments, environment variables,

or files that are available to the system. Containers allow the developer to decouple this configuration information from the application, which allows for true application portability. The ConfigMap API allows for the injection of supplied configuration information. ConfigMaps are very adaptable to the application's requirements and can provide key/value pairs or complex bulk data such as JSON, XML, or proprietary configuration data.

The ConfigMaps not only provide configuration information for pods, but can also provide information to be consumed for more complex system services such as controllers, CRDs, operators, and so on. As mentioned earlier, the ConfigMap API is meant more for string data that is not really sensitive data. If your application requires more sensitive data, the Secrets API is more appropriate.

For your application to use the ConfigMap data, it can be injected as either a volume mounted into the pod or as environment variables.

## Secrets

Many of the attributes and reasons for which you would want to use a ConfigMap apply to secrets. The main differences lie in the fundamental nature of a Secret. Secret data should be stored and handled in a way that can be easily hidden and possibly encrypted at rest if the environment is configured as such. The Secret data is represented as base64-encoded information, and it is critical to understand that this is not encrypted. As soon as the secret is injected into the pod, the pod itself can see the secret data in plain text.

Secret data is meant to be small amounts of data, limited by default in Kubernetes to 1 MB in size, for the base64-encoded data, so ensure that the actual data is approximately 750 KB because of the overhead of the encoding. There are three types of secrets in Kubernetes:

`generic`
: This is typically just regular key/value pairs that are created from a file, a directory, or from string literals using the `--from-literal=` parameter, as follows:

```
kubectl create secret generic mysecret --from-literal=key1=$3cr3t1 --
from-literal=key2=@3cr3t2`
```

`docker-registry`
: This is used by the kubelet when passed in a pod template if there is an `imagePullsecret` to provide the credentials needed to authenticate to a private Docker registry:

```
kubectl create secret docker-registry registryKey --docker-server
myreg.azurecr.io --docker-username myreg --docker-password $up3r
$3cr3tP@ssw0rd --docker-email ignore@dummy.com
```

`tls`
: This creates a Transport Layer Security (TLS) secret from a valid public/private key pair. As long as the cert is in a valid PEM format, the key pair will be encoded as a secret and can be passed to the pod to use for SSL/TLS needs:

```
kubectl create secret tls www-tls --key=./path_to_key/wwwtls.key --
cert=./path_to_crt/wwwtls.crt
```

Secrets are also mounted into tmpfs only on the nodes that have a pod that requires the secret and are deleted when the pod that needs it is gone. This prevents any secrets from being left behind on the disk of the node. Although this might seem secure, it is important to know that by default, secrets are stored in the etcd datastore of Kubernetes in plain text, and it is important that the system administrators or cloud service provider take efforts to ensure that the security of the etcd environment, including mTLS between the etcd nodes and enabling encryption at rest for the etcd data. More recent versions of Kubernetes use etcd3 and have the ability to enable etcd native encryption; however, this is a manual process that must be configured in the API server configuration by specifying a provider and the proper key media to properly encrypt secret data held in etcd. As of Kubernetes v1.10 (it has been promoted to beta in v1.12), we have the KMS provider, which promises to provide a more secure key process by using third-party KMS systems to hold the proper keys.

## Common Best Practices for the ConfigMap and Secrets APIs

The majority of issues that arise from the use of a ConfigMap or secret are incorrect assumptions on how changes are handled when the data held by the object is updated. By understanding the rules of the road and adding a few tricks to make it easier to abide by those rules, you can steer away from trouble:

- To support dynamic changes to your application without having to redeploy new versions of the pods, mount your ConfigMaps/Secrets as a volume and configure your application with a file watcher to detect the changed file data and reconfigure itself as needed. The following code shows a Deployment that mounts a ConfigMap and a Secret file as a volume:

```
apiVersion: v1
kind: ConfigMap
metadata:
    name: nginx-http-config
    namespace: myapp-prod
data:
  config: |
    http {
      server {
```

```
        location / {
        root /data/html;
        }

        location /images/ {
          root /data;
        }
      }
    }
apiVersion: v1
kind: Secret
metadata:
  name: myapp-api-key
type: Opaque
data:
  myapikey: YWRtd5thSaW4=

apiVersion: apps/v1
kind: Deployment
metadata:
  name: mywebapp
  namespace: myapp-prod
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 8080
    volumeMounts:
    - mountPath: /etc/nginx
      name: nginx-config
    - mountPath: /usr/var/nginx/html/keys
      name: api-key
  volumes:
    - name: nginx-config
      configMap:
        name: nginx-http-config
        items:
        - key: config
          path: nginx.conf
    - name: api-key
      secret:
        name: myapp-api-key
        secretname: myapikey
```

There are a couple of things to consider when using `volumeMounts`. First, as soon as the ConfigMap/Secret is created, add it as a volume in your pod's specification. Then mount that volume into the container's filesystem. Each property name in the ConfigMap/Secret will become a new file in the mounted directory, and the contents of each file will be the value specified in the ConfigMap/Secret. Second, avoid mounting ConfigMaps/Secrets using the `volumeMounts.subPath` property. This will prevent the data from being dynamically updated in the volume if you update a ConfigMap/Secret with new data.

- ConfigMap/Secrets must exist in the namespace for the pods that will consume them prior to the pod being deployed. The optional flag can be used to prevent the pods from not starting if the ConfigMap/Secret is not present.
- Use an admission controller to ensure specific configuration data or to prevent deployments that do not have specific configuration values set. An example would be if you require all production Java workloads to have certain JVM properties set in production environments. There is an alpha API called `PodPresets` that will allow ConfigMaps and secrets to be applied to all pods based on an annotation, without needing to write a custom admission controller.
- If you're using Helm to release applications into your environment, you can use a life cycle hook to ensure the ConfigMap/Secret template is deployed before the Deployment is applied.
- Some applications require their configuration to be applied as a single file such as a JSON or YAML file. ConfigMap/Secrets allows an entire block of raw data by using the `|` symbol, as demonstrated here:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-file
data:
  config: |
    {
      "iotDevice": {
        "name": "remoteValve",
        "username": "CC:22:3D:E3:CE:30",
        "port": 51826,
        "pin": "031-45-154"
      }
    }
```

- If the application uses system environment variables to determine its configuration, you can use the injection of the ConfigMap data to create an environment variable mapping into the pod. There are two main ways to do this: mounting

every key/value pair in the ConfigMap as a series of environment variables into the pod using `envFrom` and then using `configMapRef` or `secretRef`, or assigning individual keys with their respective values using the `configMapKeyRef` or `secretKeyRef`.

- If you're using the `configMapKeyRef` or `secretKeyRef` method, be aware that if the actual key does not exist, this will prevent the pod from starting.
- If you're loading all of the key/value pairs from the ConfigMap/Secret into the pod using `envFrom`, any keys that are considered invalid environment values will be skipped; however, the pod will be allowed to start. The event for the pod will have an event with reason `InvalidVariableNames` and the appropriate message about which key was skipped. The following code is an example of a Deployment with a ConfigMap and Secret reference as an environment variable:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-config
data:
  mysqldb: myappdb1
  user: mysqluser1

apiVersion: v1
kind: Secret
metadata:
  name: mysql-secret
type: Opaque
data:
  rootpassword: YWRtJasdhaW4=
  userpassword: MWYyZDigKJGUyfgKJBmU2N2Rm

apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-db-deploy
spec:
  selector:
    matchLabels:
      app: myapp-db
  template:
    metadata:
      labels:
        app: myapp-db
    spec:
      containers:
      - name: myapp-db-instance
        image: mysql
        resources:
          limits:
            memory: "128Mi"
```

```
            cpu: "500m"
        ports:
        - containerPort: 3306
        env:
          - name: MYSQL_ROOT_PASSWORD
            valueFrom:
              secretKeyRef:
                name: mysql-secret
                key: rootpassword
          - name: MYSQL_PASSWORD
            valueFrom:
              secretKeyRef:
                name: mysql-secret
                key: userpassword
          - name: MYSQL_USER
            valueFrom:
              configMapKeyRef:
                name: mysql-config
                key: user
          - name: MYSQL_DB
            valueFrom:
              configMapKeyRef:
                name: mysql-config
                key: mysqldb
```

- If there is a need to pass command-line arguments to your containers, environment variable data can be sourced using `$(ENV_KEY)` interpolation syntax:

```
[...]
spec:
  containers:
  - name: load-gen
    image: busybox
    command: ["/bin/sh"]
args: ["-c", "while true; do curl $(WEB_UI_URL); sleep 10;done"]
    ports:
    - containerPort: 8080
    env:
    - name: WEB_UI_URL
      valueFrom:
        configMapKeyRef:
          name: load-gen-config
          key: url
```

- When consuming ConfigMap/Secret data as environment variables, it is very important to understand that updates to the data in the ConfigMap/Secret will *not* update in the pod and will require a pod restart either through deleting the pods and letting the ReplicaSet controller create a new pod, or triggering a Deployment update, which will follow the proper application update strategy as declared in the Deployment specification.

- It is easier to assume that all changes to a ConfigMap/Secret require an update to the entire deployment; this ensures that even if you're using environment variables or volumes, the code will take the new configuration data. To make this easier, you can use a CI/CD pipeline to update the `name` property of the ConfigMap/Secret and also update the reference in the deployment, which will then trigger an update through normal Kubernetes update strategies of your deployment. We will explore this in the following example code. If you're using Helm to release your application code into Kubernetes, you can take advantage of an annotation in the Deployment template to check the `sha256` checksum of the ConfigMap/Secret. This triggers Helm to update the Deployment using the `helm upgrade` command when the data within a ConfigMap/Secret is changed:

```
apiVersion: apps/v1
kind: Deployment
[...]
spec:
  template:
    metadata:
      annotations:
        checksum/config: {{ include (print $.Template.BasePath "/config
map.yaml") . | sha256sum }}
[...]
```

### Best practices specific to secrets

Because of the nature of sensitive data of the Secrets API, there are naturally more specific best practices, which are mainly around the security of the data itself:

- The original specification for the Secrets API outlined a pluggable architecture to allow the actual storage of the secret to be configurable based on requirements. Solutions such as HashiCorp Vault, Aqua Security, Twistlock, AWS Secrets Manager, Google Cloud KMS, or Azure Key Vault allow the use of external storage systems for secret data using a higher level of encryption and auditability than what is offered natively in Kubernetes.
- Assign an `imagePullSecrets` to a `serviceaccount` that the pod will use to automatically mount the secret without having to declare it in the `pod.spec`. You can patch the default service account for the namespace of your application and add the `imagePullSecrets` to it directly. This automatically adds it to all pods in the namespace:

```
Create the docker-registry secret first
kubectl create secret docker-registry registryKey --docker-server
myreg.azurecr.io --docker-username myreg --docker-password $up3r$3cr3tP@ssw0rd
--docker-email ignore@dummy.com

patch the default serviceaccount for the namespace you wish to configure
```

```
kubectl patch serviceaccount default -p '{"imagePullSecrets": [{"name":
"registryKey"}]}'
```

- Use CI/CD capabilities to get secrets from a secure vault or encrypted store with a Hardware Security Module (HSM) during the release pipeline. This allows for separation of duties. Security management teams can create and encrypt the secrets, and developers just need to reference the names of the secret expected. This is also the preferred DevOps process to ensure a more dynamic application delivery process.

# RBAC

When working in large, distributed environments, it is very common that some type of security mechanism is needed to prevent unauthorized access to critical systems. There are numerous strategies around how to limit access to resources in computer systems, but the majority all go through the same phases. Using an analogy of a common experience such as flying to a foreign country can help explain the processes that happen in systems like Kubernetes. We can use the common travler's experience with a passport, travel visa, and customs or border guards to show the process:

1. Passport (subject authentication). Usually you need to have a passport issued by some government agency that will offer some sort of verification as to who you are. This would be equivalent to a user account in Kubernetes. Kubernetes relies on an external authority to authenticate users; however, service accounts are a type of account that is managed directly by Kubernetes.
2. Visa or travel policy (authorization). Countries will have formal agreements to accept travelers holding passports from other countries through formal short-term agreements such as visas. The visas will also outline what the visitor may do and for how long they may stay in the visiting country, depending on the specific type of visa. This would be equivalent to authorization in Kubernetes. Kubernetes has different authorization methods, but the most used is RBAC. This allows very granular access to different API capabilities.
3. Border patrol or customs (admission control). When entering a foreign country, usually there is a body of authority that will check the requisite documents, including the passport and visa, and, in many cases, inspect what is being brought into the country to ensure it abides by that country's laws. In Kubernetes this is equivalent to admission controllers. Admission controllers can allow, deny, or change the requests into the API based upon rules and policies that are defined. Kubernetes has many built-in admission controllers such as PodSecurity, ResourceQuota, and ServiceAccount controllers. Kubernetes also allows for dynamic controllers through the use of validating or mutating admission controllers.

The focus of this section is the least understood and the most avoided of these three areas: RBAC. Before we outline some of the best practices, we first must present a primer on Kubernetes RBAC.

## RBAC Primer

The RBAC process in Kubernetes has three main components that need to be defined: the subject, the rule, and the role binding.

### Subjects

The first component is the subject, the item that is actually being checked for access. The subject is usually a user, a service account, or a group. As mentioned earlier, users as well as groups are handled outside of Kubernetes by the authorization module used. We can categorize these as basic authentication, x.509 client certificates, or bearer tokens. The most common implementations use either x.509 client certificates or some type of bearer token using something like an OpenID Connect system such as Azure Active Directory (Azure AD), Salesforce, or Google.

Service accounts in Kubernetes are different than user accounts in that they are namespace bound, internally stored in Kubernetes; they are meant to represent processes, not people, and are managed by native Kubernetes controllers.

### Rules

Simply stated, this is the actual list of actions that can be performed on a specific object (resource) or a group of objects in the API. Verbs align to typical CRUD (Create, Read, Update, and Delete) type operations but with some added capabilities in Kubernetes such as `watch`, `list`, and `exec`. The objects align to the different API components and are grouped together in categories. Pod objects, as an example, are part of the core API and can be referenced with `apiGroup: ""` whereas deployments are under the app API Group. This is the real power of the RBAC process and probably what intimidates and confuses people when creating proper RBAC controls.

### Roles

Roles allow the definition of scope of the rules defined. Kubernetes has two types of roles, `role` and `clusterRole`, the difference being that `role` is specific to a namespace, and `clusterRole` is a cluster-wide role across all namespaces. An example Role definition with namespace scope would be as follows:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-viewer
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

### RoleBindings

The RoleBinding allows a mapping of a subject like a user or group to a specific role. Bindings also have two modes: `roleBinding`, which is specific to a namespace, and `clusterRoleBinding`, which is across the entire cluster. Here's an example RoleBinding with namespace scope:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: noc-helpdesk-view
  namespace: default
subjects:
- kind: User
  name: helpdeskuser@example.com
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role #this must be Role or ClusterRole
  name: pod-viewer # this must match the name of the Role or ClusterRole to
bind to
  apiGroup: rbac.authorization.k8s.io
```

## RBAC Best Practices

RBAC is a critical component of running a secure, dependable, and stable Kubernetes environment. The concepts underlying RBAC can be complex; however, adhering to a few best practices can ease some of the major stumbling blocks:

- Applications that are developed to run in Kubernetes rarely ever need an RBAC role and role binding associated to it. Only if the application code actually interacts directly with the Kubernetes API directly does the application require RBAC configuration.
- If the application does need to directly access the Kubernetes API to perhaps change configuration depending on endpoints being added to a service, or if it needs to list all of the pods in a specific namespace, the best practice is to create a new service account that is then specified in the pod specification. Then, create a role that has the least amount of privileges needed to accomplish its goal.

- Use an OpenID Connect service that enables identity management and, if needed, two-factor authentication. This will allow for a higher level of identity authentication. Map user groups to roles that have the least amount of privileges needed to accomplish the job.
- Along with the aforementioned practice, you should use Just in Time (JIT) access systems to allow site reliability engineers (SREs), operators, and those who might need to have escalated privileges for a short period of time to accomplish a very specific task. Alternatively, these users should have different identities that are more heavily audited for sign-on, and those accounts should have more elevated privileges assigned by the user account or group bound to a role.
- Specific service accounts should be used for CI/CD tools that deploy into your Kubernetes clusters. This ensures for auditability within the cluster and an understanding of who might have deployed or deleted any objects in a cluster.
- If you're using Helm to deploy applications, the default service account is Tiller, deployed to `kube-system`. It is better to deploy Tiller into each namespace with a service account specifically for Tiller that is scoped for that namespace. In the CI/CD tool that calls the Helm install/upgrade command, as a prestep, initialize the Helm client with the service account and the specific namespace for the deployment. The service account name can be the same for each namespace, but the namespace should be specific. It is important to call out that as of this publication, Helm v3 is in alpha state and one of its core principles is that Tiller is no longer needed to run in a cluster. An example Helm Init with a Service account and namespace would look like this:

```
kubectl create namespace myapp-prod

kubectl create serviceaccount tiller --namespace myapp-prod

cat  <<EOF | kubectl apply -f -
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: tiller
  namespace: myapp-prod
rules:
- apiGroups: ["", "batch", "extensions", "apps"]
  resources: ["*"]
  verbs: ["*"]
EOF

cat <<EOF | kubectl apply -f -
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: tiller-binding
  namespace: myapp-prod
```

```
subjects:
- kind: ServiceAccount
  name: tiller
  namespace: myapp-prod
roleRef:
  kind: Role
  name: tiller
  apiGroup: rbac.authorization.k8s.io
  EOF

helm init --service-account=tiller --tiller-namespace=myapp-prod

helm install ./myChart --name myApp --namespace myapp-prod --set global.name-
space=myapp-prod
```

Some public Helm charts do not have value entries for namespace choices to deploy the application components. This might require customization of the Helm chart directly or using an elevated Tiller account that can deploy to any namespace and has rights to create namespaces.

- Limit any applications that require `watch` and `list` on the Secrets API. This basically allows the application or the person who deployed the pod to view the secrets in that namespace. If an application needs to access the Secrets API for specific secrets, limit using `get` on any specific secrets that the application needs to read outside of those that it is directly assigned.

# Summary

Principles for developing applications for cloud native delivery is a topic for another day, but it is universally accepted that strict separation of configuration from code is a key principal for success. With native objects for nonsensitive data, the ConfigMap API, and for sensitive data, the Secrets API, Kubernetes can now manage this process in a declarative approach. As more and more critical data is represented and stored natively in the Kubernetes API, it is critical to secure access to those APIs through proper gated security processes such as RBAC and integrated authentication systems.

As you'll see throughout the rest of this book, these principles permeate every aspect of the proper deployment of services into a Kubernetes platform to build a stable, reliable, secure, and robust system.

# CHAPTER 5
# Continuous Integration, Testing, and Deployment

In this chapter, we look at the key concepts of how to integrate a continuous integration/continuous deployment (CI/CD) pipeline to deliver your applications to Kubernetes. Building a well-integrated pipeline will enable you to deliver applications to production with confidence, so here we look at the methods, tools, and processes to enable CI/CD in your environment. The goal of CI/CD is to have a fully automated process, from a developer checking in code to rolling out the new code to production. You want to avoid manually rolling out updates to your apps deployed to Kubernetes because it can be very error prone. Manually managing application updates in Kubernetes leads to configuration drift and fragile deployment updates, and overall agility delivering an application is lost.

We cover the following topics in this chapter:

- Version control
- CI
- Testing
- Tagging images
- CD
- Deployment strategies
- Testing Deployments
- Chaos testing

We also go through an example CI/CD pipeline, which consists of the following tasks:

- Pushing code changes to the Git repository
- Running a build of the application code
- Running test against the code
- Building a container image on a successful test
- Pushing the container image to a container registry
- Deploying the application to Kubernetes
- Running a test against a deployed application
- Performing rolling upgrades on Deployments

## Version Control

Every CI/CD pipeline starts with version control, which maintains a running history of application and configuration code changes. Git has become the industry standard as a source-control management platform, and every Git repository will contain a *master branch*. A master branch contains your production code. You will have other branches for feature and development work that eventually will also be merged to your master branch. There are many ways to set up a branching strategy, and the setup will be very dependent on the organization structure and separation of duties. We find that including both application code and configuration code, such as a Kubernetes manifest or Helm charts, helps promote good DevOps principles of communication and collaboration. Having both application developers and operation engineers collaborate in a single repository builds confidence in a team to deliver an application to production.

## Continuous Integration

CI is the process of integrating code changes continuously into a version-control repository. Instead of committing large changes less often, you commit smaller changes more often. Each time a code change is committed to the repository, a build is kicked off. This allows you to have a quicker feedback loop into what might have broken the application if problems indeed arise. At this point you might be asking, "Why do I need to know about how the application is built, isn't that the application developer's role?" Traditionally, this might have been the case, but as companies move toward embracing a DevOps culture, the operations team comes closer to the application code and software development workflows.

There are many solutions that provide CI, with Jenkins being one of the more popular tools.

## Testing

The goal of running tests in the pipeline is to quickly provide a feedback loop for code changes that break the build. The language that you're using will determine the testing framework you use. For example, Go applications can use `go test` for running a suite of unit tests against your code base. Having an extensive test suite helps to avoid delivering bad code into your production environment. You'll want to ensure that if tests fail in the pipeline, the build fails after the test suite runs. You don't want to build the container image and push it to a registry if you have failing tests against your code base.

Again, you might be asking, "Isn't creating tests a developer's job?" As you begin automating the delivery of infrastructure and applications to production, you need to think about running automated tests against all of the pieces of the code base. For example, in Chapter 2, we talked about using Helm to package applications for Kubernetes. Helm includes a tool called `helm lint`, which runs a series of tests against a chart to examine any potential issues with the chart provided. There are many different tests that need to be run in an end-to-end pipeline. Some are the developer's responsibility, like unit testing for the application, but others, like smoke testing, will be a joint effort. Testing the code base and its delivery to production is a team effort and needs to be implemented end to end.

## Container Builds

When building your images, you should optimize the size of the image. Having a smaller image decreases the time it takes to pull and deploy the image, and also increases the security of the image. There are multiple ways of optimizing the image size, but some do have trade-offs. The following strategies will help you build the smallest image possible for your application:

*Multistage builds*
: These allow you to remove the dependencies not needed for your applications to run. For example, with Golang, we don't need all the build tools used to build the static binary, so multistage builds allow you in a single Dockerfile to run a build step with the final image containing only the static binary that's needed to run the application.

*Distroless base images*
: These remove all the unneeded binaries and shells from the image. This really reduces the size of the image and increases the security. The trade-off with distroless images is you don't have a shell, so you can't attach a debugger to the image. You might think this is great, but it can be a pain to debug an application. Distroless images contain no package manager, shell, or other typical OS packages, so

you might not have access to the debugging tools you are accustomed to with a typical OS.

*Optimized base images*
: These are images that focus on removing the cruft out of the OS layer and provide a slimmed-down image. For example, Alpine provides a base image that starts at just 10 MB, and it also allows you to attach a local debugger for local development. Other distros also typically offer an optimized base image, such as Debian's Slim image. This might be a good option for you because its optimized images give you capabilities you expect for development while also optimizing for image size and lower security exposure.

Optimizing your images is extremely important and often overlooked by users. You might have reasons due to company standards for OSes that are approved for use in the enterprise, but push back on these so that you can maximize the value of containers.

We have found that companies starting out with Kubernetes tend to be successful with using their current OS but then choose a more optimized image, like Debian Slim. After you mature in operationalizing and developing against a container environment, you'll be comfortable with distroless images.

## Container Image Tagging

Another step in the CI pipeline is to build a Docker image so that you have an image artifact to deploy to an environment. It's important to have an image tagging strategy so that you can easily identify the versioned images you have deployed to your environments. One of the most important things we can't preach enough about is not to use "latest" as an image tag. Using that as an image tag is not a *version* and will lead to not having the ability to identify what code change belongs to the rolled-out image. Every image that is built in the CI pipeline should have a unique tag for the built image.

There are multiple strategies we've found to be effective when tagging images in the CI pipeline. The following strategies allow you to easily identify the code changes and the build with which they are associated:

*BuildID*
: When a CI build kicks off, it has a buildID associated with it. Using this part of the tag allows you to reference which build assembled the image.

*Build System-BuildID*
: This one is the same as BuildID but adds the Build System for users who have multiple build systems.

*Git Hash*
: On new code commits, a Git hash is generated, and using the hash for the tag allows you to easily reference which commit generated the image.

*githash-buildID*
: This allows you to reference both the code commit and the buildID that generated the image. The only caution here is that the tag can be kind of long.

# Continuous Deployment

CD is the process by which changes that have passed successfully through the CI pipeline are deployed to production without human intervention. Containers provide a great advantage for deploying changes into production. Container images become an immutable object that can be promoted through dev and staging and into production. For example, one of the major issues we've always had has been maintaining consistent environments. Almost everyone has experienced a Deployment that works fine in staging, but when it gets promoted to production, it breaks. This is due to having *configuration drift*, with libraries and versioning of components differing in each environment. Kubernetes gives us a declarative way to describe our Deployment objects that can be versioned and deployed in a consistent manner.

One thing to keep in mind is that you need to have a solid CI pipeline set up before focusing on CD. If you don't have a robust set of tests to catch issues early in the pipeline, you'll end up rolling bad code to all your environments.

# Deployment Strategies

Now that we learned the principles of CD, let's take a look at the different rollout strategies that you can use. Kubernetes provides multiple strategies to roll out new versions of your application. And even though it has a built-in mechanism to provide rolling updates, you can also utilize some more advanced strategies. Here, we examine the following strategies to deliver updates to your application:

- Rolling updates
- Blue/green deployments
- Canary deployments

*Rolling updates* are built into Kubernetes and allow you to trigger an update to the currently running application without downtime. For example, if you took your frontend app that is currently running frontend:v1 and updated the Deployment to frontend:v2, Kubernetes would update the replicas in a rolling fashion to frontend:v2. Figure 5-1 depicts a rolling update.

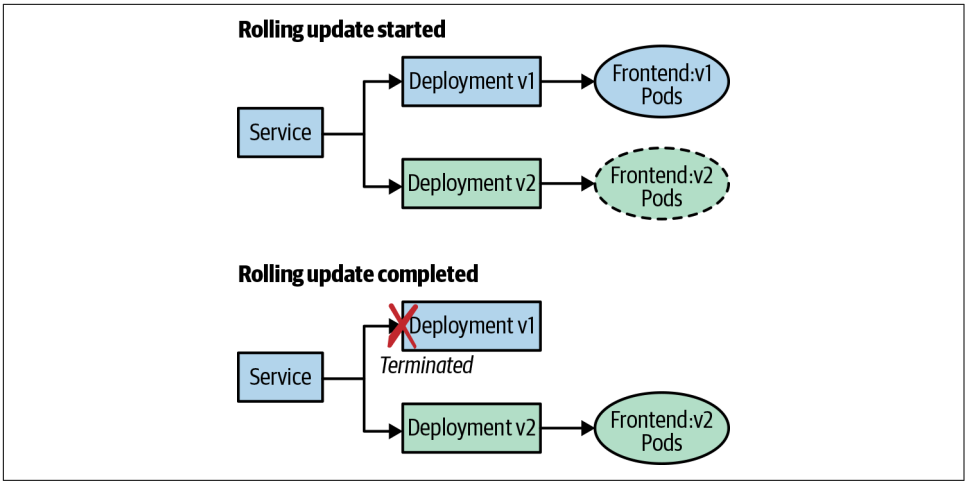



*Figure 5-1. A Kubernetes rolling update*

A Deployment object also lets you configure the maximum amount of replicas to be updated and the maximum unavailable pods during the rollout. The following manifest is an example of how you specify the rolling update strategy:

```
kind: Deployment
apiVersion: v1
metadata:
  name: frontend
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: frontend
        image: brendanburns/frontend:v1
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1 # Maximum amount of replicas to update at one time
      maxUnavailable: 1 # Maximum amount of replicas unavailable during rollout
```

You need to be cautious with rolling updates because using this strategy can cause dropped connections. To deal with this issue, you can utilize *readiness probes* and *preStop* life cycle hooks. The readiness probe ensures that the new version deployed is ready to accept traffic, whereas the preStop hook can ensure that connections are drained on the current deployed application. The life cycle hook is called before the container exits and is synchronous, so it must complete before the final termination signal is given. The following example implements a readiness probe and life cycle hook:

```
kind: Deployment
apiVersion: v1
metadata:
  name: frontend
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: frontend
        image: brendanburns/frontend:v1
        livenessProbe:
          # ...
        readinessProbe:
          httpGet:
            path: /readiness # probe endpoint
            port: 8888
        lifecycle:
          preStop:
            exec:
              command: ["/usr/sbin/nginx","-s","quit"]
  strategy:
    # ...
```

The preStop life cycle hook in this example will gracefully exit NGINX, whereas a SIGTERM conducts a nongraceful, quick exit.

Another concern with rolling updates is that you now have two versions of the application running at the same time during the rollover. Your database schema needs to support both versions of the application. You can also use a feature flag strategy in which your schema indicates the new columns created by the new app version. After the rolling update has completed, the old columns can be removed.

We have also defined a readiness and liveness probe in our Deployment manifest. A readiness probe will ensure that your application is ready to serve traffic before putting it behind the service as an endpoint. The liveness probe ensures that your application is healthy and running, and restarts the pod if it fails its liveness probe. Kubernetes can automatically restart a failed pod only if the pod exits on error. For example, the liveness probe can check its endpoint and restart it if we had a deadlock from which the pod did not exit.

*Blue/green deployments* allow you to release your application in a predictable manner. With blue/green deployments, you control when the traffic is shifted over to the new environment, so it gives you a lot of control over the rollout of a new version of your application. With blue/green deployments, you are required to have the capacity to deploy both the existing and new environment at the same time. These types of deployments have a lot of advantages, such as easily switching back to your previous version of the application. There are some things that you need to consider with this deployment strategy, however:

- Database migrations can become difficult with this deployment option because you need to consider in-flight transactions and schema update compatibility.
- There is the risk of accidental deletion of both environments.
- You need extra capacity for both environments.
- There are coordination issues for hybrid deployments in which legacy apps can't handle the deployment.

Figure 5-2 depicts a blue/green deployment.

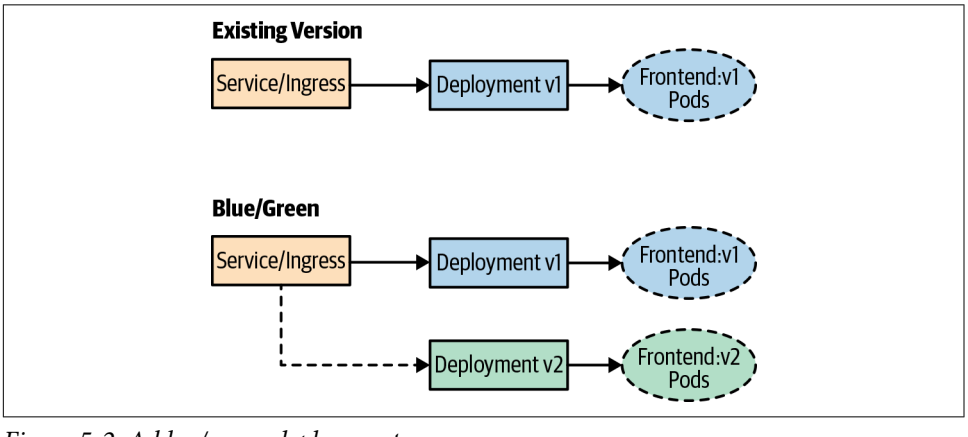



*Figure 5-2. A blue/green deployment*

*Canary deployments* are very similar to blue/green deployments, but they give you much more control over shifting traffic to the new release. Most modern ingress implementations will give you the ability to release a percentage of traffic to a new release, but you can also implement a service mesh technology, like Istio, Linkerd, or HashiCorp Consul, which give you a number of features that help implement this deployment strategy.

Canary deployments allow you to test new features for only a subset of users. For example, you might roll out a new version of an application and only want to test the deployment for 10% of your user base. This allows you to reduce the risk of a bad deployment or broken features to a much smaller subset of users. If there are no errors with the deployment or new features, you can begin shifting a greater percentage of traffic to the new version of the application. There are also some more advanced techniques that you can use with canary deployments in which you release to only a specific region of users or just target only users with a specific profile. These types of releases are often referred to as A/B or dark releases because users are unaware they are testing new feature deployments.

With canary deployments, you have some of the same considerations that you have with blue/green deployments, but there are some additional considerations as well. You must have:

- The ability to shift traffic to a percentage of users
- A firm knowledge of steady state to compare against a new release
- Metrics to understand whether the new release is in a "good" or "bad" state

Figure 5-3 provides an example of a canary deployment.

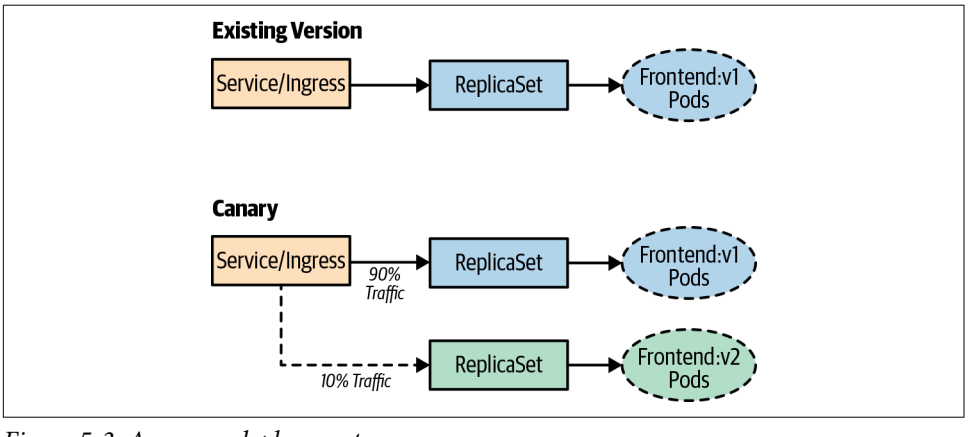



*Figure 5-3. A canary deployment*

Canary releases also suffer from having multiple versions of the application running at the same time. Your database schema needs to support both versions of the application. When using these strategies, you'll need to really focus on how to handle dependent services and having multiple versions running. This includes having strong API contracts and ensuring that your data services support the multiple versions you have deployed at the same time.

# Testing in Production

Testing in production helps you to build confidence in the resiliency, scalability, and UX of your application. This comes with the caveat that *testing in production* doesn't come without challenges and risk, but it's worth the effort to ensure reliability in your systems. There are important aspects you need to address up front when embarking on the implementation. You need to ensure that you have an in-depth observability strategy in place, in which you have the ability to identify the effects of testing in production. Without being able to observe metrics that affect the end users' experience of your applications, you won't have a clear indication of what to focus on when trying

to improve the resiliency of your system. You also need a high degree of automation in place to be able to automatically recover from failures that you inject into your systems.

There are many tools that you'll need to implement to reduce risk and effectively test your systems when they're in production. Some of the tools we have already discussed in this chapter, but there are a few new ones, like distributed tracing, instrumentation, chaos engineering, and traffic shadowing. To recap, here are the tools we have already mentioned:

- Canary deployments
- A/B testing
- Traffic shifting
- Feature flags

Chaos engineering was developed by Netflix. It is the practice of deploying experiments into live production systems to discover weaknesses within those systems. Chaos engineering allows you to learn about the behavior of your system by observing it during a controlled experiment. Following are the steps that you want to implement before doing a "game-day" experiment:

1. Build a hypothesis and learn about your steady state.
2. Have a varying degree of real-world events that can affect the system.
3. Build a control group and experiment to compare to steady state.
4. Perform experiments to form the hypothesis.

It's extremely important that when you're running experiments, you minimize the "blast radius" to ensure that the issues that might arise are minimal. You'll also want to ensure that when you're building experiments, you focus on automating them, given that running experiments can be labor intensive.

By this point, you might be asking, "Why wouldn't I just test in staging?" We find there are some inherent problems when testing in staging, such as the following:

- Nonidentical deployment of resources.
- Configuration drift from production.
- Traffic and user behavior tend to be generated synthetically.
- The number of requests generated don't mimic a real workload.
- Lack of monitoring implemented in staging.
- The data services deployed contain differing data and load than in production.

We can't stress this enough: ensure that you have solid confidence in the monitoring you have in place for production, because this practice tends to fail users who don't have adequate observability of their production systems. Also, starting with smaller experiments to first learn about your experiments and their effects will help build confidence.

# Setting Up a Pipeline and Performing a Chaos Experiment

The first step in the process is to get a GitHub repository forked so that you can have your own repository to use through the chapter. You will need to use the GitHub interface to fork the repository.

## Setting Up CI

Now that you have learned about CI, you will set up a build of the code that we cloned previously.

For this example, we use the hosted *drone.io*. You'll need to sign up for a free account. Log in with your GitHub credentials (this registers your repositories in Drone and allows you to synchronize the repositories). After you're logged in to Drone, select Activate on your forked repository. The first thing that you need to do is add some secrets to your settings so that you can push the app to your Docker Hub registry and also deploy the app to your Kubernetes cluster.

Under your repository in Drone, click Settings and add the following secrets (see Figure 5-4):

- `docker_username`
- `docker_password`
- `kubernetes_server`
- `kubernetes_cert`
- `kubernetes_token`

The Docker username and password will be whatever you used to register on Docker Hub. The following steps show you how to create a Kubernetes service account and certificate and retrieve the token.

For the Kubernetes server, you will need a publicly available Kubernetes API endpoint.

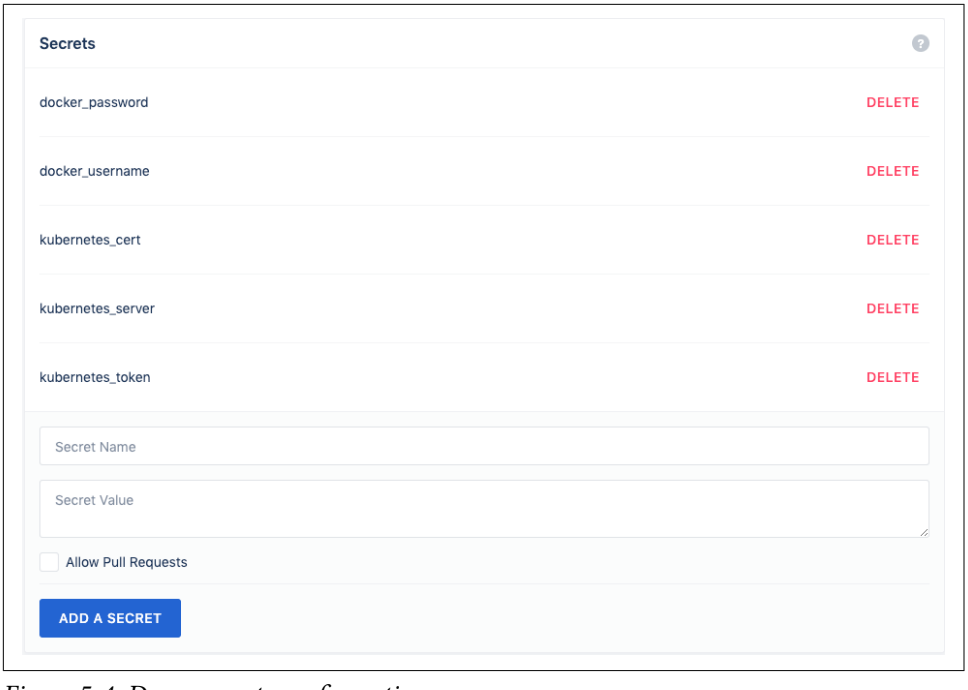



*Figure 5-4. Drone secrets configuration*

You will need cluster-admin privileges on your Kubernetes cluster to perform the steps in this section.

You can retrieve your API endpoint by using the following command:

```
kubectl cluster-info
```

You should see something like the following: Kubernetes master is running at *https://kbp.centralus.azmk8s.io:443*. You'll store this in the `kubernetes_server` secret.

Now let's create a service account that Drone will use to connect to the cluster. Use the following command to create the `serviceaccount`:

```
kubectl create serviceaccount drone
```

Now use the following command to create a `clusterrolebinding` for the `serviceaccount`:

```
kubectl create clusterrolebinding drone-admin \
  --clusterrole=cluster-admin \
  --serviceaccount=default:drone
```

Next, retrieve your `serviceaccount` token:

```
TOKENNAME=`kubectl -n default get serviceaccount/drone -o json-
path='{.secrets[0].name}'`
TOKEN=`kubectl -n default get secret $TOKENNAME -o jsonpath='{.data.token}' |
base64 -d`
echo $TOKEN
```

You'll want to store the output of the token in the `kubernetes_token` secret.

You will also need the user certificate to authenticate to the cluster, so use the following command and paste the `ca.crt` for the `kubernetes_cert` secret:

```
kubectl get secret $TOKENNAME -o yaml | grep 'ca.crt:'
```

Now, build your app in a Drone pipeline and then push it to Docker Hub.

The first step is the *build step*, which will build your Node.js frontend. Drone utilizes container images to run its steps, which gives you a lot of flexibility in what you can do with it. For the build step, use a Node.js image from Docker Hub:

```
pipeline:
  build:
    image: node
    commands:
      - cd frontend
      - npm i redis --save
```

When the build completes, you'll want to test it, so we include a *test step*, which will run `npm` against the newly built app:

```
test:
    image: node
    commands:
      - cd frontend
      - npm i redis --save
      - npm test
```

Now that you have successfully built and tested your app, you next move on to a *publish step* to create a Docker image of the app and push it to Docker Hub.

In the *.drone.yml* file, make the following code change:

```
repo: <your-registry>/frontend

publish:
    image: plugins/docker
    dockerfile: ./frontend/Dockerfile
    context: ./frontend
    repo: dstrebel/frontend
    tags: [latest, v2]
    secrets: [ docker_username, docker_password ]
```

After the Docker build step finishes, it will push the image to your Docker registry.

## Setting Up CD

For the deployment step in your pipeline, you will push your application to your Kubernetes cluster. You will use the deployment manifest that is under the frontend app folder in your repository:

```
kubectl:
    image: dstrebel/drone-kubectl-helm
    secrets: [ kubernetes_server, kubernetes_cert, kubernetes_token ]
    kubectl: "apply -f ./frontend/deployment.yaml"
```

After the pipeline finishes its deployment, you will see the pods running in your cluster. Run the following command to confirm that the pods are running:

```
kubectl get pods
```

You can also add a test step that will retrieve the status of the deployment by adding the following step in your Drone pipeline:

```
  test-deployment:
    image: dstrebel/drone-kubectl-helm
    secrets: [ kubernetes_server, kubernetes_cert, kubernetes_token ]
    kubectl: "get deployment frontend"
```

## Performing a Rolling Upgrade

Let's demonstrate a rolling upgrade by changing a line in the frontend code. In the *server.js* file, change the following line and then commit the change:

```
console.log('api server is running.');
```

You will see the deployment rolling out and rolling updates happening to the existing pods. After the rolling update finishes, you'll have the new version of the application deployed.

## A Simple Chaos Experiment

There are a variety of tools in the Kubernetes ecosystem that can help with performing chaos experiments in your environment. They range from sophisticated hosted Chaos as a Service solutions to basic chaos experiment tools that kill pods in your environment. Following are some of the tools with which we've seen users have success:

*Gremlin*
: Hosted chaos service that provides advanced features for running chaos experiments

*PowerfulSeal*
: Open source project that provides advanced chaos scenarios

*Chaos Toolkit*
: Open source project with a mission to provide a free, open, and community-driven toolkit and API to all the various forms of chaos engineering tools

*KubeMonkey*
: Open source tool that provides basic resiliency testing for pods in your cluster

Let's set up a quick chaos experiment to test the resiliency of your application by automatically terminating pods. For this experiment, we'll use Chaos Toolkit:

```
pip install -U chaostoolkit

pip install chaostoolkit-kubernetes

export FRONTEND_URL="http://$(kubectl get svc frontend -o jsonpath="{.sta
tus.loadBalancer.ingress[*].ip}"):8080/api/"

chaos run experiment.json
```

# Best Practices for CI/CD

Your CI/CD pipeline won't be perfect on day one, but consider some of the following best practices to iteratively improve on the pipeline:

- With CI, focus on automation and providing quick builds. Optimizing the build speed will provide developers quick feedback if their changes have broken the build.
- Focus on providing reliable tests in your pipeline. This will give developers rapid feedback on issues with their code. The faster the feedback loop to developers, the more productive they'll become in their workflow.
- When deciding on CI/CD tools, ensure that the tools allow you to define the pipeline as code. This will allow you to version-control the pipeline with your application code.
- Ensure that you optimize your images so that you can reduce the size of the image and also reduce the attack surface when running the image in production. Multistage Docker builds allow you to remove packages not needed for the application to run. For example, you might need Maven to build the application, but you don't need it for the actual running image.
- Avoid using "latest" as an image tag, and utilize a *tag* that can be referenced back to the buildID or Git commit.
- If you are new to CD, utilize Kubernetes rolling upgrades to start out. They are easy to use and will get you comfortable with deployment. As you become more comfortable and confident with CD, look at utilizing blue/green and canary deployment strategies.

- With CD, ensure that you test how client connections and database schema upgrades are handled in your application.
- Testing in production will help you build reliability into your application, and ensure that you have good monitoring in place. With testing in production, also start at a small scale and limit the blast radius of the experiment.

## Summary

In this chapter, we discussed the stages of building a CI/CD pipeline for your applications, which let you reliably deliver software with confidence. CI/CD pipelines help reduce risk and increase throughput of delivering applications to Kubernetes. We also discussed the different deployment strategies that can be utilized for delivering applications.

CHAPTER 6

# Versioning, Releases, and Rollouts

One of the main complaints of traditional monolithic applications is that over time they begin to grow too large and unwieldy to properly upgrade, version, or modify at the speed the business requires. Many can argue that this is one of the main critical factors that led to more Agile development practices and the advent of microservice architectures. Being able to quickly iterate on new code, solve new problems, or fix hidden problems before they become major issues, as well as the promise of zero-downtime upgrades, are all goals that development teams strive for in this ever-changing internet economy world. Practically, these issues can be solved with proper processes and procedures in place, no matter the type of system, but this usually comes at a much higher cost of both technology and human capital to maintain.

The adoption of containers as the runtime for application code allows for the isolation and composability that was helpful in designing systems that could get close, but still required a high level of human automation or system management to maintain at a dependable level over large system footprints. As the system grew, more brittleness was introduced, and systems engineers began to build complex automation processes to deliver on complex release, upgrade, and failure detection mechanisms. Service orchestrators such as Apache Mesos, HashiCorp Nomad, and even specialized container-based orchestrators such as Kubernetes and Docker Swarm evolved this into more primitive components to their runtime. Now, systems engineers can solve more complex system problems as the table stakes have been elevated to include the versioning, release, and deployment of applications into the system.

## Versioning

This section is not meant to be a primer on software versioning and the history behind it; there are countless articles and computer science course books on the subject. The main thing is to pick a pattern and stick with it. The majority of software

companies and developers have agreed that some form of *semantic versioning* is the most useful, especially in a microservice architecture in which a team that writes a certain microservice will depend on the API compatibility of other microservices that make up the system.

For those new to semantic versioning, the basics are that it follows a three-part version number in a pattern of *major version*, *minor version*, and *patch*, usually expressed in a *dot notation* such as 1(major).2(minor).3(patch). The patch signifies an incremental release that includes a bug fix or very minor change that has no API changes. The minor version signifies updates that might have new API changes but is backward compatible with the previous version. This is a key attribute for developers working with other microservices they might not be involved in developing. Knowing that I have my service written to communicate with version 1.4.7 of another microservice that has been recently upgraded to 1.5.7 should signify that I might not need to change my code unless I want to take advantage of any new API features. The major version is a breaking change increment to the code. In most cases, the API is no longer compatible between major versions of the same code. There are many slight modifications to this process, including a "4" version to indicate the stage of the software in its development life cycle, such as 1.4.7.0 for alpha code, and 1.4.7.3 for release. The most important thing is that there is consistency across the system.

# Releases

In truth, Kubernetes does not really have a release controller, so there is no native concept of a release. This is usually added to a Deployment `metadata.labels` specification and/or in the `pod.spec.template.metadata.label` specification. When to include either is very important, and based on how CD is used to update changes to deployments, it can have varied effects. When Helm for Kubernetes was introduced, one of its main concepts was the notion of a release to differentiate the running instance of the same Helm chart in a cluster. This concept is easily reproducible without Helm; however, Helm natively keeps track of releases and their history, so many CD tools integrate Helm into their pipelines to be the actual release service. Again, the key here is consistency in how versioning is used and where it is surfaced in the system state of the cluster.

Release names can be quite useful if there is institutional agreement as to the definition of certain names. Often labels such as `stable` or `canary` are used, which helps to also give some kind of operational control when tools such as service meshes are added to make fine-grained routing decisions. Large organizations that drive numerous changes for different audiences will also adopt a ring architecture that can also be denoted such as `ring-0`, `ring-1`, and so on.

This topic requires a little side trip into the specifics of labels in the Kubernetes declarative model. Labels themselves are very much free form and can be any

key/value pair that follows the syntactical rules of the API. The key is not really the content but how each controller handles labels, changes to labels, and selector matching of labels. Jobs, Deployments, ReplicaSets, and DaemonSets support selector-based matching of pods via labels through direct mapping or set-based expressions. It is important to understand that label selectors are immutable after they are created, which means if you add a new selector and the pod's labels have a corresponding match, a new ReplicaSet is made, not an upgrade to an existing ReplicaSet. This becomes very important to understand when dealing with rollouts, which we discuss next.

# Rollouts

Prior to the Deployment controller being introduced in Kubernetes, the only mechanism that existed to control how applications were rolled out by the Kubernetes controller process was using the command-line interface (CLI) command `kubectl rolling-update` on the specific `replicaController` that was to be updated. This was very difficult for declarative CD models because this was not part of the state of the original manifest. One had to carefully ensure that manifests were updated correctly, versioned properly so as to not accidentally roll the system back, and archived when no longer needed. The Deployment controller added the ability to automate this update process using a specific strategy and then allowing the system to read the declarative new state based on changes to the `spec.template` of the deployment. This last fact is often misunderstood by early users of Kubernetes and causes frustration when they change a label in the Deployment metadata fields, reapply a manifest, and no update has been triggered. The Deployment controller is able to determine changes to the specification and will take action to update the Deployment based on a strategy that is defined by the specification. Kubernetes deployments support two strategies, `rollingUpdate` and `recreate`, the former being the default.

If a rolling update is specified, the deployment will create a new ReplicaSet to scale to the number of required replicas, and the old ReplicaSet will scale down to zero based on specific values for `maxUnavailble` and `maxSurge`. In essence, those two values will prevent Kubernetes from removing older pods until a sufficient number of newer pods have come online, and will not create new pods until a certain number of old pods have been removed. The nice thing is that the Deployment controller will keep a history of the updates, and through the CLI, you can roll back deployments to previous versions.

The `recreate` strategy is a valid strategy for certain workloads that can handle a complete outage of the pods in a ReplicaSet with little to no degradation of service. In this strategy the Deployment controller will create a new ReplicaSet with the new configuration and will delete the prior ReplicaSet before bringing the new pods online. Services that sit behind queue-based systems are an example of a service that could handle

this type of disruption, because messages will queue while waiting for the new pods to come online, and message processing will resume as soon as the new pods come online.

## Putting It All Together

Within a single service deployment, a few key areas are affected by versioning, release, and rollout management. Let's examine an example deployment and then break down the specific areas of interest as they relate to best practices:

```
# Web Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gb-web-deploy
  labels:
    app: guest-book
    appver: 1.6.9
    environment: production
    release: guest-book-stable
    release number: 34e57f01
spec:
  strategy:
    type: rollingUpdate
    rollingUpdate:
      maxUnavailbale: 3
      maxSurge: 2
  selector:
    matchLabels:
      app: gb-web
      ver: 1.5.8
    matchExpressions:
      - {key: environment, operator: In, values: [production]}
  template:
    metadata:
      labels:
        app: gb-web
        ver: 1.5.8
        environment: production
    spec:
      containers:
      - name: gb-web-cont
        image: evillgenius/gb-web:v1.5.5
        env:
        - name: GB_DB_HOST
          value: gb-mysql
        - name: GB_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-pass
              key: password
```

```
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 80
---
# DB Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gb-mysql
  labels:
    app: guest-book
    appver: 1.6.9
    environment: production
    release: guest-book-stable
    release number: 34e57f01
spec:
  selector:
    matchLabels:
      app: gb-db
      tier: backend
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: gb-db
        tier: backend
        ver: 1.5.9
        environment: production
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        env:
        - name: MYSQL_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-pass
              key: password
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
```

```
---
# DB Backup Job
apiVersion: batch/v1
kind: Job
metadata:
  name: db-backup
  labels:
    app: guest-book
    appver: 1.6.9
    environment: production
    release: guest-book-stable
    release number: 34e57f01
  annotations:
    "helm.sh/hook": pre-upgrade
    "helm.sh/hook": pre-delete
    "helm.sh/hook": pre-rollback
    "helm.sh/hook-delete-policy": hook-succeeded
spec:
  template:
    metadata:
      labels:
        app: gb-db-backup
        tier: backend
        ver: 1.6.1
        environment: production
    spec:
      containers:
      - name: mysqldump
        image: evillgenius/mysqldump:v1
        env:
        - name: DB_NAME
          value: gbdb1
        - name: GB_DB_HOST
          value: gb-mysql
        - name: GB_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-pass
              key: password
        volumeMounts:
          - mountPath: /mysqldump
            name: mysqldump
      volumes:
        - name: mysqldump
          hostPath:
            path: /home/bck/mysqldump
      restartPolicy: Never
  backoffLimit: 3
```

Upon first inspection, things might look a little off. How can a deployment have a version tag and the container image the deployment uses have a different version tag? What will happen if one changes and the other does not? What does release mean in

this example, and what effect on the system will that have if it changes? If a certain label is changed, when will it trigger an update to my deployment? We can find the answers to these questions by looking at some of the best practices for versioning, releases, and rollouts.

## Best Practices for Versioning, Releases, and Rollouts

Effective CI/CD and the ability to offer reduced or zero downtime deployments are both dependent on using consistent practices for versioning and release management. The best practices noted below can help to define consistent parameters that can assist DevOps teams in delivering smooth software deployments:

- Use semantic versioning for the application in its entirety that differs from the version of the containers and the version of the pods deployment that make up the entire application. This allows for independent life cycles of the containers that make up the application and the application as a whole. This can become quite confusing at first, but if a principled hierarchical approach is taken to when one changes the other, you can easily track it. In the previous example, the container itself is currently on `v1.5.5`; however, the pod specification is a `1.5.8`, which could mean that changes were made to the pod specification, such as new ConfigMaps, additional secrets, or updated replica values, but the specific container used has not changed its version. The application itself, the entire guestbook application and all of its services, is at `1.6.9`, which could mean that operations made changes along the way that were beyond just this specific service, such as other services that make up the entire application.
- Use a release and release version/number label in your deployment metadata to track releases from CI/CD pipelines. The release name and release number should coordinate with the actual release in the CI/CD tool records. This allows for traceability through the CI/CD process into the cluster and allows for easier rollback identification. In the previous example, the release number comes directly from the release ID of the CD pipeline that created the manifest.
- If Helm is being used to package services for deployment into Kubernetes, take special care to bundle together those services that need to be rolled back or upgraded together into the same Helm chart. Helm allows for easy rollback of all components of the application to bring the state back to what it was before the upgrade. Because Helm actually processes the templates and all of the Helm directives before passing a flattened YAML configuration, the use of life cycle hooks allows for proper ordering of the application of specific templates. Operators can use proper Helm life cycle hooks to ensure that upgrades and rollback will happen correctly. The previous example for the `Job` specification uses Helm life cycle hooks to ensure that the template runs a backup of the database before a rollback, upgrade, or delete of the Helm release. It also ensures that the `Job` is

deleted after the job is run successfully, which, until the TTL Controller comes out of alpha in Kubernetes, would require manual cleanup.

- Agree on a release nomenclature that makes sense for the operational tempo of the organization. Simple `stable`, `canary`, and `alpha` states are quite adequate for most situations.

## Summary

Kubernetes has allowed for more complex, Agile development processes to be adopted within companies large and small. The ability to automate much of the complex processes that would usually require large amounts of human and technical capital has now been democratized to allow for even startups to take advantage of this cloud pattern with relative ease. The true declarative nature of Kubernetes really shines when planning the proper use of labels and using native Kubernetes controller capabilities. By properly identifying operational and development states within the declarative properties of the applications deployed into Kubernetes, organizations can tie in tooling and automation to more easily manage the complex processes of upgrades, rollouts, and rollbacks of capabilities.

CHAPTER 7

# Worldwide Application Distribution and Staging

So far throughout this book, we have seen a number of different practices for building, developing, and deploying applications, but there is a whole different set of concerns when it comes to deploying and managing an application with a worldwide footprint.

There are many different reasons why an application might need to scale to a global deployment. The first and most obvious one is simply scale. It might be that your application is so successful or mission critical that it simply needs to be deployed around the world in order to provide the capacity needed for its users. Examples of such applications include a worldwide API gateway for a public cloud provider, a large-scale IoT product with a worldwide footprint, a highly successful social network, and more.

Although there are relatively few of us who will build out systems that require worldwide scale, many more applications require a worldwide footprint for latency. Even with containers and Kubernetes there is no getting around the speed of light, and thus to minimize latency to our applications, it is sometimes necessary to distribute our applications around the world to minimize the distance to our users.

Finally, an even more common reason for global distribution is locality. Either for reasons of bandwidth (e.g., a remote sensing platform) or data privacy (geographic restrictions), it is sometimes necessary to deploy an application in specific locations for the application to be possible or successful.

In all of these cases, your application is no longer simply present in a small handful of production clusters. Instead it is distributed across tens to hundreds of different geographic locations, and the management of these locations, as well as the demands of

rolling out a globally reliable service, is a significant challenge. This chapter covers approaches and practices for doing this successfully.

## Distributing Your Image

Before you can even consider running your application around the world, you need to have that image available in clusters located around the globe. The first thing to consider is whether your image registry has automatic geo-replication. Many image registries provided by cloud providers will automatically distribute your image around the world and resolve a request for that image to the storage location nearest to the cluster from which you are pulling the image. Many clouds enable you to decide where you want to replicate the image; for example, you might know of locations where you are not going to be present. An example of such a registry is the Microsoft Azure container registry, but others provide similar services. If you use a cloud-provided registry that supports geo-replication, distributing your image around the world is simple. You push the image into the registry, select the regions for geo-distribution, and the registry takes care of the rest.

If you are not using a cloud registry, or your provider does not support automatic geo-distribution of images, you will need to solve that problem yourself. One option is to use a registry located in a specific location. There are several concerns about such an approach. Image pull latency often dictates the speed with which you can launch a container in a cluster. This in turn can determine how quickly you can respond to a machine failure, given that generally in the case of a machine failure, you will need to pull the container image down to a new machine.

Another concern about a single registry is that it can be a single point of failure. If the registry is located in a single region or a single datacenter, it's possible that the registry could go offline due to a large-scale incident in that datacenter. If your registry goes offline, your CI/CD pipeline will stop working, and you'll be unable to deploy new code. This obviously has a significant impact on both developer productivity and application operations. Additionally, a single registry can be much more expensive because you will be using significant bandwidth each time you launch a new container, and even though container images are generally fairly small, the bandwidth can add up. Despite these negatives, a single registry solution can be the appropriate answer for small-scale applications running in only a few global regions. It certainly is simpler to set up than full-scale image replication.

If you cannot use cloud-provided geo-replication and you need to replicate your image, you are on your own to craft a solution for image replication. To implement such a service, you have two options. The first is to use geographic names for each image registry (e.g., `us.my-registry.io`, `eu.my-registry.io`, etc.). The advantage of this approach is that it is simple to set up and manage. Each registry is entirely independent, and you can simply push to all registries at the end of your CI/CD pipeline.

The downside is that each cluster will require a slightly different configuration to pull the image from the nearest geographic location. However, given that you likely will have geographic differences in your application configurations anyway, this downside is relatively easy to manage and likely already present in your environment.

## Parameterizing Your Deployment

When you have replicated your image everywhere, you need to parameterize your deployments for different global locations. Whenever you are deploying to a variety of different regions, there are bound to be differences in the configuration of your application in the different regions. For example, if you don't have a geo-replicated registry, you might need to tweak the image name for different regions, but even if you have a geo-replicated image, it's likely that different geographic locations will present different load on your application, and thus the size (e.g., the number of replicas) as well as other configuration can be different between regions. Managing this complexity in a manner that doesn't incur undue toil is key to successfully managing a worldwide application.

The first thing to consider is how to organize your different configurations on disk. A common way to achieve this is by using a different directory for each global region. Given these directories, it might be tempting to simply copy the same configurations into each directory, but doing this is guaranteed to lead to drift and changes between configurations in which some regions are modified and other regions are forgotten. Instead, using a template-based approach is the best idea so that most of the configuration is retained in a single template that is shared by all regions, and then parameters are applied to that template to produce the region-specific templates. Helm is a commonly used tool for this sort of templating (for details, see Chapter 2).

## Load-Balancing Traffic Around the World

Now that your application is running around the world, the next step is to determine how to direct traffic to the application. In general, you want to take advantage of geographic proximity to ensure low-latency access to your service. But you also want to failover across geographic regions in case of an outage or any other source of service failure. Correctly setting up the balancing of traffic to your various regional deployments is key to the establishment of both a performant and reliable system.

Let's begin with the assumption that you have a single hostname that you want to serve as your service. For example, *myapp.myco.com*. One initial decision that you need to make is whether you want to use the Domain Name System (DNS) protocol to implement load balancing across your regional endpoints. If you use DNS for load balancing, the IP address that is returned when a user makes a DNS query to

*myapp.myco.com* is based on both the location of the user accessing your service as well as the current availability of your service.

# Reliably Rolling Out Software Around the World

After you have templatized your application so that you have proper configurations for each region, the next important problem is how to deploy these configurations around the world. It might be tempting to simultaneously deploy your application worldwide so that you can efficiently and quickly iterate your application, but this, although Agile, is an approach that can easily leave you with a global outage. Instead, for most production applications, a more carefully staged approach to rolling out your software around the world is more appropriate. When combined with things like global load balancing, these approaches can maintain high availability even in the face of major application failures.

Overall, when approaching the problem of a global rollout, the goal is to roll out software as quickly as possible, while simultaneously detecting issues quickly—ideally before they affect any other users. Let's assume that by the time you are performing a global rollout, your application has already passed basic functional and load testing. Before a particular image (or images) is certified for a global rollout, it should have gone through enough testing that you believe the application is operating correctly. It iss important to note that this *does not* mean that your application *is* operating correctly. Though testing catches many problems, in the real world, application problems are often first noticed when they are rolled out to production traffic. This is because the true nature of production traffic is often difficult to simulate with perfect fidelity. For example, you might test with only English language inputs, whereas in the real world, you see input from a variety of languages. Or your set of test inputs is not comprehensive for the real-world data your application ingests. Of course, any time that you do see a failure in production that wasn't caught by testing, it is a strong indicator that you need to extend and expand your testing. Nonetheless, it is still true that many problems are caught during a production rollout.

With this in mind, each region that you roll out to is an opportunity to discover a new problem. And, because the region is a production region, it is also a potential outage to which you will need to react. These factors combine to set the stage for how you should approach regional rollouts.

## Pre-Rollout Validation

Before you even consider rolling out a particular version of your software around the world, it's critically important to validate that software in some sort of synthetic testing environment. If you have your CD pipeline set up correctly, all code prior to a particular release build will have undergone some form of unit testing, and possibly limited integration testing. However, even with this testing in place, it's important to

consider two other sorts of tests for a release before it begins its journey through the release pipeline. The first is complete integration testing. This means that you assemble the entirety of your stack into a full-scale deployment of your application but without any real-world traffic. This complete stack generally will include either a copy of your production data or simulated data on the same size and scale as your true production data. If in the real world, the data in your application is 500 GB, it's critical that in preproduction testing your dataset is roughly the same size (and possibly even literally the same dataset).

Generally speaking, this is the most difficult part of setting up a complete integration test environment. Often, production data is really present only in production, and generating a synthetic dataset of the same size and scale is quite difficult. Because of this complexity, setting up a realistic integration testing dataset is a great example of a task that it pays to do early on in the development of an application. If you set up a synthetic copy of your dataset early, when the dataset itself is quite small, your integration test data grows gradually at the same pace as your production data. This is generally significantly more manageable than if you attempt to duplicate your production data when you are already at scale.

Sadly, many people don't realize that they need a copy of their data until they are already at a large scale and the task is difficult. In such cases it might be possible to deploy a read/write-deflecting layer in front of your production data store. Obviously, you don't want your integration tests writing to production data, but it is often possible to set up a proxy in front of your production data store that reads from production but stores writes in a side table that is also consulted on subsequent reads.

Regardless of how you manage to set up your integration testing environment, the goal is the same: to validate that your application behaves as expected when given a series of test inputs and interactions. There are a variety of ways to define and execute these tests—from the most manual, a worksheet of tests and human effort (not recommended because it is fairly error prone), through tests that simulate browsers and user interactions, like clicks and so forth. In the middle are tests that probe RESTful APIs but don't necessarily test the web UI built on top of those APIs. Regardless of how you define your integration tests, the goal should be the same: an automated test suite that validates the correct behavior of your application in response to a complete set of real-world inputs. For simple applications it may be possible to perform this validation in premerge testing, but for most large-scale real-world applications, a complete integration environment is required.

Integration testing will validate the correct operation of your application, but you should also load-test the application. It is one thing to demonstrate that the application behaves correctly, it is quite another to demonstrate that it stands up to real-world load. In any reasonably high-scale system, a significant regression in performance—for example, a 20% increase in request latency—has a significant

impact on the UX of the application and, in addition to frustrating users, can cause an application to completely fail. Thus, it is critical to ensure that such performance regressions do not happen in production.

Like integration testing, identifying the correct way to load-test an application can be a complex proposition; after all, it requires that you generate a load similar to production traffic but in a synthetic and reproduceable way. One of the easiest ways to do this is to simply replay the logs of traffic from a real-world production system. Doing this can be a great way to perform a load-test whose characteristics match what your application will experience when deployed. However, using replay isn't always foolproof. For example, if your logs are old, and your application or dataset has changed, it's possible that the performance on old, replayed logs will be different that the performance on fresh traffic. Additionally, if you have real-world dependencies that you haven't mocked, it's possible that the old traffic will be invalid when sent over to the dependencies (e.g., the data might no longer exist).

Because of these challenges, many systems, even critical systems, are developed for a long time without a load test. Like modeling your production data, this is a clear example of something that is easier to maintain if you start earlier. If you build a load-test when your application has only a handful of dependencies, and improve and iterate the load-test as you adapt your application, you will have a far easier time than if you attempt to retrofit load-testing onto an existing large-scale application.

Assuming that you have crafted a load test, the next question is the metrics to watch when load-testing your application. The obvious ones are requests per second and request latency because those are clearly the user-facing metrics.

When measuring latency, it's important to realize that this is actually a distribution, and you need to measure both the mean latency as well as the outlier percentiles (like the 90th and 99th percentile) since they represent the "worst" UX of your application. Problems with very long latencies can be hidden if you just look at the averages, but if 10% of your users are having a bad time, it can have a significant impact on the success of your product.

In addition, it's worth looking at the resource usage (CPU, memory, network, disk) of the application under load test. Though these metrics do not directly contribute to the UX, large changes in resource usage for your application should be identified and understood in preproduction testing. If your application is suddenly consuming twice as much memory, it's something you will want to investigate, even if you pass your load test, because eventually such significant resource growth will affect the quality and availability of your application. Depending on the circumstances, you might continue bringing a release to production, but at the same time, you need to understand why the resource footprint of your application is changing.

## Canary Region

When your application appears to be operating correctly, the first step should be a *canary region*. A canary region is a deployment that receives real-world traffic from people and teams who want to validate your release. These can be internal teams that depend on your service, or they might be external customers who are using your service. Canaries exist to give a team some early warning about changes that you are about to roll out that might break them. No matter how good your integration and load testing, it's always possible that a bug will slip through that isn't covered by your tests, but is critical to some user or customer. In such cases, it is much better to catch these issues in a space where everyone using or deploying against the service understands that there is a higher probability of failure. This is what the canary region is.

Canaries must be treated as a production region in terms of monitoring, scale, features, and so on. However, because it is the first stop on the release process, it is also the location most likely to see a broken release. This is OK; in fact it is precisely the point. Your customers will knowingly use a canary for lower-risk use cases (e.g., development or internal users) so that they can get an early indication of any breaking changes that you might be rolling out as part of a release.

Because the goal of a canary is to get early feedback on a release, it is a good idea to leave the release in the canary region for a few days. This enables a broad collection of customers to access it before you move on to additional regions. The need for this length of time is that sometimes a bug is probabilistic (e.g., 1% of requests) or it manifests only in an edge case that takes some time to present itself. It might not even be severe enough to trigger automated alerts, but there might be a problem in business logic that is visible only via customer interactions.

## Identifying Region Types

When you begin thinking about rolling out your software across the world, it's important to think about the different characteristics of your different regions. After you begin rolling out software to production regions, you need to run it through integration testing as well as initial canary testing. This means that any issues you find will be issues that did not manifest in either of these settings. Think about your different regions. Do some get more traffic than others? Are some accessed in a different way? An example of a difference might be that in the developing world, traffic is more likely to come from mobile web browsers. Thus, a region that is geographically close to more developing countries might have significantly more mobile traffic than your test or canary regions.

Another example might be input language. Regions in non-English speaking areas of the world might send more Unicode characters that could manifest bugs in string or character handling. If you are building an API-driven service, some APIs might be more popular in some regions versus others. All of these things are examples of

differences that might be present in your application and might be different than your canary traffic. Each of these differences is a possible source of a production incident. Build a table of different characteristics that you think are important. Identifying these characteristics will help you plan your global rollout.

## Constructing a Global Rollout

Having identified the characteristics of your regions, you want to identify a plan for rolling out to all regions. Obviously, you want to minimize the impact of a production outage, so a great first region to start with is a region that looks mostly like your canary and has light user traffic. Such a region is very unlikely to have problems, but if they do occur, the impact is also smaller because the region receives less traffic.

With a successful rollout to the first production region, you need to decide how long to wait before moving on to the next region. The reason for waiting is not to artificially delay your release; rather, it's to wait long enough for a fire to send up smoke. This time-to-smoke period is a measure of generally how long it takes between a rollout completing and your monitoring seeing some sign of a problem. Clearly if a rollout contains a problem, the minute the rollout completes, the problem is present in your infrastructure. But even though it is present, it can take some time to manifest. For example, a memory leak might take an hour or more before the impact of the leaked memory is clearly discernible in monitoring or is affecting users. The time-to-smoke is the probability distribution that indicates how long you should wait in order to have a strong probability that your release is operating correctly. Generally speaking, a decent rule of thumb is doubling the average time it takes for a problem to manifest.

If, over the past six months, each outage took an average of an hour to show up, waiting two hours between regional rollouts gives you a decent probability that your release is successful. If you want to derive richer (and more meaningful) statistics based on the history of your application, you can estimate this time-to-smoke even more closely.

Having successfully rolled out to a canary-like, low-traffic region, it's time to roll out to a canary-like, high-traffic region. This is a region where the input data looks like that in your canary, but it receives a large volume of traffic. Because you successfully rolled out to a similar looking region with lower traffic, at this point the only thing you are testing is your application's ability to scale. If you safely perform this rollout, you can have strong confidence in the quality of your release.

After you have rolled out to a high-traffic region receiving canary-like traffic, you should follow the same pattern for other potential differences in traffic. For example, you might roll out to a low-traffic region in Asia or Europe next. At this point, it might be tempting to accelerate your rollout, but it is critically important to roll out only to a single region that represents any significant change in either input or load to

your release. After you are confident that you have tested all of the potential variability in the production input to your application, you then can start parallizing the release to speed it up with strong confidence that it is operating correctly and your rollout can complete successfully.

## When Something Goes Wrong

So far, we have seen the pieces that go into setting up a worldwide rollout for your software system, and we have seen the ways that you can structure this rollout to minimize the chances that something goes wrong. But what do you do when something actually does go wrong? All emergency responders know that in the heat and panic of a crisis, your brain is significantly stressed and it is much more difficult to remember even the simplest processes. Add to this pressure the knowledge that when an outage happens, everyone in the company from the CEO down is going to be feverishly waiting for the "all clear" signal, and you can see how easy it is to make a mistake under this pressure. Additionally, in such circumstances, a simple mistake, like forgetting a particular step in a recovery process, can make a bad situation an order of magnitude worse.

For all of these reasons, it is critical that you are capable of responding quickly, calmly, and correctly when a problem happens with a rollout. To ensure that everything necessary is done, and done in the correct order, it pays to have a clear checklist of tasks organized in the order in which they are to be executed as well as the expected output for each step. Write down every step, no matter how obvious it might seem. In the heat of the moment, even the most obvious and easy steps can be the ones that are forgotten and accidentally skipped.

The way that other first responders ensure a correct response in a high-stress situation is to practice that response without the stress of the emergency. The same practice applies to all the activities that you might take in response to a problem with your rollout. You begin by identifying all of the steps needed to respond to an issue and perform a rollback. Ideally, the first response is to "stop the bleeding," to move user traffic away from the impacted region(s) and into a region where the rollout hasn't happened and your system is operating correctly. This is the first thing you should practice. Can you successfully direct traffic away from a region? How long does it take?

The first time you attempt to move traffic using a DNS-based traffic load balancer, you will realize just how long and in how many ways our computers cache DNS entries. It can take nearly a day to fully drain traffic away from a region using a DNS-based traffic shaper. Regardless of how your first attempt to drain traffic goes, take notes. What worked well? What went poorly? Given this data, set a goal for how long a traffic drain should take in terms of time to drain a percentage of traffic, for example, being able to drain 99% of traffic in less than 10 minutes. Keep practicing until

you can achieve that goal. You might need to make architectural changes to make this possible. You might need to add automation so that humans aren't cutting and pasting commands. Regardless of necessary changes, practice will ensure that you are more capable at responding to an incident and that you will learn where your system design needs to be improved.

The same sort of practice applies to every action that you might take on your system. Practice a full-scale data recovery. Practice a global rollback of your system to a previous version. Set goals for the length of time it should take. Note any places where you made mistakes, and add validation and automation to eliminate the possibility of mistakes. Achieving your incident reaction goals in practice gives you confidence that you will be able to respond correctly in a real incident. But just like every emergency responder continues to train and learn, you too need to set up a regular cadence of practice to ensure that everyone on a team stays well versed in the proper responses and (perhaps more important) that your responses stay up to date as your system changes.

## Worldwide Rollout Best Practices

- Distribute each image around the world. A successful rollout depends on the release bits (binaries, images, etc.) being nearby to where they will be used. This also ensures reliability of the rollout in the presence of networking slowdowns or irregularities. Geographic distribution should be a part of your automated release pipeline for guaranteed consistency.
- Shift as much of your testing as possible to the left by having as much extensive integration and replay testing of your application as possible. You want to start a rollout only with a release that you strongly believe to be correct.
- Begin a release in a canary region, which is a preproduction environment in which other teams or large customers can validate *their* use of your service before you begin a larger-scale rollout.
- Identify different characteristics of the regions where you are rolling out. Each difference can be one that causes a failure and a full or partial outage. Try to roll out to low-risk regions first.
- Document and practice your response to any problem or process (e.g., a rollback) that you might encounter. Trying to remember what to do in the heat of the moment is a recipe for forgetting something and making a bad problem worse.

## Summary

It might seem unlikely today, but most of us will end up running a worldwide scale system sometime during our careers. This chapter described how you can gradually build and iterate your system to be a truly global design. It also discussed how you can set up your rollout to ensure minimal downtime of the system while it is being updated. Finally, we covered setting up and practicing the processes and procedures necessary to react when (note that we didn't say "if") something goes wrong.

CHAPTER 8

# Resource Management

In this chapter, we focus on the best practices for managing and optimizing Kubernetes resources. We discuss workload scheduling, cluster management, pod resource management, namespace management, and scaling applications. We also dive into some of the advanced scheduling techniques that Kubernetes provides through affinity, anti-affinity, taints, tolerations, and nodeSelectors.

We show you how to implement resource limits, resource requests, pod Quality of Service, `PodDisruptionBudgets`, `LimitRangers`, and anti-affinity policies.

## Kubernetes Scheduler

The Kubernetes scheduler is one of the main components that is hosted in the control plane. The scheduler allows Kubernetes to make placement decisions for pods deployed to the cluster. It deals with optimization of resources based on constraints of the cluster as well as user-specified constraints. It uses a scoring algorithm that is based on predicates and priorities.

### Predicates

The first function Kubernetes uses to make a scheduling decision is the predicate function, which determines what nodes the pods can be scheduled on. It implies a hard constraint, so it returns a value of true or false. An example would be when a pod requests 4 GB of memory and a node cannot satisfy this requirement. The node would return a false value and would be removed from viable nodes for the pod to be scheduled to. Another example would be if the node is set to unschedulable; it would then be removed from the scheduling decision.

The scheduler checks the predicates based on order of restrictiveness and complexity. As of this writing, the following are the predicates that the scheduler checks for:

```
CheckNodeConditionPred,
CheckNodeUnschedulablePred,
GeneralPred,
HostNamePred,
PodFitsHostPortsPred,
MatchNodeSelectorPred,
PodFitsResourcesPred,
NoDiskConflictPred,
PodToleratesNodeTaintsPred,
PodToleratesNodeNoExecuteTaintsPred,
CheckNodeLabelPresencePred,
CheckServiceAffinityPred,
MaxEBSVolumeCountPred,
MaxGCEPDVolumeCountPred,
MaxCSIVolumeCountPred,
MaxAzureDiskVolumeCountPred,
MaxCinderVolumeCountPred,
CheckVolumeBindingPred,
NoVolumeZoneConflictPred,
CheckNodeMemoryPressurePred,
CheckNodePIDPressurePred,
CheckNodeDiskPressurePred,
MatchInterPodAffinityPred
```

## Priorities

Whereas predicates indicate a true or false value and dismiss a node for scheduling, the priority value ranks all of the valid nodes based on a relative value. The following priorities are scored for nodes:

```
EqualPriority
MostRequestedPriority
RequestedToCapacityRatioPriority
SelectorSpreadPriority
ServiceSpreadingPriority
InterPodAffinityPriority
LeastRequestedPriority
BalancedResourceAllocation
NodePreferAvoidPodsPriority
NodeAffinityPriority
TaintTolerationPriority
ImageLocalityPriority
ResourceLimitsPriority
```

The scores will be added, and then a node is given its final score to indicate its priority. For example, if a pod requires 600 millicores and there are two nodes, one with 900 millicores available and one with 1,800 millicores, the node with 1,800 millicores available will have a higher priority.

If nodes are returned with the same priority, the scheduler will use a `selectHost()` function, which selects a node in a round-robin fashion.

# Advanced Scheduling Techniques

For most cases, Kubernetes does a good job of optimally scheduling pods for you. It takes into account pods that are placed only on nodes that have sufficient resources. It also tries to spread pods from the same ReplicaSet across nodes to increase availability and will balance resource utilization. When this is not good enough, Kubernetes gives you the flexibility to influence how resources are scheduled. For example, you might want to schedule pods across availability zones to mitigate a zonal failure causing downtime to your application. You might also want to colocate pods to a specific host for performance benefits.

## Pod Affinity and Anti-Affinity

Pod affinity and anti-affinity let you set rules to place pods relative to other pods. These rules allow you to modify the scheduling behavior and override the scheduler's placement decisions.

For example, an anti-affinity rule would allow you to spread pods from a ReplicaSet across multiple datacenter zones. It does this by utilizing keylabels set on the pods. Setting the key/value pairs instructs the scheduler to schedule the pods on the same node (affinity) or prevent the pods from scheduling on the same nodes (anti-affinity).

Following is an example of setting a pod anti-affinity rule:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  selector:
    matchLabels:
      app: frontend
  replicas: 4
  template:
    metadata:
      labels:
        app: frontend
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - frontend
            topologyKey: "kubernetes.io/hostname"
      containers:
```

```
      - name: nginx
        image: nginx:alpine
```

This manifest of an NGINX deployment has four replicas and the selector label `app=frontend`. The deployment has a PodAntiAffinity stanza configured that will ensure that the scheduler does not colocate replicas on a single node. This ensures that if a node fails, there are still enough replicas of NGINX to serve data from its cache.

## nodeSelector

A nodeSelector is the easiest way to schedule pods to a particular node. It uses label selectors with key/value pairs to make the scheduling decision. For example, you might want to schedule pods to a specific node that has specialized hardware, such as a GPU. You might ask, "Can't I do this with a node taint?" The answer is, yes, you can. The difference is that you use a nodeSelector when you want to *request* a GPU-enabled node, whereas a taint *reserves* a node for only GPU workloads. You can use both node taints and nodeSelectors together to reserve the nodes for only GPU workloads, and use the nodeSelector to automatically select a node with a GPU.

Following is an example of labeling a node and using a nodeSelector in the pod specification:

```
kubectl label node <node_name> disktype=ssd
```

Now, let's create a pod specification with a nodeSelector key/value of `disktype: ssd`:

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
  labels:
    env: prod
spec:
  containers:
  - name: frontend
    image: nginx:alpine
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

Using the nodeSelector schedules the pod to only nodes that have the label `disktype=ssd`:

## Taints and Tolerations

*Taints* are used on nodes to repel pods from being scheduled on them. But isn't that what anti-affinity is for? Yes, but taints take a different approach than pod anti-affinity and serve a different use case. For example, you might have pods that require

a specific performance profile, and you do not want to schedule any other pods to the specific node. Taints work in conjunction with *tolerations*, which allow you to override tainted nodes. The combination of the two gives you fine-grained control over anti-affinity rules.

In general, you will use taints and tolerations for the following use cases:

- Specialized node hardware
- Dedicated node resources
- Avoiding degraded nodes

There are multiple taint types that affect scheduling and running containers:

*NoSchedule*
: A hard taint that prevents scheduling on the node

*PreferNoSchedule*
: Schedules only if pods cannot be scheduled on other nodes

*NoExecute*
: Evicts already-running pods on the node

*NodeCondition*
: Taints a node if it meets a specific condition

Figure 8-1 shows an example of a node that is tainted with `gpu=true:NoSchedule`. Pod Spec 1 has a toleration key with `gpu`, so it will be scheduled to the tainted node. Pod Spec 2 has a toleration key of `no-gpu`, so it will not be scheduled to the node.

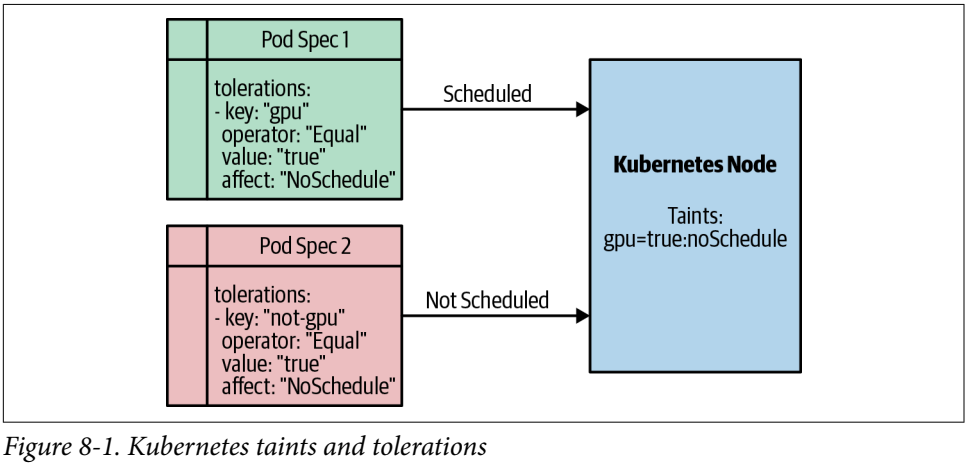



*Figure 8-1. Kubernetes taints and tolerations*

When a pod cannot be scheduled due to tainted nodes, you'll see an error message like the following:

```
Warning:  FailedScheduling  10s (x10 over 2m)  default-scheduler  0/2 nodes are
available: 2 node(s) had taints that the pod did not tolerate.
```

Now that we've seen how we can manually add taints to affect scheduling, there is also the powerful concept of *taint-based eviction*, which allows the eviction of running pods. For example, if a node becomes unhealthy due to a bad disk drive, the taint-based eviction can reschedule the pods on the host to another healthy node in the cluster.

# Pod Resource Management

One of the most important aspects of managing applications in Kubernetes is appropriately managing pod resources. Managing pod resources consists of managing CPU and memory to optimize the overall utilization of your Kubernetes cluster. You can manage these resources at the container level and at the namespace level. There are other resources, such as network and storage, but Kubernetes doesn't yet have a way to set requests and limits for those resources.

For the scheduler to optimize resources and make intelligent placement decisions, it needs to understand the requirements of an application. As an example, if a container (application) needs a minimum of 2 GB to perform, we need to define this in our pod specification, so the scheduler knows that the container requires 2 GB of memory on the host to which it schedules the container.

## Resource Request

A Kubernetes resource *request* defines that a container requires *X* amount of CPU or memory to be scheduled. If you were to specify in the pod specification that a container requires 8 GB for its resource request and all your nodes have 7.5 GB of memory, the pod would not be scheduled. If the pod is not able to be scheduled, it will go into a *pending* state until the required resources are available.

So let's take a look at how this works in our cluster.

To determine the available free resource in your cluster, use `kubectl top`:

```
kubectl top nodes
```

The output should look like this (the memory size might be different for your cluster):

```
NAME                       CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
aks-nodepool1-14849087-0   524m         27%    7500Mi          33%
aks-nodepool1-14849087-1   468m         24%    3505Mi          27%
aks-nodepool1-14849087-2   406m         21%    3051Mi          24%
aks-nodepool1-14849087-3   441m         22%    2812Mi          22%
```

As this example shows, the largest amount of memory available to a host is 7,500 Mi, so let's schedule a pod that requests 8,000 Mi of memory:

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-request
spec:
  containers:
  - name: memory-request
    image: polinux/stress
    resources:
      requests:
        memory: "8000Mi"
```

Notice that the pod will stay pending, and if you look at the events on the pods, you'll see that no nodes are avalaible to schedule the pods:

```
kubectl describe pods memory-request
```

The output of the event should look like this:

```
Events:
  Type     Reason            Age                From               Message
  Warning  FailedScheduling  27s (x2 over 27s)  default-scheduler  0/3 nodes
are available: 3 Insufficient memory.
```

## Resource Limits and Pod Quality of Service

Kubernetes resource *limits* define the maximum CPU or memory that a pod is given. When you specify limits for CPU and memory, each takes a different action when it reaches the specified limit. With CPU limits, the container is throttled from using more than its specified limit. With memory limits, the pod is restarted if it reaches its limit. The pod might be restarted on the same host or a different host within the cluster.

Specifying limits for containers is a good practice to ensure that applications are allotted their fair share of resources within the cluster:

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo
  namespace: cpu-example
spec:
  containers:
  - name: frontend
    image: nginx:alpine
    resources:
      limits:
        cpu: "1"
```

```
      requests:
        cpu: "0.5"
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo
  namespace: qos-example
spec:
  containers:
  - name: qos-demo-ctr
    image: nginx:alpine
    resources:
      limits:
        memory: "200Mi"
        cpu: "700m"
      requests:
        memory: "200Mi"
        cpu: "700m"
```

When a pod is created, it's assigned one of the following Quality of Service (QoS) classes:

- Guaranteed
- Burstable
- Best effort

The pod is assigned a QoS of *guaranteed* when CPU and memory both have request and limits that match. A *burstable* QoS is when the limits are set higher than the request, meaning that the container is guaranteed its request, but it can also burst to the limit set for the container. A pod is assigned *best effort* when no request or limits are set for the containers in the pod.

Figure 8-2 depicts how QoS is assigned to pods.

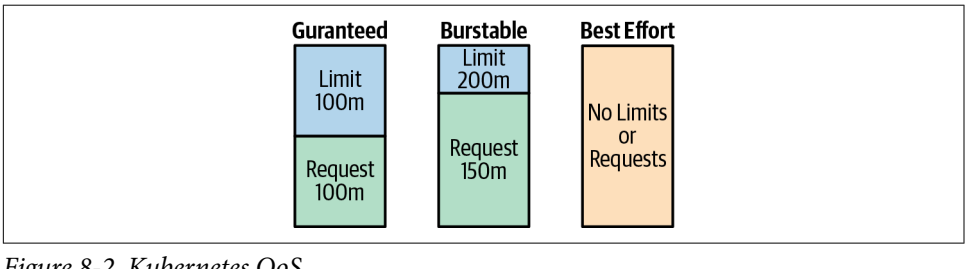



*Figure 8-2. Kubernetes QoS*

With guaranteed QoS, if you have multiple containers in your pod, you'll need to have memory request and limits set for each container, and you'll also need CPU request and limits set for each container. If the request and limits are not set for all containers, it will not be assigned guaranteed QoS.

## PodDisruptionBudgets

At some point in time, Kubernetes might need to *evict* pods from a host. There are two types of evictions: *voluntary* and *involuntary* disruptions. Involuntary disruptions can be caused by hardware failure, network partitions, kernel panics, or a node being out of resources. Voluntary evictions can be caused by performing maintenance on the cluster, the Cluster Autoscaler deallocating nodes, or updating pod templates. To minimize the impact to your application, you can set a `PodDisruptionBudget` to ensure uptime of the application when pods need to be evicted. A `PodDisruptionBudget` allows you to set a policy on the minimum available and maximum unavailable pods during voluntary eviction events. An example of a voluntary eviction would be when draining a node to perform maintenance on the node.

For example, you might specify that no more than 20% of pods belonging to your application can be down at a given time. You could also specify this policy in terms of *X* number of replicas that must always be available.

### Minimum available

In the following example, we set a `PodDisruptionBudget` to handle a minimum available to 5 for app: front-end.

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: frontend-pdb
spec:
  minAvailable: 5
  selector:
    matchLabels:
      app: frontend
```

In this example, the `PodDisruptionBudget` specifies that for the frontend app there must always be five replica pods available at any given time. In this scenario, an eviction can evict as many pods as it wants, as long as five are available.

### Maximum unavailable

In the next example, we set a `PodDisruptionBudget` to handle a maximum unavailable to 10 replicas for the frontend app:

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: frontend-pdb
spec:
  maxUnavailable: 20%
  selector:
    matchLabels:
      app: frontend
```

In this example, the `PodDisruptionBudget` specifies that no more than 20% of replica pods can be unavailable at any given time. In this scenario, an eviction can evict a maximum of 20% of pods during a voluntary disruption.

It's essential that when designing your Kubernetes cluster you think about the sizing of the cluster resources so that you can handle a number of failed nodes. For example, if you have a four-node cluster and one node fails, you will be losing a quarter of your cluster capacity.

When specifying a pod disruption budget as a percentage, it might not correlate to a specific number of pods. For example, if your application has seven pods and you specify `maxAvailable` to `50%`, it's not clear whether that is three or four pods. In this case, Kubernetes rounds up to the closest integer, so the `maxAvailable` would be four pods.

## Managing Resources by Using Namespaces

*Namespaces* in Kubernetes give you a nice logical separation of resources deployed to a cluster. This allows you to set resource quotas per namespace, Role-Based Access Control (RBAC) per namespace, and also network policies per namespace. It gives you soft multitenancy features, so you can separate out workloads in a cluster without dedicating specific infrastructure to a team or application. This allows you to get the most out of your cluster resource while also maintaining a logical form of separation.

For example, you could create a namespace per team and give each team a quota on the number of resources that it can utilize, such as CPU and memory.

When designing how you want to configure a namespace, you should think about how you want to control access to a specific set of applications. If you have multiple teams that will be using a single cluster, it is typically best to allocate a namespace to each team. If the cluster is dedicated to only one team, it might make sense to allocate a namespace for each service deployed to the cluster. There's no single solution to this; your team organization and responsibilities will drive the design.

After deploying a Kubernetes cluster, you'll see the following namespaces in your cluster:

`kube-system`
: Kubernetes internal components are deployed here, such as `coredns`, `kube-proxy`, and `metrics-server`.

`default`
: This is the default namespace that is used when you don't specify a namespace in the resource object.

`kube-public`
: Used for anonymous and unauthenticated content, and reserved for system usage.

You'll want to avoid using the default namespace because it can make it really easy to make mistakes when managing resources within your cluster.

When working with namespaces, you need to use the `-namespace` flag, or `-n` for short, when working with `kubectl`:

```
kubectl create ns team-1

kubectl get pods --namespace team-1
```

You can also set your `kubectl` context to a specific namespace, which is useful so that you don't need to add the `-namespace` flag with every command. You can set your namespace context by using the following command:

```
kubectl config set-context my-context --namespace=team-1
```

When dealing with multiple namespaces and clusters, it can be a pain to set different namespaces and cluster context. We've found that using kubens and kubectx can help make it easy to switch between these different namespaces and contexts.

## ResourceQuota

When multiple teams or applications share a single cluster, it's important to set up `ResourceQuotas` on your namespaces. `ResourceQuotas` allow you to divvy up the cluster in logical units so that no single namespace can consume more than its share of resources in the cluster. The following resources can have a quota set for them:

- Compute resources
  - `requests.cpu`: Sum of CPU requests cannot exceed this amount
  - `limits.cpu`: Sum of CPU limits cannot exceed this amount
  - `requests.memory`: Sum of memory requests cannot exceed this amount
  - `limit.memory`: Sum of memory limits cannot exceed this amount

- Storage resources
  - `requests.storage`: Sum of storage requests cannot exceed this value
  - `persistentvolumeclaims`: The total number of PersistentVolume claims that can exist in the namespace
  - `storageclass.request`: Volume claims associated with the specified storage-class cannot exceed this value
  - `storageclass.pvc`: The total number of PersistentVolume claims that can exist in the namespace
- Object count quotas (only an example set)
  - count/pvc
  - count/services
  - count/deployments
  - count/replicasets

As you can see from this list, Kubernetes gives you fine-grained control over how you carve up resource quotas per namespace. This allows you to more efficiently operate resource usage in a multitenant cluster.

Let's see how these quotas actually work by setting up a quota on a namespace. Apply the following YAML file to the `team-1` namespace:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-demo
  namespace: team-1
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
    persistentvolumeclaims: "5"
    requests.storage: "10Gi
```

```
kubectl apply quota.yaml -n team-1
```

This example sets quotas for CPU, memory, and storage on the `team-1` namespace.

Now let's try to deploy an application to see how the resource quotas affect the deployment:

```
kubectl run nginx-quotatest --image=nginx --restart=Never --replicas=1 --
port=80 --requests='cpu=500m,memory=4Gi' --limits='cpu=500m,memory=4Gi' -n
team-1
```

This deployment will fail with the following error due to the memory quota exceeding `2Gi` of memory:

```
Error from server (Forbidden): pods "nginx-quotatest" is forbidden: exceeded
quota: mem-cpu-demo
```

As this example demonstrates, setting resource quotas can let you deny deployment of resources based on policies you set for the namespace.

## LimitRange

We've discussed setting `request` and `limits` at the container level, but what happens if the user forgets to set these in the pod specification? Kubernetes provides an admission controller that allows you to automatically set these when there are none indicated in the specification.

First, create a namespace to work with quotas and `LimitRanges`:

```
kubectl create ns team-1
```

Apply a `LimitRange` to the namespace to apply `defaultRequest` in `limits`:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: team-1-limit-range
spec:
  limits:
  - default:
      memory: 512Mi
    defaultRequest:
      memory: 256Mi
    type: Container
```

Save this to *limitranger.yaml* and then run `kubectl apply`:

```
kubectl apply -f limitranger.yaml -n team-1
```

Verify that the `LimitRange` applies default limits and requests:

```
 kubectl run team-1-pod --image=nginx -n team-1
```

Next, let's describe the pod to see what requests and limits were set on it:

```
kubectl describe pod team-1-pod -n team-1
```

You should see the following requests and limits set on the pod specification:

```
Limits:
      memory:  512Mi
    Requests:
      memory:  256Mi
```

It's important to use `LimitRange` when using `ResourceQuotas`, because if no request or limits are set in the specification, the deployment will be rejected.

## Cluster Scaling

One of the first decisions you need to make when deploying a cluster is the instance size you'll want to use within your cluster. This becomes more of an art than science, especially when you're mixing workloads in a single cluster. You'll first want to identify what a good starting point is for the cluster; aiming for a good balance of CPU and memory is one option. After you've decided on a sensible size for the cluster, you can use a couple of Kubernetes core primitives to manage the scaling of your cluster.

### Manual scaling

Kubernetes makes it easy to scale your cluster, especially if you're using tools like Kops or a managed Kubernetes offering. Scaling your cluster manually is typically just choosing a new number of nodes, and the service will add the new nodes to your cluster.

These tools also allow you to create node pools, which allows you to add new instance types to an already running cluster. This becomes very useful when running mixed workloads within a single cluster. For example, one workload might be more CPU driven, whereas the other workloads might be memory-driven applications. Node pools allow you to mix multiple instance types within a single cluster.

But perhaps you don't want to manually do this and want it to autoscale. There are things that you need to take into consideration with cluster autoscaling, and we have found that most users are better off starting with just manually scaling their nodes proactively when resources are needed. If your workloads are highly variable, cluster autoscaling can be very useful.

### Cluster autoscaling

Kubernetes provides a Cluster Autoscaler add-on that allows you to set the minimum nodes available to a cluster and also the maximum number of nodes to which your cluster can scale. The Cluster Autoscaler bases its scale decision on when a pod goes pending. For example, if the Kubernetes scheduler tries to schedule a pod with a memory request of 4,000 Mib and the cluster has only 2,000 Mib available, the pod will go into a pending state. After the pod is pending, the Cluster Autoscaler will add a node to the cluster. As soon as the new node is added to the cluster, the pending pod is scheduled to the node. The downside of the Cluster Autoscaler is that a new node is added only before a pod goes pending, so your workload may end up waiting for a new node to come online when it is scheduled. As of Kubernetes v1.15, the Cluster Autoscaler doesn't support scaling based on custom metrics.

The Cluster Autoscaler can also reduce the size of the cluster after resources are no longer needed. When the resources are no longer needed, it will drain the node and reschedule the pods to new nodes in the cluster. You'll want to use a `PodDisruptionBudget` to ensure that you don't negatively affect your application when it performs its drain operation to remove the node from the cluster.

## Application Scaling

Kubernetes provides multiple ways to scale applications in your cluster. You can scale an application by manually changing the number of replicas within a deployment. You can also change the ReplicaSet or replication controller, but we don't recommend managing your applications through those implementations. Manual scaling is perfectly fine for workloads that are static or when you know the times that the workload spikes, but for workloads that experience sudden spikes or workloads that are not static, manual scaling is not ideal for the application. Happily, Kubernetes also provides a Horizontal Pod Autoscaler (HPA) to automatically scale workloads for you.

Let's first take a look at how you can manually scale a deployment by applying the following Deployment manifest:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 3
  template:
    metadata:
      name: frontend
      labels:
        app: frontend
    spec:
      containers:
      - image: nginx:alpine
        name: frontend
        resources:
          requests:
            cpu: 100m
```

This example deploys three replicas of our frontend service. We then can scale this deployment by using the `kubectl scale` command:

```
kubectl scale deployment frontend --replicas 5
```

This results in five replicas of our frontend service. This is great, but let's look at how we can add some intelligence and automatically scale the application based on metrics.

## Scaling with HPA

The Kubernetes HPA allows you to scale your deployments based on CPU, memory, or custom metrics. It performs a watch on the deployment and pulls metrics from the Kubernetes `metrics-server`. It also allows you to set the minimum and maximum number of pods available. For example, you can define an HPA policy that sets the minimum number of pods to 3 and the maximum number of pods to 10, and it scales when the deployment reaches 80% CPU usage. Setting the minimum and maximum is critical because you don't want the HPA to scale the replicas to an infinite amount due to an application bug or issue.

The HPA has the following default setting for sync metrics, upscaling, and downscaling replicas:

`horizontal-pod-autoscaler-sync-period`
: Default of 30 seconds for syncing metrics

`horizontal-pod-autoscaler-upscale-delay`
: Default of three minutes between two upscale operations

`horizontal-pod-autoscaler-downscale-delay`
: Default of five minutes between two downscale operations

You can change the defaults by using their relative flags, but you need to be careful when doing so. If your workload is extremely variable, it's worth playing around with the settings to optimize them for your specific use case.

Let's go ahead and set up an HPA policy for the frontend application that you deployed in the previous exercise.

First, expose the deployment on port 80:

```
kubectl expose deployment frontend --port 80
```

Next, set the autoscale policy:

```
kubectl autoscale deployment frontend --cpu-percent=50 --min=1 --max=10
```

This sets the policy to scale your app from a minimum of 1 replica to a maximum of 10 replicas and will invoke the scale operation when the CPU load reaches 50%.

Let's generate some load so that we can see the deployment autoscale:

```
kubectl run -i --tty load-generator --image=busybox /bin/sh

Hit enter for command prompt
while true; do wget -q -O- http://frontend.default.svc.cluster.local; done

kubectl get hpa
```

You might need to wait a few minutes to see the replicas scale up automatically.

To learn more about the internal details of the autoscaling algorithm, check out the design proposal.

## HPA with Custom Metrics

In Chapter 4, we introduced the role that the metrics server plays in monitoring our systems in Kubernetes. With the Metrics Server API, we can also support scaling our applications with custom metrics. The Custom Metrics API and Metrics Aggregator allows third-party providers to plug in and extend the metrics, and HPA can then scale based on these external metrics. For example, instead of just basic CPU and memory metrics, you could scale based on a metric you're collecting on an external storage queue. By utilizing custom metrics for autoscaling, you have the ability to scale application-specific metrics or external service metrics.

## Vertical Pod Autoscaler

The Vertical Pod Autoscaler (VPA) differs from the HPA in that it doesn't scale replicas; instead, it automatically scales requests. Earlier in the chapter, we talked about setting requests on our pods and how that guarantees *X* amount of resources for a given container. The VPA frees you from manually adjusting these requests, and automatically scales up and scales down pod requests for you. For workloads that can't scale out due to their architecture, this works well for automatically scaling the resources. For example, a MySQL database doesn't scale the same way as a stateless web frontend. With MySQL, you might want to set the Master nodes to automatically scale up based on workload.

The VPA is more complex than the HPA, and it consists of three components:

`Recommender`
: Monitors the current and past resource consumption, and provides recommended values for the container's CPU and memory requests

`Updater`
: Checks which of the pods have the correct resources set, and if they don't, kills them so that they can be re-created by their controllers with the updated requests

`Admission Plugin`
: Sets the correct resource requests on new pods

As of Kubernetes v1.15, the VPA is not recommended for production deployments.

## Resource Management Best Practices

- Utilize pod anti-affinity to spread workloads across multiple availability zones to ensure high availability for your application.
- If you're using specialized hardware, such as GPU-enabled nodes, ensure that only workloads that need GPUs are scheduled to those nodes by utilizing taints.
- Use `NodeCondition` taints to proactively avoid failing or degraded nodes.
- Apply nodeSelectors to your pod specifications to schedule pods to specialized hardware that you have deployed in the cluster.
- Before going to production, experiment with different node sizes to find a good mix of cost and performance for node types.
- If you're deploying a mix of workloads with different performance characteristics, utilize node pools to have mixed node types in a single cluster.
- Ensure that you set memory and CPU limits for all pods deployed to your cluster.
- Utilize `ResourceQuota`s to ensure that multiple teams or applications are alotted their fair share of resources in the cluster.
- Implement `LimitRange` to set default limits and requests for pod specifications that don't set limits or requests.
- Start with manual cluster scaling until you understand your workload profiles on Kubernetes. You can use autoscaling, but it comes with additional considerations around node spin-up time and cluster scale down.
- Use the HPA for workloads that are variable and that have unexpected spikes in their usage.

## Summary

In this chapter, we discussed how you can optimally manage Kubernetes and application resources. Kubernetes provides many built-in features to manage resources that you can use to maintain a reliable, highly utilized, and efficient cluster. Cluster and pod sizing can be difficult at first, but through monitoring your applications in production you can discover ways to optimize your resources.

CHAPTER 9

# Networking, Network Security, and Service Mesh

Kubernetes is effectively a manager of distributed systems across a cluster of connected systems. This immediately puts critical importance on how the connected systems communicate with one another, and networking is the key to this. Understanding how Kubernetes facilitates communication among the distributed services it manages is important for the effective application of interservice communication.

This chapter focuses on the principles that Kubernetes places on the network and best practices around applying these concepts in different situations. With any discussion of networking, security is usually brought along for the ride. The traditional models of network security boundaries being controlled at the network layer are not absent in this new world of distributed systems in Kubernetes, but how they are implemented and the capabilities offered change slightly. Kubernetes brings along a native API for network security policies that will sound eerily similar to firewall rules of old.

The last section of this chapter delves into the new and scary world of service meshes. The term "scary" is used in jest, but it is quite the Wild West when it comes to service mesh technology in Kubernetes.

## Kubernetes Network Principles

Understanding how Kubernetes uses the underlying network to facilitate communication among services is critical to understanding how to effectively plan application architectures. Usually, networking topics start to give most people major headaches. We are going to keep this rather simple because this is more of a best practice guidance than a lesson on container networking. Luckily for us, Kubernetes has laid down some rules of the road for networking that help to give us a start. The rules outline

how communication is expected to behave between different components. Let's take a closer look at each of these rules:

*Container-to-container communication in the same pod*
: All containers in the same pod share the same network space. This effectively allows localhost communication between the containers. It also means that containers in the same pod need to expose different ports. This is done using the power of Linux namespaces and Docker networking to allow these containers to be on the same local network through the use of a paused container in every pod that does nothing but host the networking for the pod. Figure 9-1 shows how Container A can communicate directly with Container B using localhost and the port number that the container is listening on.

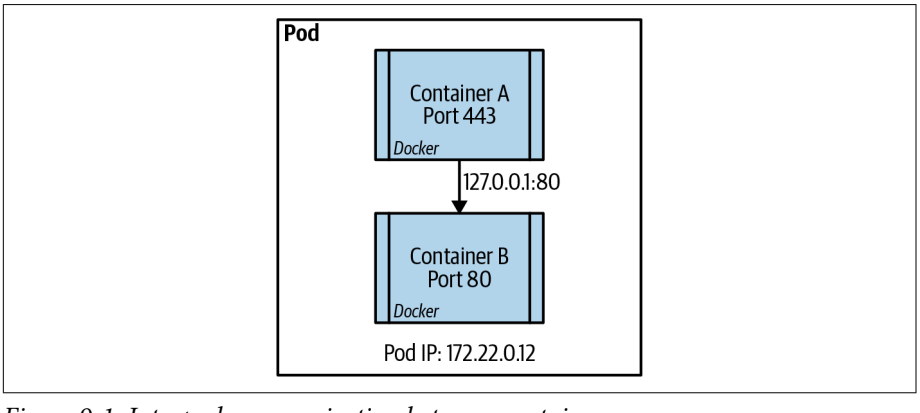



*Figure 9-1. Intrapod communication between containers*

*Pod-to-pod communication*
: All pods need to communicate with one another without any network address translation (NAT). This means that the IP address that a pod is seen as by the receiving pod is the sender's actual IP address. This is handled in different ways, depending on the network plug-in used, which we discuss in more detail later in the chapter. This rule is true between pods on the same node and pods that are on different nodes in the same cluster. This also extends to the node being able to communicate directly to the pod with no NAT involved. This allows host-based agents or system daemons to communicate to the pods as needed. Figure 9-2 is a representation of the communication processes between pods in the same node and pods in different nodes of the cluster.

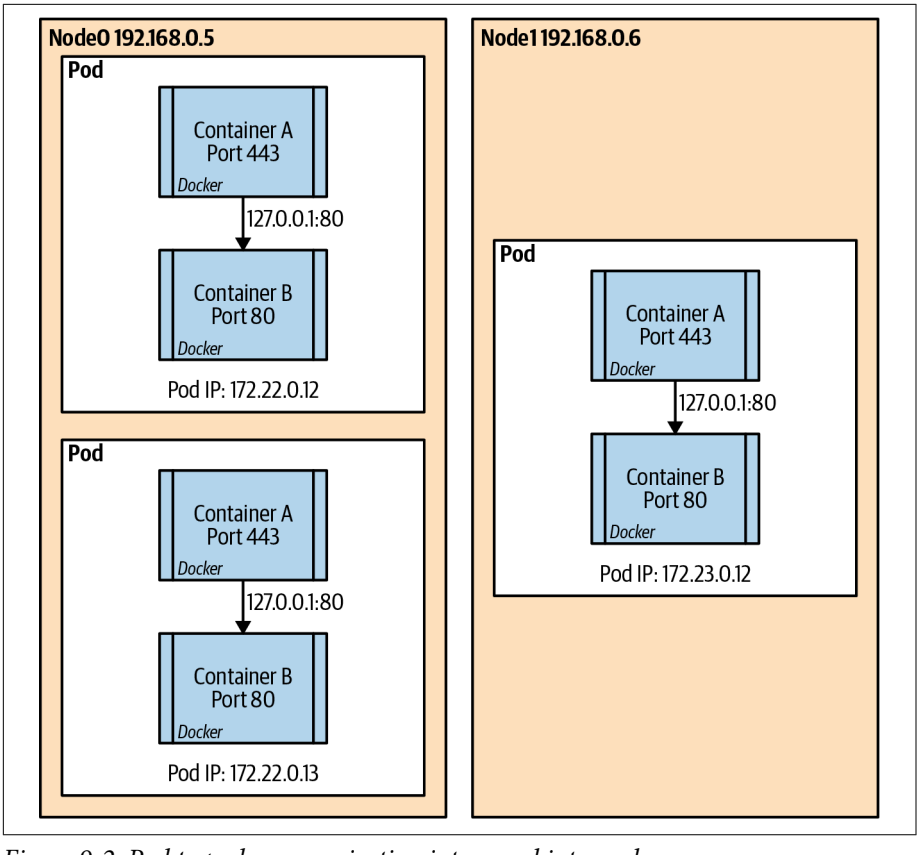



*Figure 9-2. Pod to pod communication intra- and internode*

*Service-to-pod communication*

Services in Kubernetes represent a durable IP address and port that is found on each node that will forward all traffic to the endpoints that are mapped to the service. Over the different iterations of Kubernetes, the method in favor of enabling this has changed, but the two main methods are via the use of iptables or the newer IP Virtual Server (IPVS). Most implementations today use the iptables implementation to enable a pseudo-Layer 4 load balancer on each node. Figure 9-3 is a visual representation of how the service is tied to the pods via label selectors.

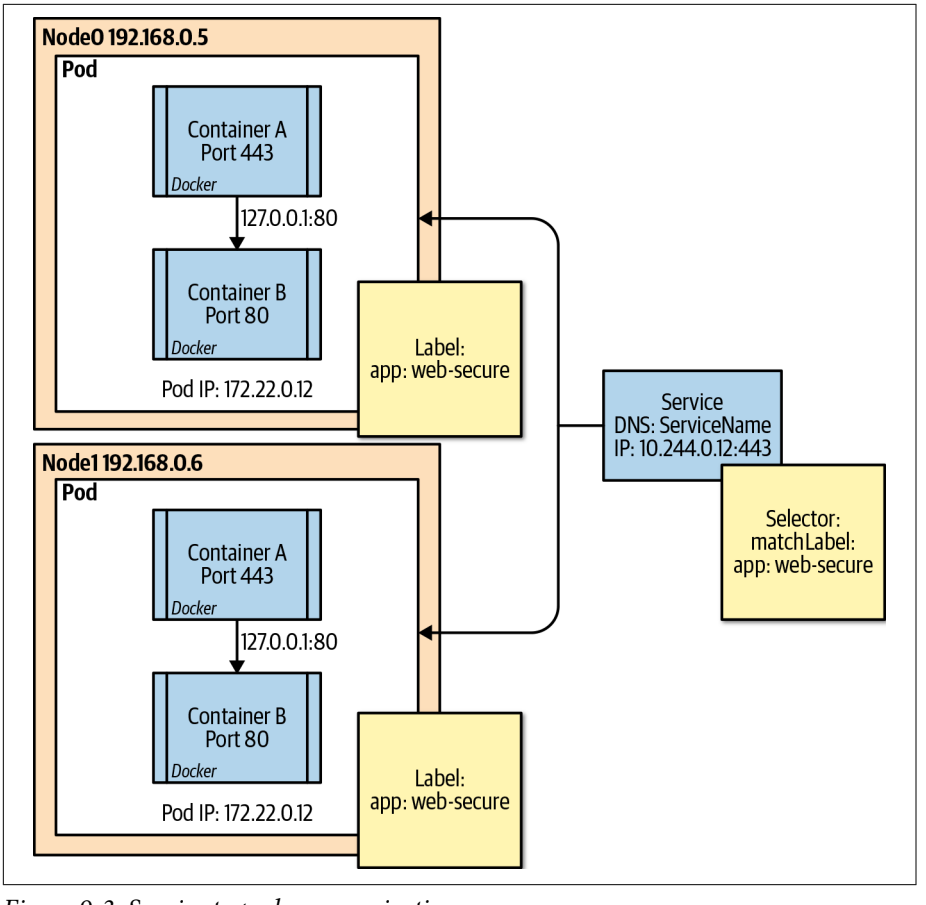



*Figure 9-3. Service to pod communication*

# Network Plug-ins

Early on, the Special Interest Group (SIG) guided the networking standards to more of a pluggable architecture, which opened the door for numerous third-party networking projects, which in many cases injected value-added capabilities into Kubernetes workloads. These network plug-ins come in two flavors. The most basic is called Kubenet and is the default plug-in provided by Kubernetes natively. The second type of plug-in follows the Container Network Interface (CNI) specification, which is a generic plug-in network solution for containers.

## Kubenet

Kubenet is the most basic network plug-in that comes out of the box in Kubernetes. It is the simplest of the plug-ins and provides a Linux bridge, `cbr0`, that's a virtual

Ethernet pair for the pods connected to it. The pod then gets an IP address from a Classless Inter-Domain Routing (CIDR) range that is distributed across the nodes of the cluster. There is also an IP masquerade flag that should be set to allow traffic destined to IPs outside the pod CIDR range to be masqueraded. This obeys the rules of pod-to-pod communication because only traffic destined outside the pod CIDR undergoes network address translation (NAT). After the packet leaves a node to go to another node, some kind of routing is put in place to facilitate the process to forward the traffic to the correct node.

## Kubenet Best Practices

- Kubenet allows for a simplistic network stack and does not consume precious IP addresses on already crowded networks. This is especially true of cloud networks that are extended to on-premises datacenters.
- Ensure that the pod CIDR range is large enough to handle the potential size of the cluster and the pods in each cluster. The default pods per node set in kubelet is 110, but you can adjust this.
- Understand and plan accordingly for the route rules to properly allow traffic to find pods in the proper nodes. In cloud providers, this is usually automated, but on-premises or edge cases will require automation and solid network management.

## The CNI Plug-in

The CNI plug-in has some basic requirements set aside by the specification. These specifications dictate the interfaces and minimal API actions that the CNI offers and how it will interface with the container runtime that is used in the cluster. The network management components are defined by the CNI, but they all must include some type of IP address management and minimally allow for the addition and deletion of a container to a network. The full original specification that was originally derived from the `rkt` networking proposal is available.

The Core CNI project provides libraries that you can use to write plug-ins that provide the basic requirements and that can call other plug-ins that perform various functions. This adaptability led to numerous CNI plug-ins that you can use in container networking from cloud providers like the Microsoft Azure native CNI and the Amazon Web Services (AWS) VPC CNI plug-in, to traditional network providers such as Nuage CNI, Juniper Networks Contrail/Tunsten Fabric, and VMware NSX.

## CNI Best Practices

Networking is a critical component of a functioning Kubernetes environment. The interaction between the virtual components within Kubernetes and the physical

network environment should be carefully designed to ensure dependable application communication:

1. Evaluate the feature set needed to accomplish the overall networking goals of the infrastructure. Some CNI plug-ins provide native high availability, multicloud connectivity, Kubernetes network policy support, and various other features.
2. If you are running clusters via public cloud providers, verify that any CNI plug-ins that are not native to the cloud provider's Software-Defined Network (SDN) are actually supported.
3. Verify that any network security tools, network observability, and management tools are compatible with the CNI plug-in of choice, and if not, research which tools can replace the existing ones. It is important to not lose either observability or security capabilities because the needs will be expanded when moving to a large-scale distributed system such as Kubernetes. You can add tools like Weaveworks Weave Scope, Dynatrace, and Sysdig to any Kubernetes environment, and each offers its own benefits. If you're running in a cloud provider's managed service, such as Azure AKS, Google GCE, or AWS EKS, look for native tools like Azure Container Insights and Network Watcher, Google Stackdriver, and AWS CloudWatch. Whatever tool you use, it should at least provide insight into the network stack and the Four Golden signals, made popular by the amazing Google SRE team and Rob Ewashuck: Latency, Traffic, Errors, and Saturation.
4. If you're using CNIs that do not provide an overlay network separate from the SDN network space, ensure that you have proper network address space to handle node IPs, pod IPs, internal load balancers, and overhead for cluster upgrade and scale out processes.

## Services in Kubernetes

When pods are deployed into a Kubernetes cluster, because of the basic rules of Kubernetes networking and the network plug-in used to facilitate these rules, pods can directly communicate only with other pods within the same cluster. Some CNI plug-ins give the pods IPs on the same network space as the nodes, so technically, after the IP of a pod is known, it can be accessed directly from outside the cluster. This, however, is not an efficient way to access services being served by a pod, because of the ephemeral nature of pods in Kubernetes. Imagine that you have a function or system that needs to access an API that is running in a pod in Kubernetes. For a while, that might work with no issue, but at some point there might be a voluntary or involuntary disruption that will cause that pod to disappear. Kubernetes will potentially create a replacement pod with a new name and IP address, so naturally there needs to be some mechanism to find the replacement pod. This is where the service API comes to the rescue.

The service API allows for a durable IP and port to be assigned within the Kubernetes cluster and automatically mapped to the proper pods as endpoints to the service. This magic happens through the aforementioned iptables or IPVS on Linux nodes to create a mapping of the assigned service IP and port to the endpoint's or pod's actual IPs. The controller that manages this is called the `kube-proxy` service, which actually runs on each node in the cluster. It is responsible for manipulating the iptables rules on each node.

When a service object is defined, the type of service needs to be defined. The service type will dictate whether the endpoints are exposed only within the cluster or outside of the cluster. There are four basic service types that we will discuss briefly in the following sections.

## Service Type ClusterIP

ClusterIP is the default service type if one is not declared in the specification. ClusterIP means that the service is assigned an IP from a designated service CIDR range. This IP is as long lasting as the service object, so it provides an IP and port and protocol mapping to backend pods using the selector field; however, as we will see, there are cases for which you can have no selector. The declaration of the service also provides for a Domain Name System (DNS) name for the service. This facilitates service discovery within the cluster and allows for workloads to easily communicate to other services within the cluster by using DNS lookup based on the service name. As an example, if you have the service definition shown in the following example and need to access that service from another pod inside the cluster via an HTTP call, the call can simply use *http://web1-svc* if the client is in the same namespace as the service:

```
apiVersion: v1
kind: Service
metadata:
  name: web1-svc
spec:
  selector:
    app: web1
  ports:
  - port: 80
    targetPort: 8081
```

If it is required to find services in other namespaces, the DNS pattern would be `<service_name>.<namespace_name>.svc.cluster.local`.

If no selector is given in a service definition, the endpoints can be explicitly defined for the service by using an endpoint API definition. This will basically add an IP and port as a specific endpoint to a service instead of relying on the selector attribute to automatically update the endpoints from the pods that are in scope by the selector match. This can be useful in a few scenarios in which you have a specific database that is not in a cluster that is to be used for testing but you will change the service

later to a Kubernetes-deployed database. This is sometimes called a *headless service* because it is not managed by `kube-proxy` as other services are, but you can directly manage the endpoints, as shown in Figure 9-4.

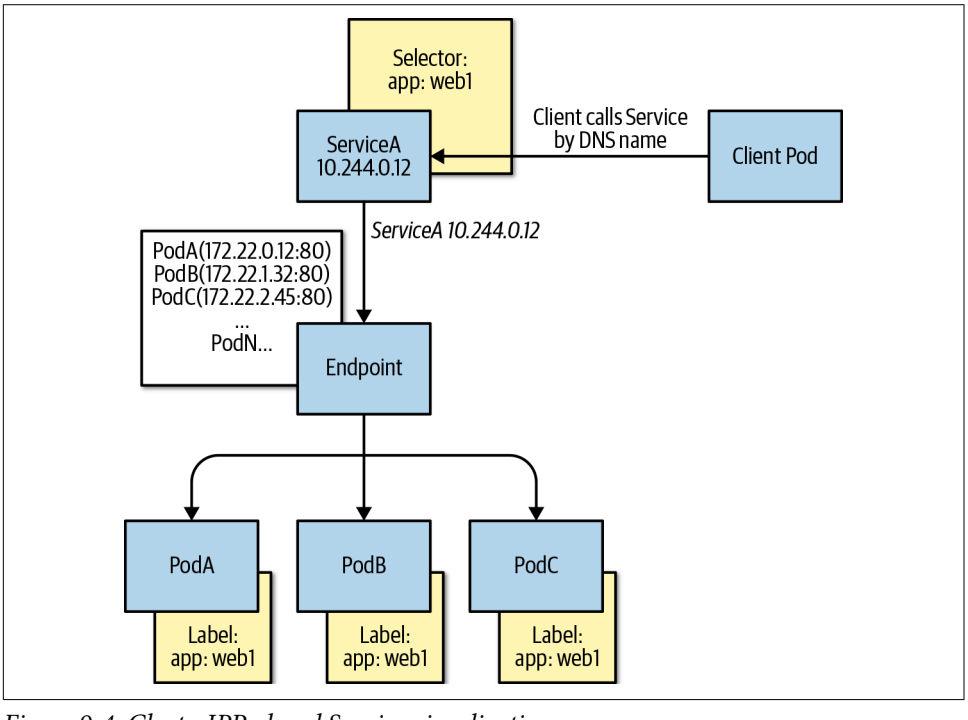



*Figure 9-4. ClusterIPPod and Service visualization*

## Service Type NodePort

The NodePort service type assigns a high-level port on each node of the cluster to the Service IP and port on each node. The high-level NodePorts fall within the 30,000 through 32,767 ranges and can either be statically assigned or explicitly defined in the service specification. NodePorts are usually used for on-premises clusters or bespoke solutions that do not offer automatic load-balancing configuration. To directly access the service from outside the cluster, use NodeIP:NodePort, as depicted in Figure 9-5.

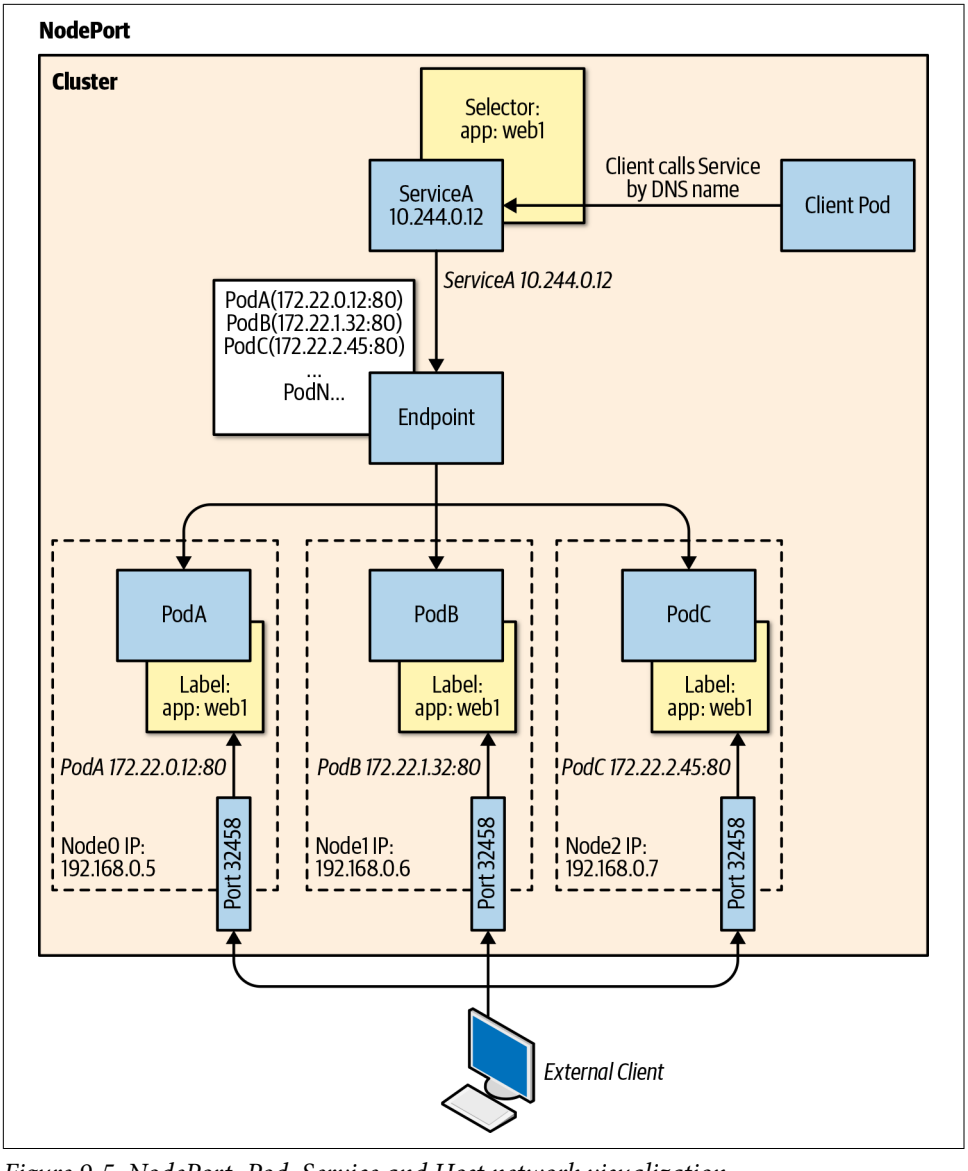



*Figure 9-5. NodePort–Pod, Service and Host network visualization*

## Service Type ExternalName

The ExternalName service type is seldom used in practice, but it can be helpful for passing cluster-durable DNS names to external DNS named services. A common example is an external database service from a cloud provider that has a unique DNS provided by the cloud provider, such as `mymongodb.documents.azure.com`.

Technically, this can be added very easily to a pod specification using an `Environment` variable, as discussed in Chapter 6; however, it might be more advantageous to use a more generic name in the cluster, such as `prod-mongodb`, which enables the change of the actual database it points to by just changing the service specification instead of having to recycle the pods because the `Environment` variable has changed:

```
kind: Service
apiVersion: v1
metadata:
  name: prod-mongodb
  namespace: prod
spec:
  type: ExternalName
  externalName: mymongodb.documents.azure.com
```

## Service Type LoadBalancer

LoadBalancer is a very special service type because it enables automation with cloud providers and other programmable cloud infrastructure services. The `LoadBalancer` type is a single method to ensure the deployment of the load-balancing mechanism that the infrastructure provider of the Kubernetes cluster provides. This means that in most cases, `LoadBalancer` will work roughly the same way in AWS, Azure, GCE, OpenStack, and others. In most cases, this entry will create a public-facing load-balanced service; however, each cloud provider has some specific annotations that enable other features, such as internal-only load balancers, AWS ELB configuration parameters, and so on. You can also define the actual load-balancer IP to use and the source ranges to allow within the service specification, as seen in the code sample that follows and the visual representation in Figure 9-6:

```
kind: Service
apiVersion: v1
metadata:
  name: web-svc
spec:
  type: LoadBalancer
  selector:
    app: web
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8081
  loadBalancerIP: 13.12.21.31
  loadBalancerSourceRanges:
  - "142.43.0.0/16"
```

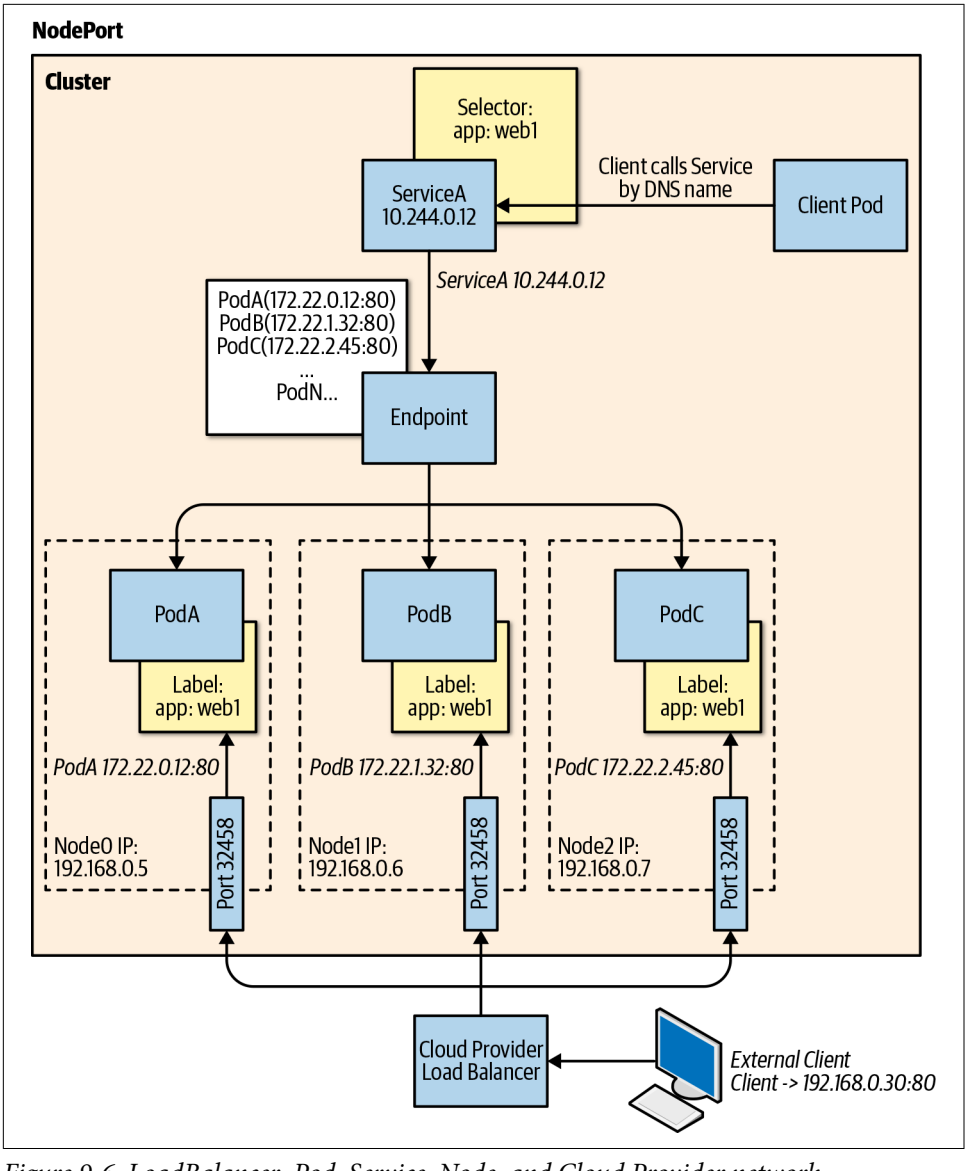



*Figure 9-6. LoadBalancer–Pod, Service, Node, and Cloud Provider network visualization*

## Ingress and Ingress Controllers

Although not technically a service type in Kubernetes, the Ingress specification is an important concept for ingress to workloads in Kubernetes. Services, as defined by the Service API, allow for a basic level of Layer 3/4 load balancing. The reality is that

many of the stateless services that are deployed in Kubernetes require a high level of traffic management and usually require application-level control: more specifically, HTTP protocol management.

The Ingress API is basically an HTTP-level router that allows for host- and path-based rules to direct to specific backend services. Imagine a website hosted on *www.evillgenius.com* and two different paths that are hosted on that site, */registration* and */labaccess*, that are served by two different services hosted in Kubernetes, `reg-svc` and `labaccess-svc`. You can define an ingress rule to ensure that requests to *www.evillgenius/registration* are forwarded to the `reg-svc` service and the correct endpoint pods, and, similarly, that requests to *www.evillgenius.com/labaccess* are forwarded to the correct endpoints of the `labaccess-svc` service. The Ingress API also allows for host-based routing to allow for different hosts on a single ingress. An additional feature is the ability to declare a Kubernetes secret that holds the certificate information for Transport Layer Security (TLS) termination on port 443. When a path is not specified, there is usually a default backend that can be used to give a better user experience than the standard 404 error.

The details around the specific TLS and default backend configuration are actually handled by what is known as the Ingress controller. The Ingress controller is decoupled from the Ingress API and allows for operators to deploy an Ingress controller of choice, such as NGINX, Traefik, HAProxy, and others. An Ingress controller, as the name suggests, is a controller, just like any Kubernetes controller, but it's not part of the system and is instead a third-party controller that understands the Kubernetes Ingress API for dynamic configuration. The most common implementation of an Ingress controller is NGINX because it is partly maintained by the Kubernetes project; however, there are numerous examples of both open source and commercial Ingress controllers:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: labs-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  tls:
  - hosts:
    - www.evillgenius.com
    secretName: secret-tls
  rules:
  - host: www.evillgenius.com
    http:
      paths:
      - path: /registration
        backend:
          serviceName: reg-svc
```

```
      servicePort: 8088
  - path: /labaccess
    backend:
      serviceName: labaccess-svc
      servicePort: 8089
```

## Services and Ingress Controllers Best Practices

Creating a complex virtual network environment with interconnected applications requires careful planning. Effectively managing how the different services of the application communicate with one another and to the outside world requires constant attention as the application changes. These best practices will help make the management easier:

- Limit the number of services that need to be accessed from outside the cluster. Ideally, most services will be ClusterIP, and only external-facing services will be exposed externally to the cluster.
- If the services that need to be exposed are primarily HTTP/HTTPS-based services, it is best to use an Ingress API and Ingress controller to route traffic to backing services with TLS termination. Depending on the type of Ingress controller used, features such as rate limiting, header rewrites, OAuth authentication, observability, and other services can be made available without having to build them into the applications themselves.
- Choose an Ingress controller that has the needed functionality for secure ingress of your web-based workloads. Standardize on one and use it across the enterprise because many of the specific configuration annotations vary between implementations and prevent the deployment code from being portable across enterprise Kubernetes implementations.
- Evaluate cloud service provider-specific Ingress controller options to move the infrastructure management and load of the ingress out of the cluster, but still allow for Kubernetes API configuration.
- When serving mostly APIs externally, evaluate API-specific Ingress controllers, such as Kong or Ambassador, that have more fine-tuning for API-based workloads. Although NGINX, Traefik, and others might offer some API tuning, it will not be as fine-grained as specific API proxy systems.
- When deploying Ingress controllers as pod-based workloads in Kubernetes, ensure that the deployments are designed for high availability and aggregate performance throughput. Use metrics observability to properly scale the ingress, but include enough cushion to prevent client disruptions while the workload scales.

## Network Security Policy

The NetworkPolicy API built into Kubernetes allows for network-level ingress and egress access control defined with your workload. Network policies allow you to control how groups of pods are allowed to communicate with one another and with other endpoints. If you want to dig deeper into the NetworkPolicy specification, it might sound confusing, especially given that it is defined as a Kubernetes API, but it requires a network plug-in that supports the NetworkPolicy API.

Network policies have a simple YAML structure that can look complicated, but if you think of it as a simple East-West traffic firewall, it might help you to understand it a little better. Each policy specification has `podSelector`, `ingress`, `egress`, and `policyType` fields. The only required field is `podSelector`, which follows the same convention as any Kubernetes selector with a `matchLabels`. You can create multiple NetworkPolicy definitions that can target the same pods, and the effect is additive in nature. Because NetworkPolicy objects are namespaced objects, if no selector is given for a `podSelector`, all pods in the namespace fall into the scope of the policy. If there are any ingress or egress rules defined, this creates a whitelist of what is allowed to or from the pod. There is an important distinction here: if a pod falls into the scope of a policy because of a selector match, all traffic, unless explicitly defined in an ingress or egress rule, is blocked. This little, nuanced detail means that if a pod does not fall into any policy because of a selector match, all ingress and egress is allowed to the pod. This was done on purpose to allow for ease of deploying new workloads into Kubernetes without any blockers.

The `ingress` and `egress` fields are basically a list of rules based on source or destination and can be specific CIDR ranges, `podSelectors`, or `namespaceSelectors`. If you leave the ingress field empty, it is like a deny-all inbound. Similarly, if you leave the egress empty, it is deny-all outbound. Port and protocol lists are also supported to further tighten down the type of communications allowed.

The `policyTypes` field specifies to which network policy rule types the policy object is associated. If the field is not present, it will just look at the `ingress` and `egress` lists fields. The difference again is that you must explicitly call out egress in `policyTypes` and also have an egress rule list for this policy to work. Ingress is assumed, and defining it explicitly is not needed.

Let's use a prototypical example of a three-tier application deployed to a single namespace where the tiers are labeled as `tier: "web"`, `tier: "db"`, and `tier: "api"`. If you want to ensure that traffic is limited to each tier properly, create a NetworkPolicy manifest like this:

Default deny rule:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

Web layer network policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: webaccess
spec:
  podSelector:
    matchLabels:
      tier: "web"
  policyTypes:
  - Ingress
  ingress:
  - {}
```

API layer network policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-api-access
spec:
  podSelector:
    matchLabels:
      tier: "api"
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          tier: "web"
```

Database layer network policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-db-access
spec:
  podSelector:
    matchLabels:
      tier: "db"
```

```
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          tier: "api"
```

## Network Policy Best Practices

Securing network traffic in an enterprise system was once the domain of physical hardware devices with complex networking rule sets. Now, with Kubernetes network policy, a more application-centric approach can be taken to segment and control the traffic of the applications hosted in Kubernetes. Some common best practices apply no matter which policy plug-in used:

- Start off slow and focus on traffic ingress to pods. Complicating matters with ingress and egress rules can make network tracing a nightmare. As soon as traffic is flowing as expected, you can begin to look at egress rules to further control flow to sensitive workloads. The specification also favors ingress because it defaults many options even if nothing is entered into the ingress rules list.
- Ensure that the network plug-in used either has some of its own interface to the NetworkPolicy API or supports other well-known plug-ins. Example plug-ins include Calico, Cilium, Kube-router, Romana, and Weave Net.
- If the network team is used to having a "default-deny" policy in place, create a network policy such as the following for each namespace in the cluster that will contain workloads to be protected. This ensures that even if another network policy is deleted, no pods are accidentally "exposed":

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

4. If there are pods that need to be accessed from the internet, use a label to explicitly apply a network policy that allows ingress. Be aware of the entire flow in case the actual IP that a packet is coming from is not the internet, but the internal IP of a load balancer, firewall, or other network device. For example, to allow traffic from all (including external) sources for pods having the `allow-internet=true` label, do this:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: internet-access
spec:
  podSelector:
    matchLabels:
      allow-internet: "true"
  policyTypes:
  - Ingress
  ingress:
  - {}
```

5. Try to align application workloads to single namespaces for ease of creating rules because the rules themselves are namespace specific. If cross-namespace communication is needed, try to be as explicit as possible and perhaps use specific labels to identify the flow pattern:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: namespace-foo-2-namespace-bar
  namespace: bar
spec:
  podSelector:
    matchLabels:
      app: bar-app
  policyTypes:
  - Ingress
  ingress:
  - from:
    -  namespaceSelector:
        matchLabels:
          networking/namespace: foo
       podSelector:
        matchLabels:
          app: foo-app
```

6. Have a test bed namespace that has fewer restrictive policies, if any at all, to allow time to investigate the correct traffic patterns needed.

## Service Meshes

It is easy to imagine a single cluster hosting hundreds of services that load-balance across thousands of endpoints that communicate with one another, access external resources, and potentially are being accessed from external sources. This can be quite daunting when trying to manage, secure, observe, and trace all of the connections between these services, especially with the dynamic nature of the endpoints coming and going from the overall system. The concept of a *service mesh*, which is not unique

to Kubernetes, allows for control over how these services are connected and secured with a dedicated date plane and control plane. Service meshes all have different capabilities, but usually they all offer some of the following:

- Load balancing of traffic with potentially fine-grained traffic-shaping policies that are distributed across the mesh.
- Service discovery of services that are members of the mesh, which might include services within a cluster or in another cluster, or an outside system that is a member of the mesh.
- Observability of the traffic and services, including tracing across the distributed services using tracing systems like Jaeger or Zipkin that follow the OpenTracing standards.
- Security of the traffic in the mesh using mutual authentication. In some cases, not only pod-to-pod or East-West traffic is secured, but an Ingress controller is also provided that offers North-South security and control.
- Resiliency, health, and failure-prevention capabilities that allow for patterns such as circuit breaker, retries, deadlines, and so on.

The key here is that all of these features are integrated into the applications that take part in the mesh with little or no application changes. How can all of these amazing features come for free? Sidecar proxies are usually the way this is done. The majority of service meshes available today inject a proxy that is part of the data plane into each pod that is a member of the mesh. This allows for policies and security to be synchronized across the mesh by the control-plane components. This really hides the network details from the container that holds the workload and leaves it to the proxy to handle the complexity of the distributed network. To the application, it just talks via localhost to its proxy. In many cases, the control plane and data plane might be different technologies but complementary to each other.

In many cases, the first service mesh that comes to mind is Istio, a project by Google, Lyft, and IBM that uses Envoy as its data-plane proxy and uses proprietary control-plane components Mixer, Pilot, Galley, and Citadel. There are other service meshes that offer varying levels of capabilities, such as Linkerd2, which uses its own data-plane proxy built using Rust. HashiCorp has recently added more Kubernetes-centric service mesh capabilities to Consul, which allows you to choose between Consul's own proxy or Envoy, and offers commercial support for its service mesh.

The topic of service meshes in Kubernetes is a fluid one—if not overly emotional in many social media tech circles—so a detailed explanation of each mesh has no value here. I would be remiss if I did not mention the promising efforts lead by Microsoft, Linkerd, HashiCorp, Solo.io, Kinvolk, and Weaveworks around the Service Mesh Interface (SMI). The SMI hopes to set a standard interface for basic feature sets that

are expected of all service meshes. The specification as of this writing covers traffic policy such as identity and transport-level encryption, traffic telemetry that captures key metrics between services in the mesh, and traffic management to allow for traffic shifting and weighting between different services. This project hopes to take some of the variability out of the service meshes yet allow for service mesh vendors to extend and build value-added capabilities into their products to differentiate themselves from others.

## Service Mesh Best Practices

The service mesh community continues to grow every day, and as more and more enterprises help define their needs, the service mesh ecosystem will change dramatically. These best practices are, as of this writing, based on common necessities that service meshes try to solve today:

- Rate the importance of the key features service meshes offer and determine which current offerings provide the most important features with the least amount of overhead. Overhead here is both human technical debt and infrastructure resource debt. If all that is really required is mutual TLS between certain pods, would it be easier to perhaps find a CNI that offers that integrated into the plug-in?
- Is the need for a cross-system mesh such as multicloud or hybrid scenarios a key requirement? Not all service meshes offer this capability, and if they do, it is a complicated process that often introduces fragility into the environment.
- Many of the service mesh offerings are open source community-based projects, and if the team that will be managing the environment is new to service meshes, commercially supported offerings might be a better option. There are companies that are beginning to offer commercially supported and managed service meshes based on Istio, which can be helpful because it is almost universally agreed upon that Istio is a complicated system to manage.

# Summary

In addition to application management, one of the most important things that Kubernetes provides is the ability to link different pieces of your application together. In this chapter, we looked at the details of how Kubernetes works, including how pods get their IP addresses through CNI plug-ins, how those IPs are grouped together to form services, and how more application or Layer 7 routing can be implemented via Ingress resources (which in turn use services). You also saw how to limit traffic and secure your network using networking policies, and, finally, how service mesh technologies are transforming the ways in which people connect and monitor the connections between their services. In addition to setting up your application to run

and be deployed reliably, setting up the networking for your application is a crucial piece of using Kubernetes successfully. Understanding how Kubernetes approaches networking and how that intersects optimally with your application is a critical piece of its ultimate success.

CHAPTER 10

# Pod and Container Security

When it comes to pod security via the Kubernetes API, you have two main options at your disposal: PodSecurityPolicy and RuntimeClass. In this chapter, we review the purpose and use of each API and provide best practices for their use.

## PodSecurityPolicy API

The PodSecurityPolicy API is under active development. As of Kubernetes 1.15, this API was in beta. Please visit the upstream documentation for the latest updates on the feature state.

This cluster-wide resource creates a single place to define and manage all of the security-sensitive fields found in pod specifications. Prior to the creation of the PodSecurityPolicy resource, cluster administrators and/or users would need to independently define individual `SecurityContext` settings for their workloads or enable bespoke admission controllers on the cluster to enforce some aspects of pod security.

Does all of this sound too easy? PodSecurityPolicy is surprisingly difficult to implement effectively and will more often than not get turned off or evaded in other ways. We do, however, strongly suggest taking the time to fully understand PodSecurityPolicy because it's one of the single most effective means to reduce your attack surface area by limiting what can run on your cluster and with what level of privilege.

### Enabling PodSecurityPolicy

Along with the resource API, a corresponding admission controller must be enabled to enforce the conditions defined in the PodSecurityPolicy resource. This means that

the enforcement of these policies happens at the admission phase of the request flow. To learn more about how admission controllers work, refer to Chapter 17.

It's worth mentioning that enabling PodSecurityPolicy is not widely available among public cloud providers and cluster operations tools. In the cases for which it is available, it's generally shipped as an opt-in feature.

Proceed with caution when enabling PodSecurityPolicy because it's potentially workload blocking if adequate preparation isn't done at the outset.

There are two main components that you need to complete in order to start using PodSecurityPolicy:

1. Ensure that the PodSecurityPolicy API is enabled (this should already be done if you're on a currently supported version of Kubernetes).

   You can confirm that this API is enabled by running `kubectl get psp`. As long as the response isn't `the server doesn't have a resource type "PodSecurityPolicies`, you are OK to proceed.
2. Enable the PodSecurityPolicy admission controller via the `api-server flag --enable-admission-plugins`.

If you are enabling PodSecurityPolicy on an existing cluster with running workloads, you must create all necessary policies, service accounts, roles, and role bindings before enabling the admission controller.

We also recommend the addition of the `--use-service-account-credentials=true` flag to `kube-controller-manager`, which will enable service accounts to be used for each individual controller within `kube-controller-manager`. This allows for more granular policy control even within the `kube-system` namespace. You can simply run the following command to determine whether the flag has been set. It demonstrates that there is indeed a service account per controller:

```
$ kubectl get serviceaccount -n kube-system | grep '.*-controller'
attachdetach-controller             1         6d13h
certificate-controller              1         6d13h
clusterrole-aggregation-controller  1         6d13h
cronjob-controller                  1         6d13h
daemon-set-controller               1         6d13h
deployment-controller               1         6d13h
```

```
disruption-controller                1         6d13h
endpoint-controller                  1         6d13h
expand-controller                    1         6d13h
job-controller                       1         6d13h
namespace-controller                 1         6d13h
node-controller                      1         6d13h
pv-protection-controller             1         6d13h
pvc-protection-controller            1         6d13h
replicaset-controller                1         6d13h
replication-controller               1         6d13h
resourcequota-controller             1         6d13h
service-account-controller           1         6d13h
service-controller                   1         6d13h
statefulset-controller               1         6d13h
ttl-controller                       1         6d13h
```

It's extremely important to remember that having no PodSecurityPolicies defined will result in an implicit deny. This means that without a policy match for the workload, the pod will not be created.

## Anatomy of a PodSecurityPolicy

To best understand how PodSecurityPolicy enables you to secure your pods, let's work through an end-to-end example together. This will help solidify the order of operations from policy creation through use.

Before you continue, the following section requires that your cluster have PodSecurityPolicy enabled in order for it to work. To see how to enable it, refer to the previous section.

You should not enable PodSecurityPolicy on a live cluster without considering the warnings provided in the previous section. Proceed with caution.

Let's first test the experience without making any changes or creating any policies. The following is a test workload that simply runs the trusty pause container in a Deployment (save this file as *pause-deployment.yaml* on your local filesystem for use throughout this section):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pause-deployment
  namespace: default
  labels:
```

```
    app: pause
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pause
  template:
    metadata:
      labels:
        app: pause
    spec:
      containers:
      - name: pause
        image: k8s.gcr.io/pause
```

By running the following command, you can verify that you have a Deployment and a corresponding ReplicaSet but NO pod:

```
$ kubectl get deploy,rs,pods -l app=pause
NAME                                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.extensions/pause-delpoyment   0/1     0            0           41s

NAME                                                DESIRED   CURRENT   READY
AGE
replicaset.extensions/pause-delpoyment-67b77c4f69   1         0         0
41s
```

If you describe the ReplicaSet, you can confirm the cause from the event log:

```
$ kubectl describe replicaset -l app=pause
Name:           pause-delpoyment-67b77c4f69
Namespace:      default
Selector:       app=pause,pod-template-hash=67b77c4f69
Labels:         app=pause
                pod-template-hash=67b77c4f69
Annotations:    deployment.kubernetes.io/desired-replicas: 1
                deployment.kubernetes.io/max-replicas: 2
                deployment.kubernetes.io/revision: 1
Controlled By:  Deployment/pause-delpoyment
Replicas:       0 current / 1 desired
Pods Status:    0 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=pause
           pod-template-hash=67b77c4f69
  Containers:
   pause:
    Image:        k8s.gcr.io/pause
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
```

```
  Type             Status  Reason
  ----             ------  ------
  ReplicaFailure   True    FailedCreate
Events:
  Type     Reason        Age                 From                   Message
  ----     ------        ----                ----                   -------
  Warning  FailedCreate  45s (x15 over 2m7s)  replicaset-controller  Error cre-
ating: pods "pause-delpoyment-67b77c4f69-" is forbidden: unable to validate
against any pod security policy: []
```

This is because there are either no pod security policies defined or the service account is not allowed access to use the PodSecurityPolicy. You might have also noticed that all of the system pods in the `kube-system` namespace are probably still in `RUNNING` state. This is because these requests have already passed the admission phase for the request. If there were an event that restarted these pods, they would also suffer the same fate as our test workload given that there are no PodSecurityPolicy resources defined:

```
replicaset-controller  Error creating: pods "pause-delpoyment-67b77c4f69-" is
forbidden: unable to validate against any pod security policy: []
```

Let's delete the test workload deployment:

```
$ kubectl delete deploy -l app=pause
deployment.extensions "pause-delpoyment" deleted
```

Now, let's go fix this by defining pod security policies. For a complete list of policy settings, refer to the Kubernetes documentation. The following policies are basic variations of the examples provided in the Kubernetes documentation.

Call the first policy `privileged`, which we use to demonstrate how to allow privileged workloads. You can apply the following resources by using `kubectl create -f <filename>`:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: privileged
spec:
  privileged: true
  allowPrivilegeEscalation: true
  allowedCapabilities:
  - '*'
  volumes:
  - '*'
  hostNetwork: true
  hostPorts:
  - min: 0
    max: 65535
  hostIPC: true
  hostPID: true
  runAsUser:
```

```
    rule: 'RunAsAny'
  seLinux:
    rule: 'RunAsAny'
  supplementalGroups:
    rule: 'RunAsAny'
  fsGroup:
    rule: 'RunAsAny'
```

The next policy defines restricted access and will suffice for many workloads apart from those responsible for running Kubernetes cluster-wide services such as `kube-proxy`, located in the `kube-system` namespace:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
spec:
  privileged: false
  allowPrivilegeEscalation: false
  requiredDropCapabilities:
    - ALL
  volumes:
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'downwardAPI'
    - 'persistentVolumeClaim'
  hostNetwork: false
  hostIPC: false
  hostPID: false
  runAsUser:
    rule: 'RunAsAny'
  seLinux:
    rule: 'RunAsAny'
  supplementalGroups:
    rule: 'MustRunAs'
    ranges:
      - min: 1
        max: 65535
  fsGroup:
    rule: 'MustRunAs'
    ranges:
      - min: 1
        max: 65535
  readOnlyRootFilesystem: false
```

You can confirm that the policies have been created by running the following command:

```
$ kubectl get psp
NAME         PRIV    CAPS   SELINUX    RUNASUSER          FSGROUP
SUPGROUP    READONLYROOTFS   VOLUMES
```

```
privileged   true    *       RunAsAny   RunAsAny           RunAsAny    RunA-
sAny    false            *
restricted   false           RunAsAny   MustRunAsNonRoot   MustRunAs   MustRu-
nAs   false            configMap,emptyDir,projected,secret,downwardAPI,persis-
tentVolumeClaim
```

Now that we have defined these policies, we need to grant the service accounts access to `use` these policies via Role-Based Access Control (RBAC).

First, create the following `ClusterRole` that allows access to `use` the restricted PodSecurityPolicy that we created in the previous step:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psp-restricted
rules:
- apiGroups:
  - extensions
  resources:
  - podsecuritypolicies
  resourceNames:
  - restricted
  verbs:
  - use
```

Now, create the following `ClusterRole` that allows access to `use` the privileged PodSecurityPolicy we created in the previous step:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psp-privileged
rules:
- apiGroups:
  - extensions
  resources:
  - podsecuritypolicies
  resourceNames:
  - privileged
  verbs:
  - use
```

We must now create a corresponding `ClusterRoleBinding` that allows the `system:serviceaccounts` group access to `psp-restricted` `ClusterRole`. This group includes all of the `kube-controller-manager` controller service accounts:

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psp-restricted
subjects:
- kind: Group
```

```
    name: system:serviceaccounts
    namespace: kube-system
  roleRef:
    kind: ClusterRole
    name: psp-restricted
    apiGroup: rbac.authorization.k8s.io
```

Go ahead and create the test workload again. You can see that the pod is now up and running:

```
$ kubectl create -f pause-deployment.yaml
deployment.apps/pause-deployment created
$ kubectl get deploy,rs,pod
NAME                                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.extensions/pause-deployment   1/1     1            1           10s

NAME                                               DESIRED   CURRENT   READY
AGE
replicaset.extensions/pause-deployment-67b77c4f69   1         1         1
10s

NAME                                    READY   STATUS    RESTARTS   AGE
pod/pause-deployment-67b77c4f69-4gmdn   1/1     Running   0          9s
```

Update the test workload deployment to violate the restricted policy. Adding `privileged=true` should do the trick. Save this manifest as *pause-privileged-deployment.yaml* on your local filesystem and then apply it by using `kubectl apply -f <filename>`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pause-privileged-deployment
  namespace: default
  labels:
    app: pause
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pause
  template:
    metadata:
      labels:
        app: pause
    spec:
      containers:
      - name: pause
        image: k8s.gcr.io/pause
        securityContext:
          privileged: true
```

Again, you can see that both the Deployment and the ReplicaSet have been created; however, the pod has not. You can find the details of why in the event log of the ReplicaSet:

```
$ kubectl create -f pause-privileged-deployment.yaml
deployment.apps/pause-privileged-deployment created
$ kubectl get deploy,rs,pods -l app=pause
NAME                                                  READY   UP-TO-DATE
AVAILABLE   AGE
deployment.extensions/pause-privileged-deployment   0/1     0
0           37s

NAME                                                              DESIRED
CURRENT   READY   AGE
replicaset.extensions/pause-privileged-deployment-6b7bcfb9b7   1
0         0       37s
$ kubectl describe replicaset -l app=pause
Name:           pause-privileged-deployment-6b7bcfb9b7
Namespace:      default
Selector:       app=pause,pod-template-hash=6b7bcfb9b7
Labels:         app=pause
                pod-template-hash=6b7bcfb9b7
Annotations:    deployment.kubernetes.io/desired-replicas: 1
                deployment.kubernetes.io/max-replicas: 2
                deployment.kubernetes.io/revision: 1
Controlled By:  Deployment/pause-privileged-deployment
Replicas:       0 current / 1 desired
Pods Status:    0 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=pause
           pod-template-hash=6b7bcfb9b7
  Containers:
   pause:
    Image:        k8s.gcr.io/pause
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type             Status  Reason
  ----             ------  ------
  ReplicaFailure   True    FailedCreate
Events:
  Type     Reason        Age                  From                   Message
  ----     ------        ----                 ----                   -------
  Warning  FailedCreate  78s (x15 over 2m39s)  replicaset-controller  Error cre-
ating: pods "pause-privileged-deployment-6b7bcfb9b7-" is forbidden: unable to
validate against any pod security policy: [spec.containers[0].securityCon-
text.privileged: Invalid value: true: Privileged containers are not allowed]
```

The preceding example shows the exact reason why: `Privileged containers are not allowed`. Let's delete the test workload deployment.

```
$ kubectl delete deploy pause-privileged-deployment
deployment.extensions "pause-privileged-deployment" deleted
```

So far, we've dealt only with cluster-level bindings. How about we allow the test workload access to the privileged policy using a service account.

First, create a `serviceaccount` in the default namespace:

```
$ kubectl create serviceaccount pause-privileged
serviceaccount/pause-privileged created
```

Bind that `serviceaccount` to the permissive `ClusterRole`. Save this manifest as role-pause-privileged-psp-permissive.yaml on your local filesystem and then apply it by using kubectl apply -f <filename>:

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: pause-privileged-psp-permissive
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: psp-privileged
subjects:
- kind: ServiceAccount
  name: pause-privileged
  namespace: default
```

Finally, update the test workload to use the `pause-privileged` service account. Then apply it to the cluster using kubectl apply:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pause-privileged-deployment
  namespace: default
  labels:
    app: pause
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pause
  template:
    metadata:
      labels:
        app: pause
    spec:
      containers:
```

```
    - name: pause
      image: k8s.gcr.io/pause
      securityContext:
        privileged: true
     serviceAccountName: pause-privileged
```

You can see that the pod is now able to use the privileged policy:

```
$ kubectl create -f pause-privileged-deployment.yaml
deployment.apps/pause-privileged-deployment created
$ kubectl get deploy,rs,pod
NAME                                                READY   UP-TO-DATE
AVAILABLE   AGE
deployment.extensions/pause-privileged-deployment   1/1     1
1           14s

NAME                                                           DESIRED
CURRENT   READY   AGE
replicaset.extensions/pause-privileged-deployment-658dc5569f   1
1         1       14s

NAME                                              READY   STATUS    RESTARTS
AGE
pod/pause-privileged-deployment-658dc5569f-nslnw   1/1     Running   0
14s
```

You can see which PodSecurityPolicy was matched by using the following command:

```
$ kubectl get pod -l app=pause -o yaml | grep psp
      kubernetes.io/psp: privileged
```

## PodSecurityPolicy Challenges

Now that you understand how to configure and use PodSecurityPolicy, it's worth noting that there are quite a few challenges with using it in real-world environments. In this section, we describe things that we have experienced that make it challenging.

### Reasonable default policies

The real power of PodSecurityPolicy is to enable the cluster administrator and/or user to ensure that their workloads meet a certain level of security. In practice, you might often overlook just how many workloads run as root, use `hostPath` volumes, or have other risky settings that force you to craft policies with security holes just to get the workloads up and running.

### Lots of toil

Getting the policies just right is a large investment, especially where there is a large set of workloads already running on Kubernetes without PodSecurityPolicy enabled.

#### Are your developers interested in learning PodSecurityPolicy?

Will your developers want to learn PodSecurityPolicy? What would be the incentive for them to do so? Without a lot of up front coordination and automation to make enabling PodSecurityPolicy a smooth transition, it's very likely that PodSecurityPolicy won't be adopted at all.

#### Debugging is cumbersome

It's difficult to troubleshoot policy evaluation. For example, you might want to understand why your workload matched or didn't match a specific policy. Tooling or logging to make that easy doesn't exist at this stage.

#### Do you rely on artifacts outside your control?

Are you pulling images from Docker Hub or another public repository? Chances are they will violate your policies in some shape or form and will be out of your control to fix. Another common place is Helm charts: do they ship with the appropriate policies in place?

## PodSecurityPolicy Best Practices

PodSecurityPolicy is complex and can be error prone. Refer to the following best practices before implementing PodSecurityPolicy on your clusters:

- It all comes down to RBAC. Whether you like it or not, PodSecurityPolicy is determined by RBAC. It's this relationship that actually exposes all of the shortcomings in your current RBAC policy design. We cannot stress just how important it is to automate your RBAC and PodSecurityPolicy creation and maintenance. Specifically locking down access to service accounts is the key to using policy.
- Understand the policy scope. Determining how your policies will be laid out on your cluster is very important. Your policies can be cluster-wide, namespaced, or workload-specific in scope. There will always be workloads on your cluster that are part of the Kubernetes cluster operations that will need more permissive security privileges, so make sure that you have appropriate RBAC in place to stop unwanted workloads using your permissive policies.
- Do you want to enable PodSecurityPolicy on an existing cluster? Use this handy open source tool to generate policies based on your current resources. This is a great start. From there, you can hone your policies.

### PodSecurityPolicy Next Steps

As demonstrated, PodSecurityPolicy is an extremely powerful API to assist in keeping your cluster secure, but it demands a high tax for use. With careful planning and a pragmatic approach, PodSecurityPolicy can be successfully implemented on any cluster. At the very least, it will keep your security team happy.

## Workload Isolation and RuntimeClass

Container runtimes are still largely considered an insecure workload isolation boundary. There is no clear path to whether the most common runtimes of today will ever be recognized as secure. The momentum and interest among those in the industry toward Kubernetes has led to the development of different container runtimes that offer varying levels of isolation. Some are based on familiar and trusted technology stacks, whereas others are a completely new attempt to tackle the problem. Open source projects like Kata containers, gVisor, and Firecracker tout the promise of stronger workload isolation. These specific projects are either based on nested virtualization (running a super lightweight virtual machine within a virtual machine) or system call filtering and servicing.

The introduction of these container runtimes that offer different workload isolation allows users to choose many different runtimes based on their isolation guarantees in the same cluster. For example, you could have trusted and untrusted workloads running in the same cluster in different container runtimes.

RuntimeClass was introduced into Kubernetes as an API to allow container runtime selection. It is used to represent one of the supported container runtimes on the cluster when it has been configured by the cluster administrator. As a Kubernetes user, you can define specific runtime classes for your workloads by using the RuntimeClassName in the pod specification. How this is implemented under the hood is that the RuntimeClass designates a `RuntimeHandler` which is passed to the Container Runtime Interface (CRI) to implement. Node labeling or node taints then can be used in conjunction with nodeSelectors or tolerations to ensure that the workload lands on a node capable of supporting the desired RuntimeClass. Figure 10-1 demonstrates how a kubelet uses RuntimeClass when launching pods.

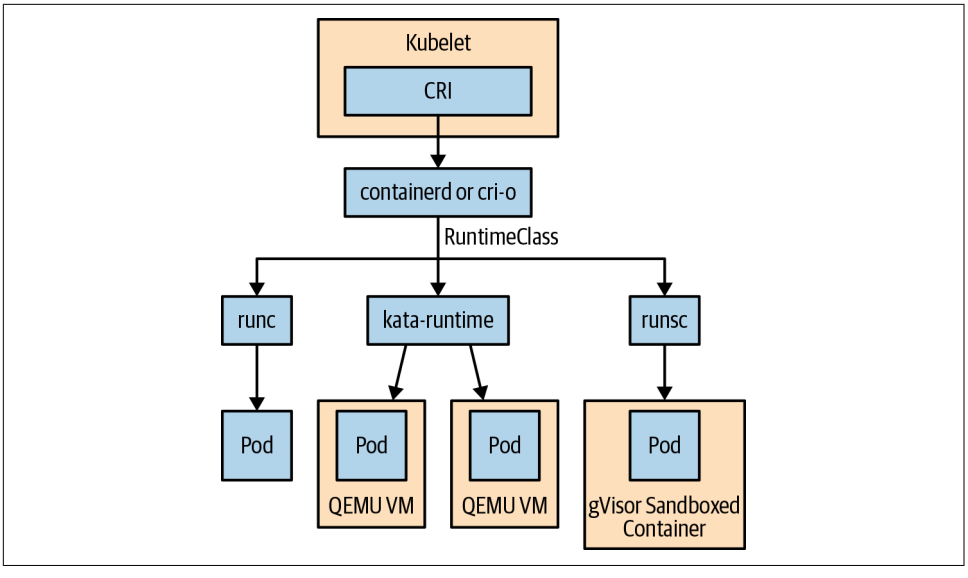



*Figure 10-1. `RuntimeClass` flow diagram*

The `RuntimeClass` API is under active development. For the latest updates on the feature state, visit the upstream documentation.

## Using RuntimeClass

If a cluster administrator has set up different RuntimeClasses, you can use them simply by specifying `runtimeClassName` in the pod specification; for example:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  runtimeClassName: firecracker
```

## Runtime Implementations

Following are some open source container runtime implementations that offer different levels of security and isolation for your consideration. This list is intended as a guide and is by no means exhaustive:

*CRI containerd*
: An API facade for container runtimes with an emphasis on simplicity, robustness, and portability.

*cri-o*
: A purpose-built, lightweight Open Container Initiative (OCI)-based implementation of a container runtime for Kubernetes.

*Firecracker*
: Built on top of the Kernel-based Virtual Machine (KVM), this virtualization technology allows you to launch microVMs in nonvirtualized environments very quickly using the security and isolation of traditional VMs.

*gVisor*
: An OCI-compatible sandbox runtime that runs containers with a new user-space kernel, which provides a low overhead, secure, isolated container runtime.

*Kata Containers*
: A community that's building a secure container runtime that provides VM-like security and isolation by running lightweight VMs that feel and operate like containers.

## Workload Isolation and RuntimeClass Best Practices

The following best practices will help you to avoid common workload isolation and RuntimeClass pitfalls:

- Implementing different workload isolation environments via RuntimeClass will complicate your operational environment. This means that workloads might not be portable across different container runtimes given the nature of the isolation they provide. Understanding the matrix of supported features across different runtimes can be complicated to understand and will lead to poor user experience. We recommend having separate clusters, each with a single runtime to avoid confusion, if possible.
- Workload isolation doesn't mean secure multitenancy. Even though you might have implemented a secure container runtime, this doesn't mean that the Kubernetes cluster and APIs have been secured in the same fashion. You must consider the total surface area of Kubernetes end to end. Just because you have an isolated workload doesn't mean that it cannot be modified by a bad actor via the Kubernetes API.
- Tooling across different runtimes is inconsistent. You might have users who rely on container runtime tooling for debugging and introspection. Having different runtimes means that you might no longer be able to run `docker ps` to list running containers. This leads to confusion and complications when troubleshooting.

# Other Pod and Container Security Considerations

In addition to PodSecurityPolicy and workload isolation, here are some other tools you may consider when determining how to handle pod and container security.

## Admission Controllers

If you're worried about diving into the deep end with PodSecurityPolicy, here are some options that offer a fraction of the functionality but might offer a viable alternative. You can use admission controllers such as `DenyExecOnPrivileged` and `DenyEscalatingExec` in conjunction with an admission webhook to add SecurityContext workload settings to achieve a similar outcome. For more information on admission control, refer to Chapter 17.

## Intrusion and Anomaly Detection Tooling

We've covered security policies and container runtimes, but what happens when you want to introspect and enforce policy within the container runtime? There are open source tools that can do this and more. They operate by either listening and filtering Linux system calls or by utilizing a Berkeley Packet Filter (BPF). One such tool is Falco. Falco is a Cloud Native Computing Foundation (CNCF) project that simply installs as a Demonset and allows you to configure and enforce policy during execution. Falco is just one approach. We encourage you to take a look at the tooling in this space to see what works for you.

# Summary

In this chapter, we covered in depth both the PodSecurityPolicy and the RuntimeClass APIs with which you can configure a granular level of security for your workloads. We have also taken a look at some open source ecosystem tooling that you can use to monitor and enforce policy within the container runtime. We have provided a thorough overview for you to make an informed decision about providing the level of security that is best suited for your workload needs.

CHAPTER 11

# Policy and Governance for Your Cluster

Have you ever wondered how you can ensure that all containers running on a cluster come only from an approved container registry? Or maybe you've been asked to ensure that services are never exposed to the internet. These are precisely the problems that policy and governance for your cluster set out to answer. As Kubernetes matures and becomes adopted by more and more enterprises, the question of policy and governance is becoming increasingly frequent. Although this area is still relatively new and upcoming, in this chapter we share what you can do to make sure that your cluster is in compliance with the defined policies of your enterprise.

## Why Policy and Governance Are Important

Whether you operate in a highly regulated environment—for example, health care or financial services—or you simply want to make sure that you maintain a level of control over what's running on your clusters, you're going to need a way to implement the stated policies of the enterprise. After these policies are defined, you will need to determine how to implement policy and maintain clusters that are compliant to these policies. These policies might be in place to meet regulatory compliance or simply to enforce best practices. Whatever the reason, you must be sure that you do not sacrifice developer agility and self-service when implementing these policies.

## How Is This Policy Different?

In Kubernetes, policy is everywhere. Whether it be network policy or pod security policy, we've all come to understand what policy is and when to use it. We trust that whatever is declared in Kubernetes resource specifications is implemented as per the policy definition. Both network policy and pod security policy are implemented at runtime. However, who manages the content that is actually defined in these

Kubernetes resource specifications? That's the job for policy and governance. Rather than implementing policy at runtime, when we talk about policy in the context of governance, what we mean is defining policy that controls the fields and values in the Kubernetes resource specifications themselves. Only Kubernetes resource specifications that are compliant against these policies are allowed and committed to the cluster state.

## Cloud-Native Policy Engine

To be able to make decisions about what resources are compliant, we need a policy engine that is flexible enough to meet a variety of needs. The Open Policy Agent (OPA) is an open source, flexible, lightweight policy engine that has become increasingly popular in the cloud-native ecosystem. Having OPA in the ecosystem has allowed many implementations of different Kubernetes governance tools to appear. One such Kubernetes policy and governance project the community is rallying around is called Gatekeeper. For the rest of this chapter, we use Gatekeeper as the canonical example to illustrate how you might achieve policy and governance for your cluster. Although there are other implementations of policy and governance tools in the ecosystem, they all seek to provide the same user experience (UX) by allowing only compliant Kubernetes resource specifications to be committed to the cluster.

## Introducing Gatekeeper

Gatekeeper is an open source customizable Kubernetes admission webhook for cluster policy and governance. Gatekeeper takes advantage of the OPA constraint framework to enforce custom resource definition (CRD)-based policies. Using CRDs allows for an integrated Kubernetes experience that decouples policy authoring from implementation. Policy templates are referred to as *constraint templates*, which can be shared and reused across clusters. Gatekeeper enables resource validation and audit functionality. One of the great things about Gatekeeper is that it's portable, which means that you can implement it on any Kubernetes clusters, and if you are already using OPA, you might be able to port that policy over to Gatekeeper.

Gatekeeper is still under active development and is subject to change. For the most recent updates on the project, visit the official upstream repository.

## Example Policies

It's important not to become too stuck in the weeds and actually consider the problem that we are trying to solve. Let's take a look at some policies that solve some of the most common compliance issues for context:

- Services must not be exposed publicly on the internet.
- Allow containers only from trusted container registries.
- All containers must have resource limits.
- Ingress hostnames must not overlap.
- Ingresses must use only HTTPS.

## Gatekeeper Terminology

Gatekeeper has adopted much of the same terminology as OPA. It's important that we cover what that terminology is so that you can understand how Gatekeeper operates. Gatekeeper uses the OPA constraint framework. Here, we introduce three new terms:

- Constraint
- Rego
- Constraint template

### Constraint

The best way to think about constraints is as restrictions that you apply to specific fields and values of Kubernetes resource specifications. This is really just a long way of saying policy. This means that when constraints are defined, you are effectively stating that you *DO NOT* want to allow this. The implications of this approach mean that resources are implicitly allowed without a constraint that issues a deny. This is important because instead of allowing the Kubernetes resources specification fields and values you want, you are denying only the ones you do not want. This architectural decision suits Kubernetes resource specifications nicely because they are ever changing.

### Rego

Rego is an OPA-native query language. Rego queries are assertions on the data stored in OPA. Gatekeeper stores rego in the constraint template.

### Constraint template

You can think of this as a policy template. It's portable and reusable. Constraint templates consist of typed parameters and the target rego that is parameterized for reuse.

## Defining Constraint Templates

Constraint templates are a Custom Resource Definition (CRD) that provide a means of templating policy so that it can be shared or reused. In addition, parameters for the policy can be validated. Let's take a look at a constraint template in the context of the earlier examples. In the following example, we share a constraint template that provides the policy "Only allow containers from trusted container registries":

```
apiVersion: templates.gatekeeper.sh/v1alpha1
kind: ConstraintTemplate
metadata:
  name: k8sallowedrepos
spec:
  crd:
    spec:
      names:
        kind: K8sAllowedRepos
        listKind: K8sAllowedReposList
        plural: k8sallowedrepos
        singular: k8sallowedrepos
      validation:
        # Schema for the `parameters` field
        openAPIV3Schema:
          properties:
            repos:
              type: array
              items:
                type: string
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8sallowedrepos

        deny[{"msg": msg}] {
          container := input.review.object.spec.containers[_]
          satisfied := [good | repo = input.constraint.spec.parame
ters.repos[_] ; good = startswith(container.image, repo)]
          not any(satisfied)
          msg := sprintf("container <%v> has an invalid image repo <%v>,
allowed repos are %v", [container.name, container.image, input.con
straint.spec.parameters.repos])
        }
```

The constraint template consists of three main components:

*Kubernetes-required CRD metadata*
: The name is the most important part. We reference this later.

*Schema for input parameters*
: Indicated by the validation field, this section defines the input parameters and their associated types. In this example, we have a single parameter called `repo` that is an array of strings.

*Policy definition*
: Indicated by the `target` field, this section contains templated rego (the language to define policy in OPA). Using a constraint template allows the templated rego to be reused and means that generic policy can be shared. If the rule matches, the constraint is violated.

## Defining Constraints

To use the previous constraint template, we must create a constraint resource. The purpose of the constraint resource is to provide the necessary parameters to the constraint template that we created earlier. You can see that the `kind` of the resource defined in the following example is `K8sAllowedRepos`, which maps to the constraint template defined in the previous section:

```
apiVersion: constraints.gatekeeper.sh/v1alpha1
kind: K8sAllowedRepos
metadata:
  name: prod-repo-is-openpolicyagent
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
    namespaces:
      - "production"
  parameters:
    repos:
      - "openpolicyagent"
```

The constraint consists of two main sections:

*Kubernetes metadata*
: Notice that this constraint is of `kind K8sAllowedRepos`, which matches the name of the constraint template.

*The spec*
: The `match` field defines the scope of intent for the policy. In this example, we are matching pods only in the production namespace.

  The parameters define the intent for the policy. Notice that they match the type from the constraint template schema from the previous section. In this case, we allow only container images that start with `openpolicyagent`.

Constraints have the following operational characteristics:

- Logically AND-ed together
  - When multiple policies validate the same field, if one violates then the whole request is rejected
- Schema validation that allows early error detection
- Selection criteria
  - Can use label selectors
  - Constrain only certain kinds
  - Constrain only in certain namespaces

## Data Replication

In some cases, you might want to compare the current resource against other resources that are in the cluster, for example, in the case of "Ingress hostnames must not overlap." OPA needs to have all of the other Ingress resources in its cache in order to evaluate the rule. Gatekeeper uses a `config` resource to manage which data is cached in OPA in order to perform evaluations such as the one previously mentioned. In addition, `config` resources are also used in the audit functionality, which we explore a bit later on.

The following example `config` resource caches v1 service, pods, and namespaces:

```
apiVersion: config.gatekeeper.sh/v1alpha1
kind: Config
metadata:
name: config
  namespace: gatekeeper-system
spec:
  sync:
    syncOnly:
    - kind: Service
      version: v1
    - kind: Pod
      version: v1
    - kind: Namespace
      version: v1
```

## UX

Gatekeeper enables real-time feedback to cluster users for resources that violate defined policy. If we consider the example from the previous sections, we allow containers only from repositories that start with `openpolicyagent`.

Let's try to create the following resource; it is not compliant given the current policy:

```
apiVersion: v1
kind: Pod
metadata:
  name: opa
  namespace: production
spec:
  containers:
    - name: opa
      image: quay.io/opa:0.9.2
```

This gives you the violation message that's defined in the constraint template:

```
$ kubectl create -f bad_resources/opa_wrong_repo.yaml
Error from server (container <opa> has an invalid image repo <quay.io/opa:
0.9.2>, allowed repos are ["openpolicyagent"]): error when creating "bad_resour
ces/opa_wrong_repo.yaml": admission webhook "validation.gatekeeper.sh" denied
the request: container <opa> has an invalid image repo <quay.io/opa:0.9.2>,
allowed repos are ["openpolicyagent"]
```

# Audit

Thus far, we have discussed only how to define policy and have it enforced as part of the request admission process. How do you handle a cluster that already has resources deployed where you want to know what is in compliance with the defined policy? That is exactly what audit sets out to achieve. When using audit, Gatekeeper periodically evaluates resources against the defined constraints. This helps with the detection of misconfigured resources according to policy and allows for remediation. The audit results are stored in the status field of the constraint, making them easy to find by simply using `kubectl`. To use audit, the resources to be audited must be replicated. For more details, refer to "Data Replication" on page 164.

Let's take a look at the constraint called `prod-repo-is-openpolicyagent` that you defined in the previous section:

```
$ kubectl get k8sallowedrepos prod-repo-is-openpolicyagent -o yaml
apiVersion: constraints.gatekeeper.sh/v1alpha1
kind: K8sAllowedRepos
metadata:
  creationTimestamp: "2019-06-04T06:05:05Z"
  finalizers:
  - finalizers.gatekeeper.sh/constraint
  generation: 2820
  name: prod-repo-is-openpolicyagent
  resourceVersion: "4075433"
  selfLink: /apis/constraints.gatekeeper.sh/v1alpha1/k8sallowedrepos/prod-repo-
is-openpolicyagent
  uid: b291e054-868e-11e9-868d-000d3afdb27e
spec:
  match:
    kinds:
```

```
    - apiGroups:
      - ""
      kinds:
      - Pod
    namespaces:
    - production
  parameters:
    repos:
    - openpolicyagent
status:
  auditTimestamp: "2019-06-05T05:51:16Z"
  enforced: true
  violations:
  - kind: Pod
    message: container <nginx> has an invalid image repo <nginx>, allowed repos
are
      ["openpolicyagent"]
    name: nginx
    namespace: production
```

Upon inspection, you can see the last time the audit ran in the `auditTimestamp` field. We also see all of the resources that violate this constraint under the `violations` field.

## Becoming Familiar with Gatekeeper

The Gatekeeper repository ships with fantastic demonstration content that walks you through a detailed example of building policies to meet compliance for a bank. We would strongly recommend walking through the demonstration for a hands-on approach to how Gatekeeper operates. You can find the demonstration in this Git repository.

## Gatekeeper Next Steps

The Gatekeeper project is continuing to grow and is looking to solve other problems in the areas of policy and governance, which includes features like these:

- Mutation (modifying resources based on policy; for example, add these labels)
- External data sources (integration with Lightweight Directory Access Protocol [LDAP] or Active Directory for policy lookup)
- Authorization (using Gatekeeper as a Kubernetes authorization module)
- Dry run (allow users to test policy before making it active in a cluster)

If these sound like interesting problems that you might be willing to help solve, the Gatekeeper community is always looking for new users and contributors to help shape the future of the project. If you would like to learn more, head over to the upstream repository on GitHub.

## Policy and Governance Best Practices

You should consider the following best practices when implementing policy and governance on your clusters:

- If you want to enforce a specific field in a pod, you need to make a determination of which Kubernetes resource specification you want to inspect and enforce. Let's consider the case of Deployments, for example. Deployments manage ReplicaSets, which manage pods. We could enforce at all three levels, but the best choice is the one that is the lowest handoff point before the runtime, which in this case is the pod. This decision, however, has implications. The user-friendly error message when we try to deploy a noncompliant pod, as seen in "UX" on page 164, is not going to be displayed. This is because the user is not creating the noncompliant resource, the ReplicaSet is. This experience means that the user would need to determine that the resource is not compliant by running a `kubectl describe` on the current ReplicaSet associated with the Deployment. Although this might seem cumbersome, this is consistent behavior with other Kubernetes features, such as pod security policy.
- Constraints can be applied to Kubernetes resources on the following criteria: kinds, namespaces, and label selectors. We would strongly recommend scoping the constraint to the resources to which you want it to be applied as tightly as possible. This ensures consistent policy behavior as the resources on the cluster grow, and means that resources that don't need to be evaluated aren't being passed to OPA, which can result in other inefficiencies.
- Synchronizing and enforcing on potentially sensitive data such as Kubernetes secrets is *not* recommended. Given that OPA will hold this in its cache (if it is configured to replicate that data) and resources will be passed to Gatekeeper, it leaves surface area for a potential attack vector.
- If you have many constraints defined, a deny of constraint means that the entire request is denied. There is no way to make this function as a logical OR.

## Summary

In this chapter, we covered why policy and governance are important and walked through a project that's built upon OPA, a cloud-native ecosystem policy engine, to provide a Kubernetes-native approach to policy and governance. You should now be prepared and confident the next time the security teams asks, "Are our clusters in compliance with our defined policy?"

CHAPTER 12

# Managing Multiple Clusters

In this chapter, we discuss best practices for managing multiple Kubernetes clusters. We dive into the details of the differences between multicluster management and federation, tools to manage multiple clusters, and operational patterns for managing multiple clusters.

You might wonder why you would need multiple Kubernetes clusters; Kubernetes was built to consolidate many workloads to a single cluster, correct? This is true, but there are scenarios such as workloads across regions, concerns of blast radius, regulatory compliance, and specialized workloads.

We discuss these scenarios and explore the tools and techniques for managing multiple clusters in Kubernetes.

## Why Multiple Clusters?

When adopting Kubernetes, you will likely have more than one cluster, and you might even start with more than one cluster to break out production from staging, user acceptance testing (UAT), or development. Kubernetes provides some multitenancy features with namespaces, which are a logical way to break up a cluster into smaller logical constructs. Namespaces allow you to define Role-Based Access Control (RBAC), quotas, pod security policies, and network policies to allow separation of workloads. This is a great way to separate out multiple teams and projects, but there are other concerns that might require you to build a multicluster architecture. Following are concerns to think about when deciding to use multicluster versus a single-cluster architecture:

- Blast radius
- Compliance

- Security
- Hard multitenancy
- Regional-based workloads
- Specialized workloads

When thinking through your architecture, *blast radius* should come front and center. This is one of the main concerns that we see with users designing for multicluster architectures. With microservice architectures we employ circuit breakers, retries, bulkheads, and rate limiting to constrain the extent of damage to our systems. You should design the same into your infrastructure layer, and multiple clusters can help with preventing the impact of cascading failures due to software issues. For example, if you have one cluster that serves 500 applications and you have a platform issue, it takes out 100% of the 500 applications. If you had a platform layer issue with 5 clusters serving those 500 applications, you affect only 20% of the applications. The downside to this is that now you need to manage five clusters, and your consolidation ratios will not be as good with a single cluster. Dan Woods wrote a great article about an actual cascading failure in a production Kubernetes environment. It is a great example of why you will want to consider multicluster architectures for larger environments.

*Compliance* is another area of concern for multicluster design because there are special considerations for Payment Card Industry (PCI), Health Insurance Portability and Accountability (HIPAA), and other workloads. It's not that Kubernetes doesn't provide some multitenant features, but these workloads might be easier to manage if they are segregated out from general purpose workloads. These compliant workloads might have specific requirements with respect to security hardening, nonshared components, or dedicated workload requirements. It's just much easier to separate these workloads than have to treat the cluster in such a specialized fashion.

*Security* in large Kubernetes clusters can become difficult to manage. As you start onboarding more and more teams to a Kubernetes cluster each team may have different security requirements and it can become very difficult to meet those needs in a large multi-tenant cluster. Even just managing RBAC, network policies, and pod security policies can become difficult at scale in a single cluster. A small change to a network policy can inadvertently open up security risk to other users of the cluster. With multiple clusters you can limit the security impact with a misconfiguration. If you decide that a larger Kubernetes cluster fits your requirements, then ensure that you have a very good operational process for making security changes and understand the blast radius of making a change to RBAC, network policy, and pod security policies.

Kubernetes doesn't provide *hard multitenancy* because it shares the same API boundary with all workloads running within the cluster. With namespacing this gives us

good soft multitenancy, but not enough to protect against hostile workloads within the cluster. Hard multitenancy is not a requirement for a lot of users; they trust the workloads that will be running within the cluster. Hard multitenancy is typically a requirement if you are a cloud provider, hosting Software as a Service (SaaS)-based software or untrusted workloads with untrusted user control.

When running workloads that need to serve traffic from in-region endpoints, your design will include multiple clusters that are based per region. When you have a globally distributed application, it becomes a requirement at that point to run multiple clusters. When you have workloads that need to be *regionally distributed*, it's a great use case for cluster federation of multiple clusters, which we dig into further later in this chapter.

*Specialized workloads*, such as high-performance computing (HPC), machine learning (ML), and grid computing, also need to be addressed in the multicluster architecture. These types of specialized workloads might require specific types of hardware, have unique performance profiles, and have specialized users of the clusters. We've seen this use case to be less prevalent in the design decision because having multiple Kubernetes node pools can help address specialized hardware and performance profiles. When you have the need for a very large cluster for an HPC or machine learning workload, you should take into consideration just dedicating clusters for these workloads.

With multicluster, you get isolation for "free," but it also has design concerns that you need to address at the outset.

## Multicluster Design Concerns

When choosing a multicluster design there are some challenges that you'll run into. Some of these challenges might deter you from attempting a multicluster design given that the design might overcomplicate your architecture. Some of the common challenges we find users running into are:

- Data replication
- Service discovery
- Network routing
- Operational management
- Continuous deployment

*Data replication* and consistency has always been the crux of deploying workloads across geographical regions and multiple clusters. When running these services, you need to decide what runs where and develop a replication strategy. Most databases have built-in tools to perform the replication, but you need to design the application

to be able to handle the replication strategy. For NoSQL-type database services this can be easier because they can can handle scaling across multiple instances, but you still need to ensure that your application can handle eventual consistency across geographic regions or at least the latency across regions. Some cloud services, such as Google Cloud Spanner and Microsoft Azure CosmosDB, have built database services to help with the complications of handling data across multiple geographic regions.

Each Kubernetes cluster deploys its own *service discovery* registry, and registries are not synchronized across multiple clusters. This complicates applications being able to easily identify and discover one another. Tools such as HashiCorp's Consul can transparently synchronize services from multiple clusters and even services that reside outside of Kubernetes. There are other tools like Istio, Linkerd, and Cillium that are building on multiple cluster architectures to extend service discovery between clusters.

Kubernetes makes networking from within the cluster very easy, as it's a flat network and avoids using network address translation (NAT). If you need to route traffic in and out of the cluster, this becomes more complicated. Ingress into the cluster is implemented as a 1:1 mapping of ingress to the cluster because it doesn't support multicluster topologies with the Ingress resource. You'll also need to consider the egress traffic between clusters and how to route that traffic. When your applications reside within a single cluster this is easy, but when introducing multicluster, you need to think about the latency of extra hops for services that have application dependencies in another cluster. For applications that have tightly coupled dependencies, you should consider running these services within the same cluster to remove latency and extra complexity.

One of the biggest overheads to managing multiclusters is the *operational management*. Instead of one or a couple of clusters to manage and keep consistent, you might now have many clusters to manage in your environment. One of the most important aspects to managing multiclusters is ensuring that you have good automation practices in place because this will help to reduce the operational burden. When automating your clusters, you need to take into account the infrastructure deployment and managing add-on features to your clusters. For managing the infrastructure, using a tool like HashioCrp's Terraform can help with deploying and managing a consistent state across your fleet of clusters.

Using an *Infrastructure as Code* (IaC) tool like Terraform will give you the benefit of providing a reproducible way to deploy your clusters. On the other hand, you also need to be able to consistently manage add-ons to the cluster, such as monitoring, logging, ingress, security, and other tools. Security is also an important aspect of operational management, and you must be able to maintain security policies, RBAC, and network policies across clusters. Later in this chapter, we dive deeper into the topic of maintaining consistent clusters with automation.

With multiple clusters and *Continuous Delivery* (CD), you now need to deal with multiple Kubernetes API endpoints versus a single API endpoint. This can cause challenges in the distribution of applications. You can easily manage multiple pipelines, but suppose that you have a hundred different pipelines to manage, which can make application distribution very difficult. With this in mind, you need to look at different approaches to managing this situation. We take a look at solutions to help manage this later in the chapter.

# Managing Multiple Cluster Deployments

One of the first steps that you want to take when managing multicluster deployments is to use an IoC tool like Terraform to set up deployments. Other deployment tools, such as kubespray, kops, or other cloud provider–specific tools, are all valid choices but, most importantly, use a tool that allows you to source control your cluster deployment for repeatability.

Automation is key to successfully managing multiple clusters in your environment. You might not have everything automated on day one, but you should make it a priority to automate all aspects of your cluster deployments and operations.

An interesting project in development is the Kubernetes Cluster API. The Cluster API is a Kubernetes project to bring declarative, Kubernetes-style APIs to cluster creation, configuration, and management. It provides optional, additive functionality on top of core Kubernetes. The Cluster API provides a cluster-level configuration declared through a common API, which will give you the ability to easily automate and build tooling around cluster automation. As of this writing, the project is still in development, so make sure to keep an eye out for it as it matures.

## Deployment and Management Patterns

Kubernetes operators were introduced as an implementation of the *Infrastructure as Software* concept. Using them allows you to abstract the deployment of applications and services in a Kubernetes cluster. For example, suppose that you want to standardize on Prometheus for monitoring your Kubernetes clusters. You would need to create and manage various objects (deployment, service, ingress, etc.) for each cluster and team. You would also need to maintain the fundamental configurations of Prometheus, such as versions, persistence, retention policies, and replicas. As you can imagine, the maintenance of such a solution could be difficult across a large number of clusters and teams.

Instead of dealing with so many objects and configurations, you could install the `prometheus-operator`. This extends the Kubernetes API, exposing multiple new object kinds called `Prometheus`, `ServiceMonitor`, `PrometheusRule`, and `AlertManager`, which allow you to specify all of the details of a Prometheus deployment using

just a few objects. You can use the `kubectl` tool to manage such objects, just as it manages any other Kubernetes API object.

Figure 12-1 shows the architecture of the `prometheus-operator`.

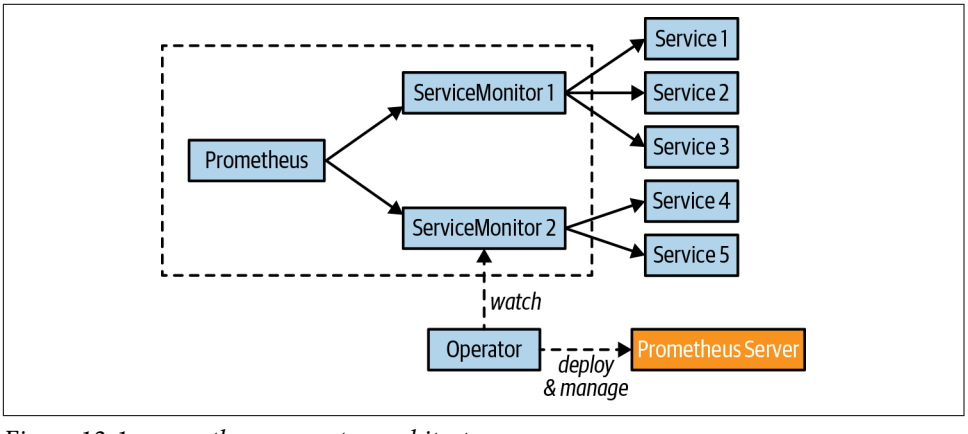



*Figure 12-1. prometheus-operator architecture*

Utilizing the *Operator* pattern for automating key operational tasks can help improve your overall cluster management capabilities. The Operator pattern was introduced by the CoreOS team in 2016 with the etcd operator and `prometheus-operator`. The Operator pattern builds on two concepts:

- Custom resource definitions
- Custom controllers

*Custom resource definitions* (CRDs) are objects that allow you to extend the Kubernetes API, based on your own API that you define.

*Custom controllers* are built on the core Kubernetes concepts of resources and controllers. Custom controllers allow you to build your own logic by watching events from Kubernetes API objects such as namespaces, Deployments, pods, or your own CRD. With custom controllers, you can build your CRDs in a declarative way. If you consider how the Kubernetes Deployment controller works in a reconciliation loop to always maintain the state of the deployment object to maintain its declarative state, this brings the same advantages of controllers to your CRDs.

When utilizing the Operator pattern, you can build in automation to operational tasks that need to be performed on operational tooling in multiclusters. Let's take the following Elasticsearch operator as an example. As in Chapter 3, we utilized the Elasticsearch, Logstash, and Kibana (ELK) stack to perform log aggregation of our cluster. The Elasticsearch operator can perform the following operations:

- Replicas for master, client, and data nodes
- Zones for highly available deployments
- Volume sizes for master and data nodes
- Resizing of cluster
- Snapshot for backups of the Elasticsearch cluster

As you can see, the operator provides automation for many tasks that you would need to perform when managing Elasticsearch, such as automating snapshots for backup and resizing the cluster. The beauty of this is that you manage all of this through familiar Kubernetes objects.

Think about how you can take advantage of different operators like the `prometheus-operator` in your environment and also how you can build your own custom operator to offload common operational tasks.

## The GitOps Approach to Managing Clusters

*GitOps* was popularized by the folks at Weaveworks, and the idea and fundamentals were based on their experience of running Kubernetes in production. GitOps takes the concepts of the software development life cycle and applies them to operations. With GitOps, your Git repository becomes your source of truth, and your cluster is synchronized to the configured Git repository. For example, if you update a Kubernetes Deployment manifest, those configuration changes are automatically reflected in the cluster state.

By using this method, you can make it easier to maintain multiclusters that are consistent and avoid configuration drift across the fleet. GitOps allows you to declaratively describe your clusters for multiple environments and drives to maintain that state for the cluster. The practice of GitOps can apply to both application delivery and operations, but in this chapter, we focus on using it to manage clusters and operational tooling.

Weaveworks Flux was one of the first tools to enable the GitOps approach, and it's the tool we will use throughout the rest of the chapter. There are many new tools that have been released into the cloud-native ecosystem that are worth a look, such as Argo CD, from the folks at Intuit, which has also been widely adopted for the GitOps approach.

Figure 12-2 presents a representation of a GitOps workflow.

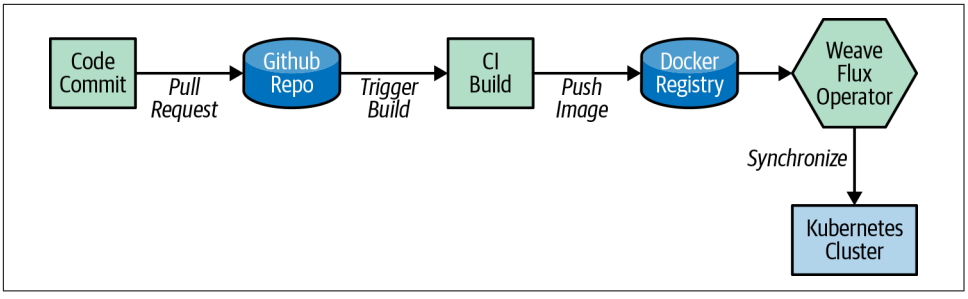



*Figure 12-2. GitOps workflow*

So, let's get Flux set up in your cluster and get a repository synchronized to the cluster:

```
git clone https://github.com/weaveworks/flux
cd flux
```

You now need to make a change to the Deployment manifest to configure it with your forked repo from Chapter 6. Modify the following line in the Deployment file to match your forked GitHub repository:

```
vim deploy/flux-deployment.yaml
```

Modify the following line with your Git repository:

```
--git-url=git@github.com:weaveworks/flux-get-started  (ex. --git-
url=git@github.com:your_repo/kbp )
```

Now, go ahead and deploy Flux to your cluster:

```
kubectl apply -f deploy
```

When Flux installs, it creates an SSH key so that it can authenticate with the Git repository. Use the Flux command-line tool to retrieve the SSH key so that you can configure access to your forked repository; first, you need to install `fluxctl`.

For MacOS:

```
brew install fluxctl
```

For Linux Snap Packages:

```
snap install fluxctl
```

For all other packages, you can find the latest binaries here:

```
fluxctl identity
```

Open GitHub, navigate to your fork, go to Setting > "Deploy keys," click "Add deploy key," give it a Title, select the "Allow write access" checkbox, paste the Flux public key, and then click "Add key." See the GitHub documentation for more information on how to manage deploy keys.

Now, if you view the Flux logs, you should see that it is synchronizing with your GitHub repository:

```
kubectl -n default logs deployment/flux -f
```

After you see that it's synchronizing with your GitHub repository, you should see that the Elasticsearch, Prometheus, Redis, and frontend pods are created:

```
kubectl get pods -w
```

With this example complete, you should be able to see how easy it is for you to synchronize your GitHub repository state with your Kubernetes cluster. This makes managing the multiple operational tools in your cluster much easier, because multiple clusters can synchronize with a single repository and remove the situation of having snowflake clusters.

## Multicluster Management Tools

When working with multiple clusters, using Kubectl can immediately become confusing because you need to set different contexts to manage the different clusters. Two tools that you will want to install right away when dealing with multiple clusters are *kubectx* and *kubens*, which allow you to easily change between multiple contexts and namespaces.

When you need a full-fleged multicluster management tool, there are a few within the Kubernetes ecosystem to look at for managing multiple clusters. Following is a summary of some of the more popular tools:

- *Rancher* centrally manages multiple Kubernetes clusters in a centrally managed user interface (UI). It monitors, manages, backs up, and restores Kubernetes clusters across on-premises, cloud, and hosted Kubernetes setups. It also has tools for controlling applications deployed across multiple clusters and provides operational tooling.
- *KQueen* provides a multitenant self-service portal for Kubernetes cluster provisioning and focuses on auditing, visibility, and security of multiple Kubernetes clusters. KQueen is an open source project that was developed by the folks at Mirantis.
- *Gardener* takes a different approach to multicluster management in that it utilizes Kubernetes primitives to provide Kubernetes as a Service to your end users. It provides support for all major cloud vendors and was developed by the folks at SAP. This solution is really geared toward users who are building a Kubernetes as a Service offering.

## Kubernetes Federation

Kubernetes first introduced Federation v1 in Kubernetes 1.3, and it has since been deprecated in lieu of Federation v2. Federation v1 set out to help with the distribution of applications to multiple clusters. Federation v1 was built utilizing the Kubernetes API and heavily relied on Kubernetes annotations, which imposed some problems in its design. The design was tightly coupled to the core Kubernetes API, which made Federation v1 quite monolithic in nature. At the time, the design decisions were probably not bad choices, but were built on the primitives that were available. The introducton of Kubernetes CRDs allowed a different way of thinking about how Federation could be designed.

Federation v2 (now called *KubeFed*) requires Kubernetes 1.11+ and is currently in alpha as of this writing. Federation v2 is built around the concept of CRDs and custom controllers, which allows you to extend Kubernetes with new APIs. Building around CRDs allows Federation to have new API types and not be restricted just to previous v1 deployment objects.

KubeFed is not necessarily about multicluster management, but providing high availability (HA) deployments across multiple clusters. It allows you to combine multiple clusters into a single management endpoint for delivering applications on Kubernetes. For example, if you have a cluster that resides in multiple public cloud environments, you can combine these clusters into a single control plane to manage deployments to all clusters to increase the resiliency of your application.

As of this writing, the following Federated resources are supported:

- Namespaces
- ConfigMaps
- Secrets
- Ingress
- Services
- Deployments
- ReplicaSets
- Horizontal Pod Autoscalers
- DaemonSets
- Jobs

To understand how this all works, let's first take a look at the architecture in Figure 12-3.

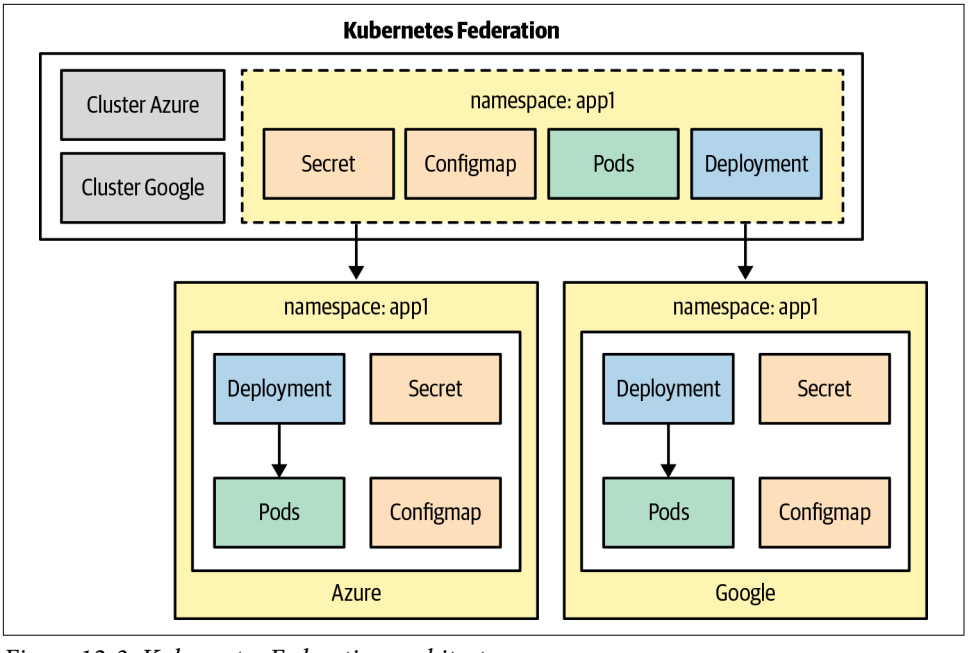



*Figure 12-3. Kubernetes Federation architecture*

It's important to understand that with Federation, not everything is just copied down to all clusters. For example, with Deployments and ReplicaSets, you define the number of replicas, which are then spread out across the clusters. This is the default for Deployments, but you can change the configuration. On the other hand, if you create a namespace, that namespace is cluster scoped and created in each cluster. Secrets, ConfigMaps, and DaemonSets work the same way and are copied down to each cluster. The Ingress resource is also different from the aforementioned objects because it creates a global multicluster resource with a single entry point into a service. As you can see from how KubeFed works, the use cases Kubefed supports are multiregion, multicloud, and global application deployments to Kubernetes.

Following is an example of a federated Deployment:

```
apiVersion: types.kubefed.io/v1beta1
kind: FederatedDeployment
metadata:
  name: test-deployment
  namespace: test-namespace
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      replicas: 5
```

```
      selector:
        matchLabels:
          app: nginx
      template:
        metadata:
          labels:
            app: nginx
        spec:
          containers:
          - image: nginx
            name: nginx
  placement:
    clusters:
    - name: azure
    - name: google
```

This example creates a federated Deployment of an NGINX pod with five replicas, which are then spread across our clusters in Azure and another cluster in Google.

Setting up federated Kubernetes clusters is beyond the scope of this book, but you can learn more about the subject by referring to the KubeFed User Guide.

KubeFed is still in alpha, so keep an eye on it, but embrace the tools that you already have or can implement now so that you can be successful with Kubernetes HA and multicluster deployments.

## Managing Multiple Clusters Best Practices

Consider the following best practices when managing multiple Kubernetes clusters:

- Limit the blast radius of your clusters to ensure cascading failures don't have a bigger impact on your applications.
- If you have regulatory concerns such as PCI, HIPPA, or HiTrust, think about utilizing multiclusters to ease the complexity of mixing these workloads with general workloads.
- If hard multitenancy is a business requirement, workloads should be deployed to a dedicated cluster.
- If multiple regions are needed for your applications, utilize a Global Load Balancer to manage traffic between clusters.
- You can break out specialized workloads such as HPC into their own individual clusters to ensure that the specialized needs for the workloads are met.
- If you're deploying workloads that will be spread across multiple regional datacenters, first ensure that there is a data replication strategy for the workload. Multiple clusters across regions can be easy, but replicating data across regions

can be complicated, so ensure that there is a sound strategy to handle asynchronous and synchronous workloads.

- Utilize Kubernetes operators like the `prometheus-operator` or Elasticsearch operator to handle automated operational tasks.
- When designing your multicluster strategy, also consider how you will do service discovery and networking between clusters. Service mesh tools like HashiCorp's Consul or Istio can help with networking across clusters.
- Be sure that your CD strategy can handle multiple rollouts between regions or multiple clusters.
- Investigate utilizing a GitOps approach to managing multiple cluster operational components to ensure consistency between all clusters in your fleet. The GitOps approach doesn't always work for everyone's environment, but you should at least investigate it to ease the operational burden of multicluster environments.

# Summary

In this chapter, we discussed different strategies for managing multiple Kubernetes clusters. It's important to think about what your needs are at the outset and whether those needs match a multicluster topology. The first scenario to think about is whether you truly need *hard* multitenancy because this will automatically require a multicluster strategy. If you don't, consider your compliance needs and whether you have the operational capacity to consume the overhead of multicluster architectures. Finally, if you're going with more, smaller clusters, ensure that you put automation around the delivery and management of them to reduce the operational burden.

CHAPTER 13

# Integrating External Services and Kubernetes

In many of the chapters in this book, we've discussed how to build, deploy, and manage services in Kubernetes. However, the truth is that systems don't exist in a vaccum, and most of the services that we build will need to interact with systems and services that exist outside of the Kubernetes cluster in which they're running. This might be because we are building new services that are being accessed by legacy infrastructure running in virtual or physical machines. Conversely, it might be because the services that we are building might need to access preexisting databases or other services that are likewise running on physical infrastructure in an on-premises datacenter. Finally, you might have multiple different Kubernetes clusters with services that you need to interconnect. For all of these reasons, the ability to expose, share, and build services that span the boundary of your Kubernetes cluster is an important part of building real-world applications.

## Importing Services into Kubernetes

The most common pattern for connecting Kubernetes with external services consists of a Kubernetes service that is consuming a service that exists outside of the Kubernetes cluster. Often, this is because Kubernetes is being used for some new application development or interface for a legacy resource like an on-premises database. This pattern often makes the most sense for incremental development of cloud-native services. Because the database layer contains significant mission-critical data, it is a heavy lift to move it to the cloud, let alone containers. At the same time, there is a great deal of value in providing a modern layer on top of such a database (e.g., supplying a GraphQL interface) as the foundation for building a new generation of applications. Likewise, moving this layer to Kubernetes often makes a great deal of sense because

rapid development and reliable continuous deployment of this middleware enables a great deal of agility with minimal risk. Of course, to achieve this, you need to make the database accessible from within Kubernetes.

When we consider the task of making an external service accessible from Kubernetes, the first challenge is simply to get the networking to work correctly. The specific details of getting networking operational are very specific to both the location of the database as well as the location of the Kubernetes cluster; thus, they are beyond the scope of this book, but generally, cloud-based Kubernetes providers enable the deployment of a cluster into a user-provided virtual network (VNET), and those virtual networks can then be peered up with an on-premises network for connectivity.

After you've established network connectivity between pods in the Kubernetes cluster and the on-premises resource, the next challenge is to make the external service look and feel like a Kubernetes service. In Kubernetes, service discovery occurs via Domain Name System (DNS) lookups and, thus, to make our external database feel like it is a native part of Kubernetes, we need to make the database discoverable in the same DNS.

## Selector-Less Services for Stable IP Addresses

The first way to achieve this is with a *selector-less* Kubernetes Service. When you create a Kubernetes Service without a selector, there are no Pods that match the service; thus, there is no load balancing performed. Instead, you can program this selector-less service to have the specific IP address of the external resource that you want to add to the Kubernetes cluster. That way, when a Kubernetes pod performs a lookup for `your-database`, the built-in Kubernetes DNS server will translate that to a service IP address of your external service. Here is an example of a selector-less service for an external database:

```
apiVersion: v1
kind: Service
metadata:
  name: my-external-database
spec:
  ports:
  - protocol: TCP
    port: 3306
    targetPort: 3306
```

When the service exists, you need to update its endpoints to contain the database IP address serving at `24.1.2.3`:

```
apiVersion: v1
kind: Endpoints
metadata:
  # Important! This name has to match the Service.
  name: my-external-database
```

```
subsets:
  - addresses:
      - ip: 24.1.2.3
    ports:
      - port: 3306
```

Figure 13-1 depicts how this integrates together within Kubernetes.

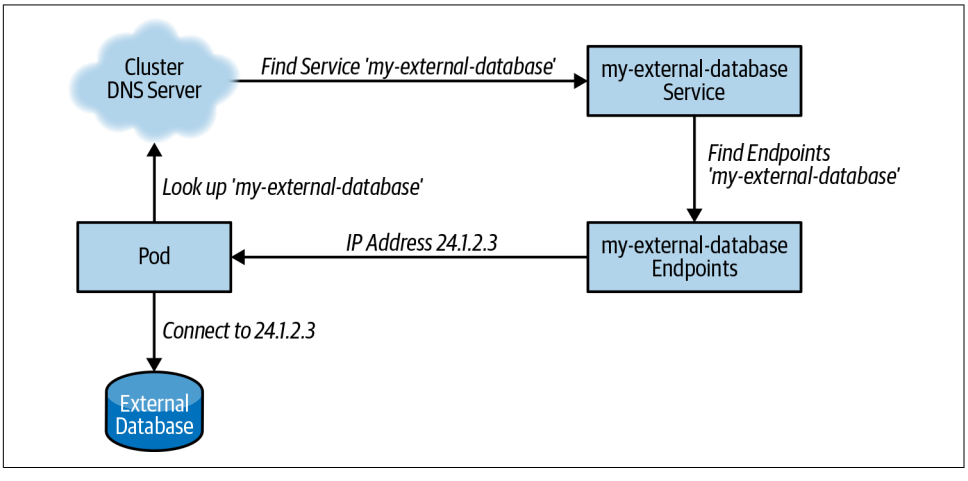



*Figure 13-1. Service integration*

## CNAME-Based Services for Stable DNS Names

The previous example assumed that the external resource that you were trying to integrate with your Kubernetes cluster had a stable IP address. Although this is often true of physical on-premises resources, depending on the network toplogy, it might not always be true, and it is significantly less likely to be true in a cloud environment where virtual machine (VM) IP addresses are more dynamic. Alternatively, the service might have multiple replicas sitting behind a single DNS-based load balancer. In these situations, the external service that you are trying to bridge into your cluster doesn't have a stable IP address, but it does have a stable DNS name.

In such a situation, you can define a CNAME-based Kubernetes Service. If you're not familiar with DNS records, a CNAME, or *Canonical Name*, record is an indication that a particular DNS address should be translated to a different *Canonical* DNS name. For example, a CNAME record for *foo.com* that contains *bar.com* indicates that anyone looking up *foo.com* should perform a recursive lookup for *bar.com* to obtain the correct IP address. You can use Kubernetes Services to define CNAME records in the Kubernetes DNS server. For example, if you have an external database with a DNS name of *database.myco.com*, you might create a CNAME *Service* that is named `myco-database`. Such a Service looks like this:

```
kind: Service
apiVersion: v1
metadata:
  name: myco-database
spec:
  type: ExternalName
  externalName: database.myco.com
```

With a Service defined in this way, any pod that does a lookup for `myco-database` will be recursively resolved to *database.myco.com*. Of course, to make this work, the DNS name of your external resource *also* needs to be resolveable from the Kubernetes DNS servers. If the DNS name is globally accessible (e.g., from a well-known DNS service provider), this will simply automatically work. However, if the DNS of the external service is located in a company-local DNS server (e.g., a DNS server that services only internal traffic), the Kubernetes cluster might not know by default how to resolve queries to this corporate DNS server.

To set up the cluster's DNS server to communicate with an alternate DNS resolver, you need to adjust its configuration. You do this by updating a Kubernetes ConfigMap with a configuration file for the DNS server. As of this writing, most clusters have moved over to the CoreDNS server. This server is configured by writing a `Corefile` configuration into a ConfigMap named `coredns` in the `kube-system` namespace. If you are still using the `kube-dns` server, it is configured in a similar manner but with a different ConfigMap.

CNAME records are a useful way to map external services with stable DNS names to names that are discoverable within your cluster. At first it might seem counterintuitive to remap a well-known DNS address to a cluster-local DNS address, but the consistency of having all services look and feel the same is usually worth the small amount of added complexity. Additionally, because the CNAME service, like all Kubernetes services, is defined per namespace, you can use namespaces to map the same service name (e.g., `database`) to different external services (e.g., `canary` or `production`), depending on the Kubernetes namespace.

## Active Controller-Based Approaches

In a limited set of circumstances, neither of the previous methods for exposing external services within Kubernetes is feasible. Generally, this is because there is neither a stable DNS address nor a single stable IP address for the service that you want to expose within the Kubernetes cluster. In such circumstances, exposing the external service within the Kubernetes cluster is significantly more complicated, but it isn't impossible.

To achieve this, you need to have some understanding of how Kubernetes Services work under the hood. Kubernetes Services are actually made up of two different resources: the Service resource, with which you are doubtless familiar, and the

Endpoints resource that represents the IP addresses that make up the service. In normal operation, the Kubernetes controller manager populates the endpoints of a service based on the selector in the service. However, if you create a selector-less service, as in the first stable-IP approach, the Endpoints resource for the service will not be populated, because there are no pods that are selected. In this situation, you need to supply the control loop to create and populate the correct Endpoints resource. You need to dynamically query your infrastructure to obtain the IP addresses for the service external to Kubernetes that you want to integrate, and then populate your service's endpoints with these IP addresses. After you do this, the mechanisms of Kubernetes take over and program both the DNS server and the `kube-proxy` correctly to load-balance traffic to your external service. Figure 13-2 presents a complete picture of how this works in practice.

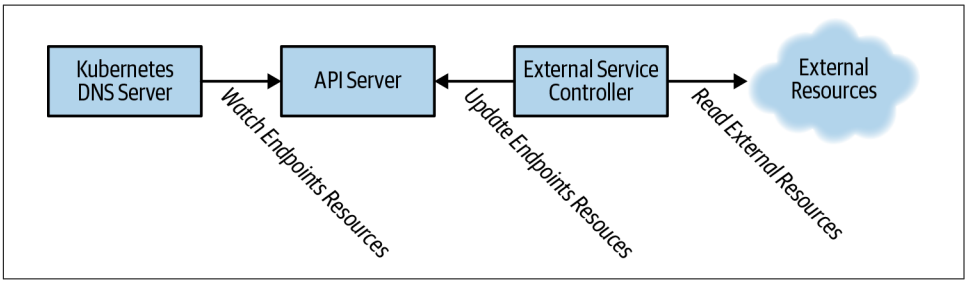



*Figure 13-2. An external service*

# Exporting Services from Kubernetes

In the previous section, we explored how to import preexisting services to Kubernetes, but you might also need to export services from Kubernetes to the preexisting environments. This might occur because you have a legacy internal application for customer management that needs access to some new API that you are developing in a cloud-native infrastructure. Alternately, you might be building new microservice-based APIs but you need to interface with a preexisting traditional web application firewall (WAF) because of internal policy or regulatory requirements. Regardless of the reason, being able to expose services from a Kubernetes cluster out to other internal applications is a critical design requirement for many applications.

The core reason that this can be challenging is because in many Kubernetes installations, the pod IP addresses are not routeable addresses from outside of the cluster. Via tools like flannel, or other networking providers, routing is established within a Kubernetes cluster to facilitate communication between pods and also between nodes and pods, but the same routing is not generally extended out to arbitrary machines in the same network. Furthermore, in the case of cloud to on-premises connectivity, the IP addresses of the pods are not always advertised back across a VPN or network peering relationship into the on-premises network. Consequently, setting up routing

between a traditional application and Kubernetes pods is the key task to enable the export of Kubernetes-based services.

## Exporting Services by Using Internal Load Balancers

The easiest way to export from Kubernetes is by using the built-in Service object. If you have had any previous experience with Kubernetes, you have no doubt seen how you can connect a cloud-based load balancer to bring external traffic to a collection of pods in the cluster. However, you might not have realized that most clouds also offer an *internal* load balancer. The internal load balancer provides the same capabilities to map a virtual IP address to a collection of pods, but that virtual IP address is drawn from an internal IP address space (e.g., `10.0.0.0/24`) and thus is only routeable from within that virtual network. You activate an internal load balancer by adding a cloud-specific annotation to your Service load balancer. For example, in Microsoft Azure, you add the `service.beta.kubernetes.io/azure-load-balancer-internal: "true"` annotation. On Amazon Web Services (AWS), the annotation is `service.beta.kubernetes.io/aws-load-balancer-internal: 0.0.0.0/0`. You place annotations in the `metadata` field in the Service resource as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  annotations:
    # Replace this as needed in other environments
    service.beta.kubernetes.io/azure-load-balancer-internal: "true"
...
```

When you export a Service via an internal load balancer, you receive a stable, routeable IP address that is visible on the virtual network outside of the cluster. You then can either use that IP address directly or set up internal DNS resolution to provide discovery for your exported service.

## Exporting Services on NodePorts

Unfortunately, in on-premises installations, cloud-based internal load balancers are unavailable. In this context using a NodePort-based service is often a good solution. A Service of type NodePort exports a listener on every node in the cluster that forwards traffic from the node's IP address and selected port into the Service that you defined, as shown in Figure 13-3.

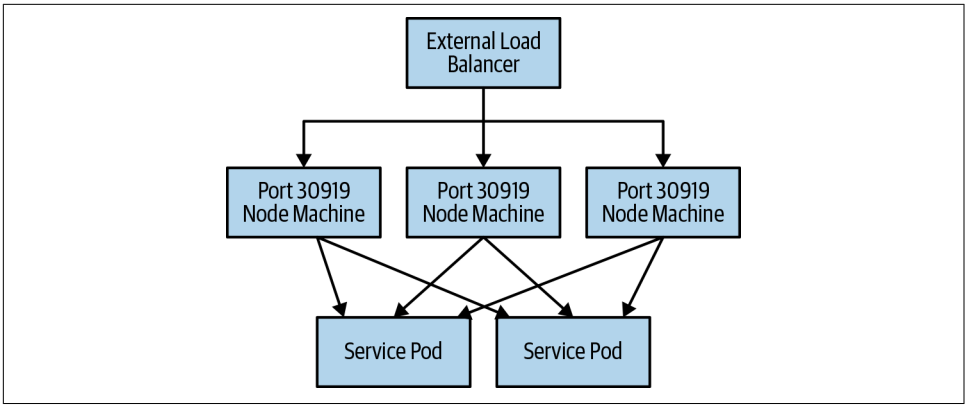



*Figure 13-3. A NodePort-based service*

Here's an example YAML file for a NodePort service:

```
apiVersion: v1
kind: Service
metadata:
  name: my-node-port-service
spec:
  type: NodePort
...
```

Following the creation of a Service of type NodePort, Kubernetes automatically selects a port for the service; you can get that port from the Service by looking at the `spec.ports[*].nodePort` field. If you want to choose the port yourself, you can specify it when you create the service, but the NodePort must be within the configured range for the cluster. The default for this range are ports between `30000` and `30999`.

Kubernetes' work is done when the service is exposed on this port. To export it to an existing application outside of the cluster, you (or your network administrator) will need to make it discoverable. Depending on the way your application is configured, you might be able to give your application a list of `${node}:${port}` pairs, and the application will perform client-side load balancing. Alternatively, you might need to configure a physical or virtual load balancer within your network to direct traffic from a virtual IP address to this list of `${node}:${port}` backends. The specific details for this configuration will be different depending on your environment.

## Integrating External Machines and Kubernetes

If neither of the previous solutions work well for you—perhaps because you want tighter integration for dynamic service discovery—the final choice for exposing Kubernetes services to outside applications is to directly integrate the machine(s) running the application into the Kubernetes cluster's service discovery and

networking mechanisms. This is significantly more invasive and complicated than either of the previous approaches, and you should use it only when necessary for your application (which should be infrequent). In some managed Kubernetes environments, it might not even be possible.

When integrating an external machine into the cluster for networking, you need to ensure that the pod network routing and DNS-based service discovery both work correctly. The easiest way to do this is actually to run the kubelet on the machine that you want to join to the cluster, but disable scheduling in the cluster. Joining a kubelet node to a cluster is beyond of the scope of this book, but there are numerous other books or online resources that describe how to achieve this. When the node is joined, you need to immediately mark it as unschedulable using the `kubectl cordon ...` command to prevent any additional work being scheduled on it. This cordoning will not prevent DaemonSets from landing pods onto the node, and thus the pods for both the KubeProxy and network routing will land on the machine and make Kubernetes-based services discoverable from any application running on that machine.

The previous approach is quite invasive to the node because it requires installing Docker or some other container runtime. Thus, it might not be feasible in many environments. A lighter weight but more complex approach is to just run the `kube-proxy` as a process on the machine and adjust the machine's DNS server. Assuming that you can set up pod routing to work correctly, running the `kube-proxy` will set up machine-level networking so that Kubernetes Service virtual IP addresses will be remapped to the pods that make up that Service. If you also change the machine's DNS to point to the Kubernetes cluster DNS server, you will have effectively enabled Kubernetes discovery on a machine that is not part of the Kubernetes cluster.

Both of these approaches are complicated and advanced, and you should not take them lightly. If you find yourself considering this level of service discovery integration, ask yourself whether it is possibly easier to actually bring the service you are connecting to the cluster into the cluster itself.

## Sharing Services Between Kubernetes

The previous sections have described how to connect Kubernetes applications to outside services and how to connect outside services to Kubernetes applications, but another significant use case is connecting services *between* Kubernetes clusters. This may be to achieve East-West failover between different regional Kubernetes clusters, or it might be to link together services run by different teams. The process of achieving this interaction is actually a combination of the designs described in the previous sections.

First, you need to expose the Service within the first Kubernetes cluster to enable network traffic to flow. Let's assume that you're in a cloud environment that supports internal load balancers, and that you receive a virtual IP address for that internal load balancer of *10.1.10.1*. Next, you need to integrate this virtual IP address into the second Kubernetes cluster to enable service discovery. You achieve this in the same manner as importing an external application into Kubernetes (first section). You create a selector-less Service and you set its IP address to be *10.1.10.1*. With these two steps you have integrated service discovery and connectivity between services within your two Kubernetes clusters.

These steps are fairly manual, and although this might be acceptable for a small, static set of services, if you want to enable tighter or automatic service integration between clusters, it makes sense to write a cluster daemon that runs in both clusters to perform the integration. This daemon would watch the first cluster for Services with a particular annotation, say something like `myco.com/exported-service`; all Services with this annotation would then be imported into the second cluster via selector-less services. Likewise, the same daemon would garbage-collect and delete any services that are exported into the second cluster but are no longer present in the first. If you set up such daemons in each of your regional clusters, you can enable dynamic, East-West connectivity between all clusters in your environment.

## Third-Party Tools

Thus far, this chapter has described the various ways to import, export, and connect services that span Kubernetes clusters and some outside resource. If you have previous experience with service mesh technologies, these concepts might seem quite familiar to you. Indeed, there are a variety of third-party tools and projects that you can use to interconnect services both with Kubernetes and with arbitrary applications and machines. Generally, these tools can provide a lot of functionality, but they are also significantly more complex operationally than the approaches described just earlier. However, if you find yourself building more and more networking interconnectivity, you should explore the space of service meshes, which is rapidly iterating and evolving. Nearly all of these third-party tools have an open source component, but they also offer commercial support that can reduce the operational overhead of running additional infrastructure.

## Connecting Cluster and External Services Best Practices

- Establish network connectivity between the cluster and on-premises. Networking can be varied between different sites, clouds, and cluster configurations, but first ensure that pods can talk to on-premises machines and vice versa.

- To access services outside of the cluster, you can use selector-less services and directly program in the IP address of the machine (e.g., the database) with which you want to communicate. If you don't have fixed IP addressess, you can instead use CNAME services to redirect to a DNS name. If you have neither a DNS name nor fixed services, you might need to write a dynamic operator that periodically synchronizes the external service IP addresses with the Kubernetes Service endpoints.
- To export services from Kubernetes, use internal load balancers or NodePort services. Internal load balancers are typically easier to use in public cloud environments where they can be bound to the Kubernetes Service itself. When such load balancers are unavailable, NodePort services can expose the service on all of the machines in the cluster.
- You can achieve connections between Kubernetes clusters through a combination of these two approaches, exposing a service externally that is then consumed as a selector-less service in the other Kubernetes cluster.

## Summary

In the real world, not every application is cloud native. Building applications in the real world often involves connecting preexisting systems with newer applications. This chapter described how you can integrate Kubernetes with legacy applications and also how to integrate different services running across multiple distinct Kubernetes clusters. Unless you have the luxury of building something brand new, cloud-native development will always require legacy integration. The techniques described in this chapter will help you achieve that.

CHAPTER 14

# Running Machine Learning in Kubernetes

The age of microservices, distributed systems, and the cloud has provided the perfect environmental conditions for the democratization of machine learning models and tooling. Infrastructure at scale has now become commoditized, and the tooling around the machine learning ecosystem is maturing. It just so happens that Kubernetes is one of the platforms that has become increasingly popular among data scientists and the wider open source community as the perfect environment to enable the machine learning workflow and life cycle. In this chapter, we will cover why Kubernetes is a great place for machine learning and provide best practices for both cluster administrators and data scientists alike on how to get the most out of Kubernetes when running machine learning workloads. Specifically, we focus on deep learning rather than traditional machine learning because deep learning has fast become the area of innovation on platforms like Kubernetes.

## Why Is Kubernetes Great for Machine Learning?

Kubernetes has quickly become the home for rapid innovation in deep learning. The confluence of tooling and libraries such as TensorFlow make this technology more accessible to a large audience of data scientists. What makes Kubernetes such a great place to run your deep learning workloads? Let's cover what Kubernetes provides:

*Ubiquitous*
: Kubernetes is everywhere. All of the major public clouds support it, and there are distributions for private clouds and infrastructure. Basing ecosystem tooling on a platform like Kubernetes allows users to run their deep learning workloads anywhere.

*Scalable*
: Deep learning workflows typically need access to large amounts of computing power in order to efficiently train machine learning models. Kubernetes ships with native autoscaling capabilities that make it easy for data scientists to achieve and fine-tune the level of scale they need to train their models.

*Extensible*
: Efficiently training a machine learning model typically requires access to specialized hardware. Kubernetes allows cluster administrators to quickly and easily expose new types of hardware to the scheduler without having to change the Kubernetes source code. It also allows custom resources and controllers to be seamlessly integrated into the Kubernetes API to support specialized workflows, such as hyperparameter tuning.

*Self-service*
: Data scientists can use Kubernetes to perform self-service machine learning workflows on demand, without needing specialized knowledge of Kubernetes itself.

*Portable*
: Machine learning models can be run anywhere, provided that the tooling is based on the Kubernetes API. This allows machine learning workloads to be portable across Kubernetes providers.

# Machine Learning Workflow

To effectively understand the needs of deep learning, you must understand the complete workflow. Figure 14-1 represents a simplified machine learning workflow.

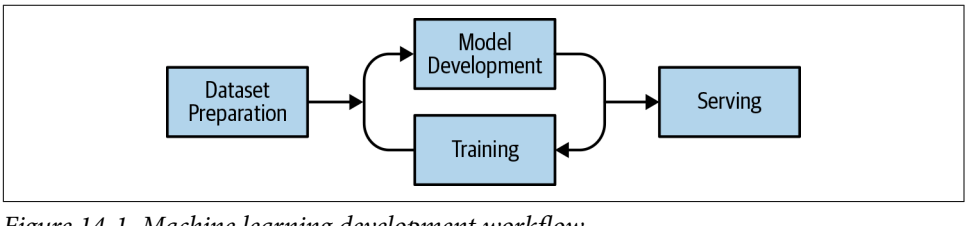



*Figure 14-1. Machine learning development workflow*

Figure 14-1 illustrates that the machine learning development workflow has the following phases:

*Dataset preparation*
: This phase includes the storage, indexing, cataloging, and metadata associated with the dataset that is used to train the model. For the purposes of this book, we consider only the storage aspect. Datasets vary in size, from hundreds of

megabytes to hundreds of terabytes. The dataset needs to be provided to the model in order for the model to be trained. You must consider storage that provides the appropriate properties to meet these needs. Typically, large-scale block and object stores are required and must be accessible via Kubernetes native storage abstractions or directly accessible APIs.

*Machine learning algorithm development*
: This is the phase in which data scientists write, share, and collaborate on machine learning algorithms. Open source tools like JupyterHub are easy to install on Kubernetes because they typically function like any other workload.

*Training*
: This is the process by which the model will use the dataset to learn how to perform the tasks for which it has been designed. The resulting artifact of training process is usually a checkpoint of the trained model state. The training process is the piece that takes advantage of all of the capabilities of Kubernetes at the same time. Scheduling, access to specialized hardware, dataset volume management, scaling, and networking will all be exercised in unison in order to complete this task. We cover more of the specifics of the training phase in the next section.

*Serving*
: This is the process of making the trained model accessible to service requests from clients so that it can make predictions based on the the data supplied from the client. For example, if you have an image-recognition model that's been trained to detect dogs and cats, a client might submit a picture of a dog, and the model should be able to determine whether it is a dog, with a certain level of accuracy.

# Machine Learning for Kubernetes Cluster Admins

In this section, we discuss topics you will need to consider before running machine learning workloads on your Kubernetes cluster. This section is specifically targeted toward cluster administrators. The largest challenge you will face as a cluster administrator responsible for a team of data scientists is understanding the terminology. There are myriad new terms that you must become familiar with over time, but rest assured, you can do it. Let's take a look at the main problem areas you'll need to address when preparing a cluster for machine learning workloads.

## Model Training on Kubernetes

Training machine learning models on Kubernetes requires conventional CPUs and graphics processing units (GPUs). Typically, the more resources you apply, the faster the training will be completed. In most cases, model training can be achieved on a single machine that has the required resources. Many cloud providers offer

multi-GPU virtual machine (VM) types, so we recommend scaling VMs vertically to four to eight GPUs before looking into distributed training. Data scientists use a technique known as *hyperparameter tuning* when training models. Hyperparameter tuning is the process of finding the optimal set of hyperparameters for model training. A hyperparameter is simply a parameter that has a set value before the training process begins. The technique involves running many of the same training jobs with a different set of hyperparameters.

### Training your first model on Kubernetes

In this example, you are going to use the MNIST dataset to train an image-classification model. The MNIST dataset is publicly available and commonly used for image classification.

To train the model, you are going to need GPUs. Let's confirm that your Kubernetes cluster has GPUs available. The following output shows that this Kubernetes cluster has four GPUs available:

```
$ kubectl get nodes -o yaml | grep -i nvidia.com/gpu
      nvidia.com/gpu: "1"
      nvidia.com/gpu: "1"
      nvidia.com/gpu: "1"
      nvidia.com/gpu: "1"
```

To run your training, you are going to using the `Job` kind in Kubernetes, given that training is a batch workload. You are going to run your training for 500 steps and use a single GPU. Create a file called *mnist-demo.yaml* using the following manifest, and save it to your filesystem:

```
apiVersion: batch/v1
kind: Job
metadata:
  labels:
    app: mnist-demo
  name: mnist-demo
spec:
  template:
    metadata:
      labels:
        app: mnist-demo
    spec:
      containers:
      - name: mnist-demo
        image: lachlanevenson/tf-mnist:gpu
        args: ["--max_steps", "500"]
        imagePullPolicy: IfNotPresent
        resources:
          limits:
           nvidia.com/gpu: 1
      restartPolicy: OnFailure
```

Now, create this resource on your Kubernetes cluster:

```
$ kubectl create -f mnist-demo.yaml
job.batch/mnist-demo created
```

Check the status of the job you just created:

```
$ kubectl get jobs
NAME         COMPLETIONS   DURATION   AGE
mnist-demo   0/1           4s         4s
```

If you take a look at the pods, you should see the training job running:

```
$ kubectl get pods
NAME               READY   STATUS    RESTARTS   AGE
mnist-demo-hv9b2   1/1     Running   0          3s
```

Looking at the pod logs, you can see the training happening:

```
$ kubectl logs mnist-demo-hv9b2
2019-08-06 07:52:21.349999: I tensorflow/core/platform/cpu_feature_guard.cc:
137] Your CPU supports instructions that this TensorFlow binary was not com-
piled to use: SSE4.1 SSE4.2 AVX AVX2 FMA
2019-08-06 07:52:21.475416: I tensorflow/core/common_runtime/gpu/gpu_device.cc:
1030] Found device 0 with properties:
name: Tesla K80 major: 3 minor: 7 memoryClockRate(GHz): 0.8235
pciBusID: d0c5:00:00.0
totalMemory: 11.92GiB freeMemory: 11.85GiB
2019-08-06 07:52:21.475459: I tensorflow/core/common_runtime/gpu/gpu_device.cc:
1120] Creating TensorFlow device (/device:GPU:0) -> (device: 0, name: Tesla
K80, pci bus id: d0c5:00:00.0, compute capability: 3.7)
2019-08-06 07:52:26.134573: I tensorflow/stream_executor/dso_loader.cc:139] suc-
cessfully opened CUDA library libcupti.so.8.0 locally
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting /tmp/tensorflow/input_data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /tmp/tensorflow/input_data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /tmp/tensorflow/input_data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /tmp/tensorflow/input_data/t10k-labels-idx1-ubyte.gz
Accuracy at step 0: 0.1255
Accuracy at step 10: 0.6986
Accuracy at step 20: 0.8205
Accuracy at step 30: 0.8619
Accuracy at step 40: 0.8812
Accuracy at step 50: 0.892
Accuracy at step 60: 0.8913
Accuracy at step 70: 0.8988
Accuracy at step 80: 0.9002
Accuracy at step 90: 0.9097
Adding run metadata for 99
...
```

Finally, you can see that the training has completed by looking at the job status:

```
$ kubectl get jobs
NAME         COMPLETIONS   DURATION   AGE
mnist-demo   1/1           27s        112s
```

To clean up the training job, simply run the following command:

```
$ kubectl delete -f mnist-demo.yaml
job.batch "mnist-demo" deleted
```

Congratulations! You just ran your first model training job on Kubernetes.

## Distributed Training on Kubernetes

Distributed training is still in its infancy and is difficult to optimize. Running a training job that requires eight GPUs will almost always be faster to train on a single eight-GPU machine compared to two machines each with four GPUs. The only time that you should resort to using distributed training is when the model doesn't fit on the biggest machine available. If you are certain that you must run distributed training, it is important to understand the architecture. Figure 14-2 depicts the distributed TensorFlow architecture, and you can see how the model and the parameters are distributed.

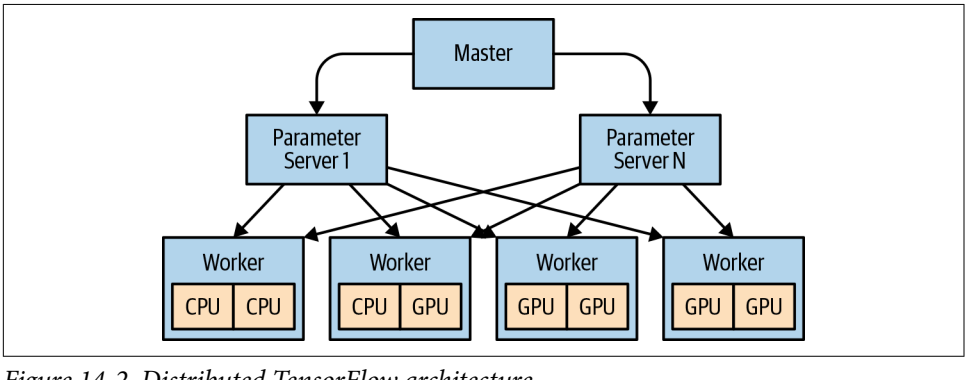



*Figure 14-2. Distributed TensorFlow architecture*

## Resource Constraints

Machine learning workloads demand very specific configurations across all aspects of your cluster. The training phases are most certainly the most resource intensive. It's also important to note, as we mentioned a moment ago, that machine learning algorithm training is almost always a batch-style workload. Specifically, it will have a start time and a finish time. The finish time of a training run depends on how quickly you can meet the resource requirements of the model training. This means that scaling is almost certainly a quicker way to finish training jobs faster, but scaling has its own set of bottlenecks.

## Specialized Hardware

Training and serving a model is almost always more efficient on specialized hardware. A typical example of such specialized hardware would be commodity GPUs. Kubernetes allows you to access GPUs via device plug-ins that make the GPU resource known to the Kubernetes scheduler and therefore able to be scheduled. There is a device plug-in framework that facilitates this capability, which means that vendors do not need to modify the core Kubernetes code to implement their specific device. These device plug-ins typically run as DaemonSets on each node, which are processes that are responsible for advertising these specific resources to the Kubernetes API. Let's take a look at the NVIDIA device plug-in for Kubernetes, which enables access to NVIDIA GPUs. After they're running, you can create a pod as follows, and Kubernetes will ensure that it is scheduled to a node that has these resource available:

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-pod
spec:
  containers:
    - name: digits-container
      image: nvidia/digits:6.0
      resources:
        limits:
          nvidia.com/gpu: 2 # requesting 2 GPUs
```

Device plug-ins are not limited to GPUs; you can use them wherever specialized hardware is needed—for example, Field Programmable Gate Arrays (FPGAs) or InfiniBand.

### Scheduling idiosyncrasies

It's important to note that Kubernetes cannot make decisions about resources that it does not have knowledge about. One of the things you might notice is that the GPUs are not running at capacity when you are training. You are therefore not achieving the level of utilization that you would like to see. Let's consider the previous example; it exposes only the number of GPU cores and omits the number of threads that can be run per core. It also doesn't expose which bus the GPU core is on, so that jobs that need access to one another or to the same memory might be colocated on the same Kubernetes nodes. These are all considerations that might be addressed by device plug-ins in the future but might leave you wondering why you cannot get 100% utilization on that beefy GPU you just purchased. It's also worth mentioning that you cannot request fractions of GPUs (for example, 0.1), which means that even if the specific GPU supports running multiple threads concurrently, you will not be able to utilize that capacity.

## Libraries, Drivers, and Kernel Modules

To access specialized hardware, you typically need purpose-built libraries, drivers, and kernel modules. You will need to ensure that these are mounted into the container runtime so that they are available to the tooling running in the container. You might ask, "Why don't I just add these to the container image itself?" The answer is simple: the tools need to match the version on the underlying host and must be configured appropriately for that specific system. There are container runtimes such as NVIDIA Docker that remove the burden of having to map host volumes into each container. In lieu of having a purpose-built container runtime, you might also be able to build an admission webhook that provides the same functionality. It's also important to consider that you might need privileged containers to access some specialized hardware, which also affects the cluster security profile. The installation of the associated libraries, drivers, and kernel modules might also be facilitated by Kubernetes device plug-ins. Many device plug-ins run checks on each machine to confirm that all installations have been completed before they advertise the schedulable GPU resources to the Kubernetes scheduler.

## Storage

Storage is one of the most critical aspects of the machine learning workflow. You need to consider storage because it directly affects the following pieces of the machine learning workflow:

- Dataset storage and distribution among worker nodes during training
- Checkpoints and saving models

### Dataset storage and distribution among worker nodes during training

During training, the dataset must be retrievable by every worker node. The storage needs are read-only, and, typically, the faster the disk, the better. The type of disk that's providing the storage is almost completely dependent on the size of the dataset. Datasets of hundreds of megabytes or gigabytes might be perfect for block storage, but datasets that are several or hundreds of terabytes in size might be better suited to object storage. Depending on the size and location of the disks that hold the datasets, there might be a performance hit on your networking.

### Checkpoints and saving models

Checkpoints are created as a model is being trained, and saving models allows you to use them for serving. In both cases, you need storage attached to each of the worker nodes to store this data. The data is typically stored under a single directory, and each worker node is writing to a specific checkpoint or save file. Most tools expect the

checkpoint and save data to be in a single location and require `ReadWriteMany`. `ReadWriteMany` simply means that the volume can be mounted as read-write by many nodes. When using Kubernetes PersistentVolumes, you will need to determine the best storage platform for your needs. The Kubernetes documentation keeps a list of volume plug-ins that support `ReadWriteMany`.

## Networking

The training phase of the machine learning workflow has a large impact on the network (specifically, when running distributed training). If we consider TensorFlow's distributed architecture, there are two discrete phases to consider that create a lot of network traffic: variable distribution from each of the parameter servers to each of the worker nodes, and also the application of gradients from each worker node back to the parameter server (see Figure 14-2). The time it takes for this exchange to happen directly affects the time it takes to train a model. So, it's a simple game of the faster, the better (within reason, of course). With most public clouds and servers today supporting 1-Gbps, 10-Gbps, and sometimes 40-Gbps network interface cards, generally network bandwidth is only a concern at lower bandwidths. You might also consider InfiniBand if you need high network bandwidth.

While raw network bandwidth is more often than not a limiting factor, there are also instances for which getting the data onto the wire from the kernel in the first place is the problem. There are open source projects that take advantage of Remote Direct Memory Access (RDMA) to further accelerate network traffic without the need to modify your worker nodes or application code. RDMA allows computers in a network to exchange data in main memory without using the processor, cache, or operating system of either computer. You might consider the open source project Freeflow, which boasts of having high network performance for container network overlays.

## Specialized Protocols

There are other specialized protocols that you can consider when using machine learning on Kubernetes. These protocols are often vendor specific, but they all seek to address distributed training scaling issues by removing areas of the architecture that quickly become bottlenecks, for example, parameter servers. These protocols often allow the direct exchange of information between GPUs on multiple nodes without the need to involve the node CPU and OS. Here are a couple that you might want to look into to more efficiently scale your distributed training:

- Message Passing Interface (MPI) is a standardized portable API for the transfer of data between distributed processes.

- NVIDIA Collective Communications Library (NCCL) is a library of topology-aware multi-GPU communication primitives.

## Data Scientist Concerns

In the previous discussion, we shared considerations that you need to make in order to be able to run machine learning workloads on your Kubernetes cluster. But what about the data scientist? Here we cover some popular tools that make it easy for data scientists to utilize Kubernetes for machine learning without having to be a Kubernetes expert.

- Kubeflow is a machine learning toolkit for Kubernetes. It is native to Kubernetes and ships with several tools necessary to complete the machine learning workflow. Tools such as Jupyter Notebooks, pipelines, and Kubernetes-native controllers make it simple and easy for data scientists to get the most out of Kubernetes as a platform for machine learning.
- Polyaxon is a tool for managing machine learning workflows that supports many popular libraries and runs on any Kubernetes cluster. Polyaxon has both commercial and open source offerings.
- Pachyderm is an enterprise-ready data science platform that has a rich suite of tools for dataset preparation, life cycle, and versioning along with the ability to build machine learning pipelines. Pachyderm has a commercial offering that you can deploy to any Kubernetes cluster.

## Machine Leaning on Kubernetes Best Practices

To achieve optimal performance for your machine learning workloads, consider the following best practices:

- Smart scheduling and autoscaling. Given that most stages of the machine learning workflow are batch by nature, we recommend that you utilize a Cluster Autoscaler. GPU-enabled hardware is costly, and you certainly do not want to be paying for it when it's not in use. We recommend batching jobs to run at specific times using either taints and tolerations or via a time-specific Cluster Autoscaler. That way, the cluster can scale to the needs of the machine learning workloads when needed, and not a moment sooner. Regarding taints and tolerations, upstream convention is to taint the node with the extended resource as the key. For example, a node with NVIDIA GPUs should be tainted as follows: `Key: nvidia.com/gpu, Effect: NoSchedule`. Using this method means that you can also utilize the `ExtendedResourceToleration` admission controller, which will automatically add the appropriate tolerations for such taints to pods requesting extended resources so that the users don't need to manually add them.

- The truth is that model training is a delicate balance. Allowing things to move faster in one area often leads to bottlenecks in others. It's an endeavor of constant observation and tuning. As a general rule of thumb, we recommend that you try to make the GPU become the bottleneck because it is the most costly resource. Keep your GPUs saturated. Be prepared to always be on the lookout for bottlenecks, and set up your monitoring to track the GPU, CPU, network, and storage utilization.
- Mixed workload clusters. Clusters that are used to run the day-to-day business services might also be used for the purposes of machine learning. Given the high performance requirements of machine learning workloads, we recommend using a separate node pool that's tainted to accept only machine learning workloads. This will help protect the rest of the cluster from any impact from the machine learning workloads running on the machine learning node pool. Furthermore, you should consider multiple GPU-enabled node pools, each with different performance characteristics to suit the workload types. We also recommend enabling node autoscaling on the machine learning node pool(s). Use mixed mode clusters only after you have a solid understanding of the performance impact that your machine learning workloads have on your cluster.
- Achieving linear scaling with distributed training. This is the holy grail of distributed model training. Most libraries unfortunately don't scale in a linear fashion when distributed. There is lots of work being done to make scaling better, but it's important to understand the costs because this isn't as simple as throwing more hardware at the problem. In our experience, it's almost always the model itself and not the infrastructure supporting it that is the source of the bottleneck. It is, however, important to review the utilization of the GPU, CPU, network, and storage before pointing fingers at the model itself. Open source tools such as Horovod seek to improve distributed training frameworks and provide better model scaling.

# Summary

We've covered a lot of ground in this chapter and have hopefully provided valuable insight into why Kubernetes is a great platform for machine learning, especially deep learning, and the considerations you need to be aware of before deploying your first machine learning workload. If you exercise the recommendations in this chapter, you will be well equipped to build and maintain a Kubernetes cluster for these specialized workloads.

CHAPTER 15

# Building Higher-Level Application Patterns on Top of Kubernetes

Kubernetes is a complex system. Although it simplifies the deployment and operations of distributed applications, it does little to make the development of such systems easy. Indeed, in adding new concepts and artifacts for the developer to interact with, it adds an additional layer of complexity in the service of simplified operations. Consequently, in many environments, it makes sense to develop higher-level abstractions in order to provide more developer-friendly primitives on top of Kubernetes. Additionally, in many large companies, it makes sense to standardize the way in which applications are configured and deployed so that everyone adheres to the same operational best practices. This can also be achieved by developing higher-level abstractions so that developers automatically adhere to these principles. However, developing these abstractions can hide important details from the developer and might introduce a walled garden that limits or complicates the development of certain applications or the integration of existing solutions. Throughout the development of the cloud, the tension between the flexibility of infrastructure and the power of the platform has been a constant. Designing the proper higher-level abstractions enables us to walk an ideal path through this divide.

## Approaches to Developing Higher-Level Abstractions

When considering how to develop a higher-level primitive on top of Kubernetes, there are two basic approaches. The first is to wrap up Kubernetes as an implementation detail. With this approach, developers who consume your platform should be largely unaware that they are running on top of Kubernetes; instead, they should think of themselves as consumers of the platform you supply, and thus Kubernetes is an implementation detail.

The second option is to use the extensibility capabilities built into Kubernetes itself. The Kubernetes Server API is quite flexible, and you can dynamically add arbitrary new resources to the Kubernetes API itself. With this approach, your new higher-level resources coexist alongside the built-in Kubernetes objects, and the users use the built-in tooling for interacting with all of the Kubernetes resources, both built-in ones and extensions. This extension model results in an environment in which Kubernetes is still front and center for your developers but with additions that reduce complexity and make it easier to use.

Given the two approaches, how do you choose the one that is appropriate? It really depends on the goals for the abstraction layer that you are building. If you are constructing a fully isolated, integrated environment in which you have strong confidence that users will not need to "break glass" and escape, and where ease of use is an important characteristic, the first option is a great choice. A good example of such a use case would be building a machine learning pipeline. The domain is relatively well understood. The data scientists who are your users are likely not familiar with Kubernetes. Enabling these data scientists to rapidly get their work done and focus on their domains rather than distributed systems is the primary goal. Thus, building a complete abstraction on top of Kubernetes makes the most sense.

On the other hand, when building a higher-level developer abstraction—for example, an easy way to deploy Java applications—it is a far better choice to extend Kubernetes rather than wrap it. The reason for this is two-fold. First, the domain of application development is extraordinarily broad. It will be difficult for you to anticipate all of the requirements and use cases for your developers, especially as the applications and business iterate and change over time. The other reason is to ensure that you can continue to take advantage of the Kubernetes ecosystem of tools. There are countless cloud-native tools for monitoring, continuous delivery, and more. Extending rather than replacing the Kubernetes API ensures that you can continue to use these tools and new ones as they are developed.

# Extending Kubernetes

Because every layer that you might build over Kubernetes is unique, it is beyond the scope of this book to describe how you might build such a layer. But the tools and techniques for extending Kubernetes are generic to any construction you might do on top of Kubernetes, and, thus, we'll spend time covering them.

## Extending Kubernetes Clusters

A complete how-to for extending a Kubernetes cluster is a large topic and more completely covered in other books like *Managing Kubernetes* and *Kubernetes: Up and Running* (O'Reilly). Rather than going over the same material here, this section focuses on providing an understanding of how to use Kubernetes extensibility. Extending

the Kubernetes cluster involves understanding the touch points for resources in Kubernetes. There are three related technical solutions. The first is the *sidecar*. Sidecar containers (shown in Figure 15-1) have been popularized in the context of service meshes. They are containers that run alongside a main application container to provide additional capabilities that are decoupled from the main application and often maintained by a separate team. For example, in service meshes, a sidecar might provide transparent mutual Transport Layer Security (mTLS) authentication to a containerized application.

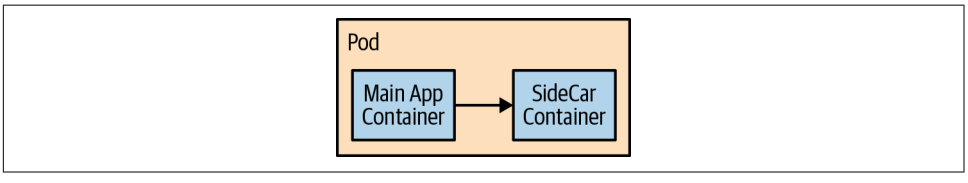



*Figure 15-1. The sidecar design*

You can use sidecars to add capabilities to your user-defined applications.

Of course, the entire goal of this effort was to make a developer's life easier, but if we require that they learn about and know how to use sidecars, we've actually made the problem worse. Fortunately, there are additional tools for extending Kubernetes that simplify things. In particular, Kubernetes features *admission controllers*. Admission controllers are interceptors that read Kubernetes API requests prior to them being stored (or "admitted") into the cluster's backing store. You can use these admission controllers to validate or modify API objects. In the context of sidecars, you can use them to automatically add sidecars to all pods created in the cluster so that developers do not need to know about the sidecars in order to reap their benefits. Figure 15-2 illustrates how admission controllers interact with the Kubernetes API.

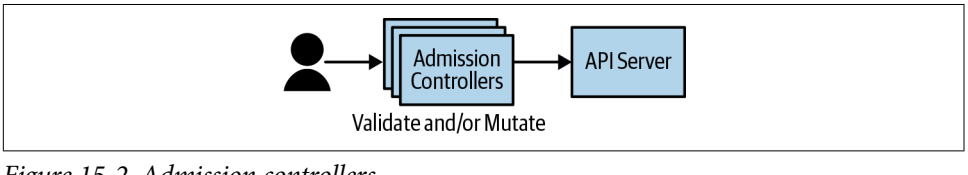



*Figure 15-2. Admission controllers*

The utility of admission controllers isn't limited to adding sidecars. You can also use them to validate objects submitted by developers to Kubernetes. For example, you could implement a *linter* for Kubernetes that ensures developers submit pods and other resources that follow best practices for using Kubernetes. A common mistake for developers is to not reserve resources for their application. For those circumstances, an admission controller-based linter could intercept such requests and reject them. Of course, you should also leave an escape hatch (for example, a special

annotation) so that advanced users can opt out of the lint rule, as appropriate. We discuss the importance of escape hatches later on in the chapter.

So far, we've only covered ways to augment existing applications and to ensure that developers follow best practices—we haven't really covered how to add higher-level abstractions. This is where custom resource definitions (CRDs) come into play. CRDs are a way to dynamically add new resources to an existing Kubernetes cluster. For example, using CRDs, you could add a new ReplicatedService resource to a Kubernetes cluster. When a developer creates an instance of a ReplicatedService, it turns around to Kubernetes and creates corresponding Deployment and Service resources. Thus, the ReplicatedService is a convenient developer abstraction for a common pattern. CRDs are generally implemented by a control loop that is deployed into the cluster itself to manage these new resource types.

## Extending the Kubernetes User Experience

Adding new resources to your cluster is a great way to provide new capabilities, but to truly take advantage of them, it's often useful to extend the Kubernetes user experience (UX) as well. By default, the Kubernetes tooling is unaware of custom resources and other extensions and thus treats them in a very generic and not particularly user-friendly manner. Extending the Kuberentes command line can provide an enhanced user experience.

Generally, the tool used for accessing Kubernetes is the `kubectl` command-line tool. Fortunately, it too has been built for extensibility. `kubectl` plug-ins are binaries that have a name like `kubectl-foo`, where `foo` is the name of the plug-in. When you invoke `kubectl foo ...` on the command line, the invocation is in turn routed to an invocation of the plug-in binary. Using `kubectl` plug-ins, you can define new experiences that deeply understand the new resources that you have added to your cluster. You are free to implement whatever kind of experiences are suitable while at the same time taking advantage of the familiarity of the `kubectl` tooling. This is especially valuable because it means that you don't need to teach developers about a new tool set. Likewise, you can gradually introduce Kubernetes-native concepts as the developers advance their Kubernetes knowledge.

# Design Considerations When Building Platforms

Countless platforms have been built to enable developer productivity. Given the opportunity to observe all of the places where these platforms have succeeded and failed, you can develop a common set of patterns and considerations so as to learn from the experience of others. Following these design guidelines can help to ensure that the platform you build is a successful one instead of a "legacy" dead end from which you must eventually move away.

## Support Exporting to a Container Image

When building a platform, many designs provide simplicity by enabling the user to simply supply code (e.g., a function in Function as a Service [FaaS]) or a native package (e.g., a JAR file in Java) instead of a complete container image. This approach has a great deal of appeal because it lets the user stay within the confines of their well-understood tools and development experience. The platform handles the containerization of the application for them.

The problem with this approach, however, comes when the developer encounters the limitations of the programming environment that you have given them. Perhaps it's because they need a specific version of a language runtime to work around a bug. Or it might be that they need to package additional resources or executables that aren't part of the way you have structured the automatic containerazation of the application.

No matter the reason, hitting this wall is an ugly moment for the developer, because it is a moment when they suddenly must learn a great deal more about how to package their application, when all they really wanted to do was to extend it slightly to fix a bug or deliver a new feature.

However, it doesn't need to be this way. If you support the exporting of your platform's programming environment into a generic container, the developer using your platform doesn't need to start from scratch and learn everything there is to know about containers. Instead, they have a complete, working container image that represents their current application (e.g., the container image containing their function and the node runtime). Given this starting point, they can then make the small tweaks necessary to adapt the container image to their needs. This sort of gradual degradation and incremental learning dramatically smoothes out the path from higher-level platform down into lower-level infrastructure and thus increases the general utility of the platform because using it doesn't introduce steep cliffs for developers.

## Support Existing Mechanisms for Service and Service Discovery

Another common story of platforms is that they evolve and interconnect with other systems. Many developers might be very happy and productive in your platform, but any real-world application will span both the platform that you build and lower-level Kubernetes applications as well as *other* platforms. Connections to legacy databases or open source applications built for Kubernetes will always become a part of a sufficiently large application.

Because of this need for interconnectivity, it's critically important that the core Kubernetes primitives for services and service discovery are used and exposed by any platform that you construct. Don't reinvent the wheel in the interest of improved

platform experience, because in doing so you will be creating a walled garden incapable of interacting with the broader world.

If you expose the applications defined in your platform as Kubernetes Services, any application anywhere within your cluster will be able to consume your applications regardless of whether they are running in your higher-level platform. Likewise, if you use the Kubernetes DNS servers for service discovery, you will be able to connect from your higher-level application platform to other applications running in the cluster, even if they are not defined in your higher-level platform. It might be tempting to build something better or easier to use, but interconnectivity across different platforms is the common design pattern for any application of sufficient age and complexity. You will always regret the decision to build a walled garden.

# Building Application Platforms Best Practices

Although Kubernetes provides powerful tools for operating software, it does considerably less to enable developers to build applications. Thus, it is often necessary to build platforms on top of Kubernetes to make developers more productive and/or Kubernetes easier. When building such platforms, you'll benefit from keeping the following best practices in mind:

- Use admission controllers to limit and modify API calls to the cluster. An admission controller can validate (and reject invalid) Kubernetes resources. A mutating admission controller can automatically modify API resources to add new sidecars or other changes that users might not even need to know about.
- Use `kubectl` plug-ins to extend the Kubernetes user experience by adding new tools to the familiar existing command-line tool. In rare occasions, a purpose-built tool might be more appropriate.
- When building platforms on top of Kubernetes, think carefully about the users of the platform and how their needs will evolve. Making things simple and easy to use is clearly a good goal, but if this also leads to users that are trapped and unable to be successful without rewriting everything outside of your platform, it will ultimately be a frustrating (and unsuccessful) experience.

# Summary

Kubernetes is a fantastic tool for simplifying the deployment and operation of software, but unfortunately, it is not always the most developer-friendly or productive environment. Because of this, a common task is to build a higher-level platform on top of Kubernetes in order to make it more approachable and usable by the average developer. This chapter described several approaches for designing such a higher-level system and provided a summary of the core extensibility infrastructure that is

available in Kubernetes. It concluded with lessons and design principles drawn from our observation of other platforms that have been built on top of Kubernetes, with the hope that they can guide the design of your platform.

CHAPTER 16

# Managing State and Stateful Applications

In the early days of container orchestration, the targeted workloads were usually stateless applications that used external systems to store state if necessary. The thought was that containers are very temporal, and orchestration of the backing storage needed to keep state in a consistent manner was difficult at best. Over time the need for container-based workloads that kept state became a reality and, in select cases, might be more performant. Kubernetes adapted over many iterations to not only allow for storage volumes mounted into the pod, but those volumes being managed by Kubernetes directly was an important component in orchestration of storage with the workloads that require it.

If the ability to mount an external volume to the container was enough, many more examples of stateful applications running at scale in Kubernetes would exist. The reality is that volume mounting is the easy component in the grand scheme of stateful applications. The majority of applications that require state to be maintained after node failure are complicated data-state engines such as relational database systems, distributed key/value stores, and complicated document management systems. This class of applications requires more coordination between how members of the clustered application communicate with one another, how the members are identified, and the order in which members either appear or disappear into the system.

This chapter focuses on best practices for managing state, from simple patterns such as saving a file to a network share, to complex data management systems like MongoDB, mySQL, or Kafka. There is a small section on a new pattern for complex systems called Operators that brings not only Kubernetes primitives, but allows for business or application logic to be added as custom controllers that can help make operating complex data management systems easier.

## Volumes and Volume Mounts

Not every workload that requires a way to maintain state needs to be a complex database or high throughput data queue service. Often, applications that are being moved to containerized workloads expect certain directories to exist and read and write pertinent information to those directories. The ability to inject data into a volume that can be read by containers in a pod is covered in Chapter 5; however, data mounted from ConfigMaps or secrets is usually read-only, and this section focuses on giving containers volumes that can be written to and will survive a container failure or, even better, a pod failure.

Every major container runtime, such as Docker, rkt, CRI-O, and even Singularity, allows for mounting volumes into a container that is mapped to an external storage system. At its simplest, external storage can be a memory location, a path on the container's host, or an external filesystem such as NFS, Glusterfs, CIFS, or Ceph. Why would this be needed, you might wonder? A useful example is that of a legacy application that was written to log application-specific information to a local filesystem. There are many possible solutions including, but not limited to, updating the application code to log out to a `stdout` or `stderr` of a sidecar container that can stream log data to an outside source via a shared pod volume or using a host-based logging tool that can read a volume for both host logs and container application logs. The last scenario can be attained by using a volume mount in the container using a Kubernetes `hostPath` mount, as shown in the following:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-webserver
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-webserver
  template:
    metadata:
      labels:
        app: nginx-webserver
    spec:
      containers:
      - name: nginx-webserver
        image: nginx:alpine
        ports:
        - containerPort: 80
        volumeMounts:
          - name: hostvol
            mountPath: /usr/share/nginx/html
      volumes:
        - name: hostvol
```

```
    hostPath:
      path: /home/webcontent
```

## Volume Best Practices

- Try to limit the use of volumes to pods requiring multiple containers that need to share data, for example adapter or ambassador type patterns. Use the `emptyDir` for those types of sharing patterns.
- Use `hostDir` when access to the data is required by node-based agents or services.
- Try to identify any services that write their critical application logs and events to local disk, and if possible change those to `stdout` or `stderr` and let a true Kubernetes-aware log aggregation system stream the logs instead of leveraging the volume map.

# Kubernetes Storage

The examples so far show basic volume mapping into a container in a pod, which is just a basic container engine capability. The real key is allowing Kubernetes to manage the storage backing the volume mounts. This allows for more dynamic scenarios where pods can live and die as needed, and the storage backing the pod will transition accordingly to wherever the pod may live. Kubernetes manages storage for pods using two distinct APIs, the PersistentVolume and PersistentVolumeClaim.

## PersistentVolume

It is best to think of a PersistentVolume as a disk that will back any volumes that are mounted to a pod. A PersistentVolume will have a claim policy that will define the scope of life of the volume independent of the life cycle of the pod that uses the volume. Kubernetes can use either dynamic or statically defined volumes. To allow for dynamically created volumes, there must be a StorageClass defined in Kubernetes. PersistentVolumes can be created in the cluster of varying types and classes, and only when a PersistentVolumeClaim matches the PersistentVolume will it actually be assigned to a pod. The volume itself is backed by a volume plug-in. There are numerous plug-ins supported directly in Kubernetes, and each has different configuration parameters to adjust:

```
apiVersion: v1
kind: PersistentVolume
metadata:
name: pv001
labels:
  tier: "silver"
spec:
```

```
  capacity:
    storage: 5Gi
  accessModes:
  - ReadWriteMany
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: nfs
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /tmp
    server: 172.17.0.2
```

## PersistentVolumeClaims

PersistentVolumeClaims are a way to give Kubernetes a resource requirement definition for storage that a pod will use. Pods will reference the claim, and then if a `persistentVolume` that matches the claim request exists, it will allocate that volume to that specific pod. At minimum, a storage request size and access mode must be defined, but a specific StorageClass can also be defined. Selectors can also be used to match certain PersistentVolumes that meet a certain criteria will be allocated:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  storageClass: nfs
    accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
  selector:
    matchLabels:
      tier: "silver"
```

The preceding claim will match the PersistentVolume created earlier because the storage class name, the selector match, the size, and the access mode are all equal.

Kubernetes will match up the PersistentVolume with the claim and bind them together. Now to use the volume, the `pod.spec` should just reference the claim by name, as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-webserver
spec:
  replicas: 3
  selector:
```

```
    matchLabels:
      app: nginx-webserver
  template:
    metadata:
      labels:
        app: nginx-webserver
    spec:
      containers:
      - name: nginx-webserver
        image: nginx:alpine
        ports:
        - containerPort: 80
        volumeMounts:
          - name: hostvol
            mountPath: /usr/share/nginx/html
      volumes:
        - name: hostvol
          persistentVolumeClaim:
            claimName: my-pvc
```

## Storage Classes

Instead of manually defining the PersistentVolumes ahead of time, administrators might elect to create StorageClass objects, which define the volume plug-in to use and any specific mount options and parameters that all PersistentVolumes of that class will use. This then allows the claim to be defined with the specific StorageClass to use, and Kubernetes will dynamically create the PersistentVolume based on the StorageClass parameters and options:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
name: nfs
provisioner: cluster.local/nfs-client-provisioner
parameters:
  archiveOnDelete: True
```

Kubernetes also allows operators to create a default storage class using the DefaultStorageClass admission plug-in. If this has been enabled on the API server, then a default StorageClass can be defined and any PersistentVolumeClaims that do not explicitly define a StorageClass. Some cloud providers will include a default storage class to map to the cheapest storage allowed by their instances.

### Container Storage Interface and FlexVolume

Often referred to as "Out-of-Tree" volume plug-ins, the Container Storage Interface (CSI) and FlexVolume enable storage vendors to create custom storage plug-ins without the need to wait for direct code additions to the Kubernetes code base like most volume plug-ins today.

The CSI and FlexVolume plug-ins are deployed on Kubernetes clusters as extensions by operators and can be updated by the storage vendors when needed to expose new functionality.

The CSI states its objective on GitHub as:

> To define an industry standard Container Storage Interface that will enable storage vendors (SP) to develop a plug-in once and have it work across a number of container orchestration (CO) systems.

The FlexVolume interface has been the traditional method used to add additional features for a storage provider. It does require specific drivers to be installed on all of the nodes of the cluster that will use it. This basically becomes an executable that is installed on the hosts of the cluster. This last component is the main detractor to using FlexVolumes, especially in managed service providers, because access to the nodes is frowned upon and the masters practically impossible. The CSI plug-in solves this by basically exposing the same functionality and being as easy to use as deploying a pod into the cluster.

## Kubernetes Storage Best Practices

Cloud native application design principles try to enforce stateless application design as much as possible; however, the growing footprint of container-based services has created the need for data storage persistence. These best practices around storage in Kubernetes in general will help to design an effective approach to providing the required storage implementations to the application design:

- If possible, enable the DefaultStorageClass admission plug-in and define a default storage class. Many times, Helm charts for applications that require PersistentVolumes default to a `default` storage class for the chart, which allows the application to be installed without too much modification.
- When designing the architecture of the cluster, either on-premises or in a cloud provider, take into consideration zone and connectivity between the compute and data layers using the proper labels for both nodes and PersistentVolumes, and using affinity to keep the data and workload as close as possible. The last thing you want is a pod on a node in zone A trying to mount a volume that is attached to a node in zone B.
- Consider very carefully which workloads require state to be maintained on disk. Can that be handled by an outside service like a database system or, if running in a cloud provider, by a hosted service that is API consistent with currently used APIs, say a mongoDB or mySQL as a service?
- Determine how much effort would be involved in modifying the application code to be more stateless.

- While Kubernetes will track and mount the volumes as workloads are scheduled, it does not yet handle redundancy and backup of the data that is stored in those volumes. The CSI specification has added an API for vendors to plug in native snapshot technologies if the storage backend can support it.
- Verify the proper life cycle of the data that volumes will hold. By default the reclaim policy is set to for dynamically provisioned persistentVolumes which will delete the volume from the backing storage provider when the pod is deleted. Sensitive data or data that can be used for forensic analysis should be set to reclaim.

## Stateful Applications

Contrary to popular belief, Kubernetes has supported stateful applications since its infancy, from mySQL, Kafka, and Cassandra to other technologies. Those pioneering days, however, were fraught with complexities and were usually only for small workloads with lots of work required to get things like scaling and durability to work.

To fully grasp the critical differences, you must understand how a typical ReplicaSet schedules and manages pods, and how each could be detrimental to traditional stateful applications:

- Pods in a ReplicaSet are scaled out and assigned random names when scheduled.
- Pods in a ReplicaSet are scaled down in an arbitrary manner.
- Pods in a ReplicaSet are never called directly through their name or IP address but through their association with a `Service`.
- Pods in a ReplicaSet can be restarted and moved to another node at any time.
- Pods in a ReplicaSet that have a PersistentVolume mapped are linked only by the claim, but any new pod with a new name can take over the claim if needed when rescheduled.

Those that have only cursory knowledge of cluster data management systems can immediately begin to see issues with these characteristics of ReplicaSet-based pods. Imagine a pod that has the current writable copy of the database just all of a sudden getting deleted! Pure pandemonium would ensue for sure.

Most neophytes to the Kubernetes world assume that StatefulSet applications are automatically database applications and therefore equate the two things. This could not be further from the truth in the sense that Kubernetes has no sense of what type of application it is deploying. It does not know that your database system requires leader election processes, that it can or cannot handle data replication between members of the set, or, for that matter, that it is a database system at all. This is where StatefulSets come in to play.

## StatefulSets

What StatefulSets do is make it easier to run applications systems that expect more reliable node/pod behavior. If we look at the list of typical pod characteristics in a ReplicaSet, StatefulSets offer almost the complete opposite. The original spec back in Kubernetes version 1.3 called `PetSets` was introduced to answer some of the critical scheduling and management needs for stateful-type applications such as complex data management systems:

- Pods in a StatefulSet are scaled out and assigned sequential names. As the set scales up, the pods get ordinal names, and by default a new pod must be fully online (pass its liveness and/or readiness probes) before the next pod is added.
- Pods in a StatefulSet are scaled down in reverse sequence.
- Pods in a StatefulSet can be addressed individually by name behind a headless Service.
- Pods in a StatefulSet that require a volume mount must use a defined PersistentVolume template. Volumes claimed by pods in a StatefulSet are not deleted when the StatefulSet is deleted.

A StatefulSet specification looks very similar to a Deployment except for the Service declaration and the PersistentVolume template. The headless Service should be created first, which defines the Service that the pods will be addressed with individually. The headless Service is the same as a regular Service but does not do the normal load balancing:

```
apiVersion: v1
kind: Service
metadata:
  name: mongo
  labels:
    name: mongo
spec:
  ports:
  - port: 27017
    targetPort: 27017
  clusterIP: None #This creates the headless Service
  selector:
    role: mongo
```

The StatefulSet definition will also look exactly like a Deployment with a few changes:

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: mongo
spec:
  serviceName: "mongo"
```

```
  replicas: 3
  template:
    metadata:
      labels:
        role: mongo
        environment: test
    spec:
      terminationGracePeriodSeconds: 10
      containers:
        - name: mongo
          image: mongo:3.4
          command:
            - mongod
            - "--replSet"
            - rs0
            - "--bind_ip"
            - 0.0.0.0
            - "--smallfiles"
            - "--noprealloc"
          ports:
            - containerPort: 27017
          volumeMounts:
            - name: mongo-persistent-storage
              mountPath: /data/db
        - name: mongo-sidecar
          image: cvallance/mongo-k8s-sidecar
          env:
            - name: MONGO_SIDECAR_POD_LABELS
              value: "role=mongo,environment=test"
  volumeClaimTemplates:
  - metadata:
      name: mongo-persistent-storage
      annotations:
        volume.beta.kubernetes.io/storage-class: "fast"
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 2Gi
```

## Operators

StatefulSets has definitely been a major factor in introducing complex stateful data systems as feasible workloads in Kubernetes. The only real issue is, as stated earlier, Kubernetes does not really understand the workload that is running in the StatefulSet. All of the other complex operations, like backups, failover, leader registration, new replica registration, and upgrades, are all operations that need to happen quite regularly and will require some careful consideration when running as StatefulSets.

Early on in the growth of Kubernetes, CoreOS site reliability engineers (SREs) created a new class of cloud native software for Kubernetes called Operators. The original

intent was to encapsulate the application domain-specific knowledge of running a specific application into a specific controller that extends Kubernetes. Imagine building up on the StatefulSet controller to be able to deploy, scale, upgrade, backup, and run general maintenance operations on Cassandra or Kafka. Some of the first Operators that were created were for etcd and Prometheus, which uses a time series database to keep metrics over time. The proper creation, backup, and restore configuration of Prometheus or etcd instances can be handled by an Operator and are basically new Kubernetes-managed objects just like a pod or Deployment.

Until recently, Operators have been one-off tools created by SREs or by software vendors for their specific application. In mid-2018, RedHat created the Operator Framework, which is a set of tools including an SDK life cycle manager and future modules that will enable features such as metering, marketplace, and registry type functions. Operators are not only for stateful applications, but because of their custom controller logic they are definitely more amenable to complex data services and stateful systems.

Operators are still an emerging technology in the Kubernetes space, but they are slowly taking a foothold with many data management system vendors, cloud providers, and SREs the world over who want to include some of the operational knowledge they have in running complex distributed systems in Kubernetes. Take a look at OperatorHub for an updated list of curated Operators.

## StatefulSet and Operator Best Practices

Large distributed applications that require state and possibly complicated management and configuration operations benefit from Kubernetes StatefulSets and Operators. Operators are still evolving, but they have the backing of the community at large, so these best practices are based on current capabilities at the time of publication:

- The decision to use Statefulsets should be taken judiciously because usually stateful applications require much deeper management that the orchestrator cannot really manage well yet (read the "Operators" on page 221 section for the possible future answer to this deficiency in Kubernetes).
- The headless Service for the StatefulSet is not automatically created and must be created at deployment time to properly address the pods as individual nodes.
- When an application requires ordinal naming and dependable scaling, it does not always mean it requires the assignment of PersistentVolumes.
- If a node in the cluster becomes unresponsive, any pods that are part of a StatefulSet are not not automatically deleted; they instead will enter a `Terminating` or `Unkown` state after a grace period. The only way to clear this pod is to remove the node object from the cluster, the kubelet beginning to work again and deleting the pod directly, or an Operator force deleting the pod. The force delete should

be the last option and great care should be taken that the node that had the deleted pod does not come back online, because there will now be two pods with the same name in the cluster. You can use `kubectl delete pod nginx-0 --grace-period=0 --force` to force delete the pod.

- Even after force deleting a pod, it might stay in an `Unknown` state, so a patch to the API server will delete the entry and cause the StatefulSet controller to create a new instance of the deleted pod: `kubectl patch pod nginx-0 -p '{"metadata":{"finalizers":null}}'`.
- If you're running a complex data system with some type of leader election or data replication confirmation processes, use `preStop hook` to properly close any connections, force leader election, or verify data synchronization before the pod is deleted using a graceful shutdown process.
- When the application that requires stateful data is a complex data management system, it might be worth a look to determine whether an Operator exists to help manage the more complicated life cycle components of the application. If the application is built in-house, it might be worth investigating whether it would be useful to package the application as an Operator to add additional manageability to the application. Look at the CoreOS Operator SDK for an example.

# Summary

Most organizations look to containerize their stateless applications and leave the stateful applications as is. As more and more cloud native applications run in cloud provider Kubernetes offerings, data gravity becomes an issue. Stateful applications require much more due diligence, but the reality of running them in clusters has been accelerated by the introduction of StatefulSets and Operators. Mapping volumes into containers allow Operators to abstract the storage subsystem specifics away from any application development. Managing stateful applications such as database systems in Kubernetes is still a complex distributed system and needs to be carefully orchestrated using the native Kubernetes primitives of pods, ReplicaSets, Deployments, and StatefulSets, but using Operators that have specific application knowledge built into them as Kubernetes-native APIs may help to elevate these systems into production-based clusters.

CHAPTER 17

# Admission Control and Authorization

Controlling access to the Kubernetes API is key to ensuring that your cluster is not only secured but also can be used as a means to impart policy and governance for all users, workloads, and components of your Kubernetes cluster. In this chapter, we share how you can use admission controllers and authorization modules to enable specific features and how you can customize them to suit your specific needs.

Figure 17-1 provides insight on how and where admission control and authorization take place. It depicts the end-to-end request flow through the Kubernetes API server until the object, if accepted, is saved to storage.

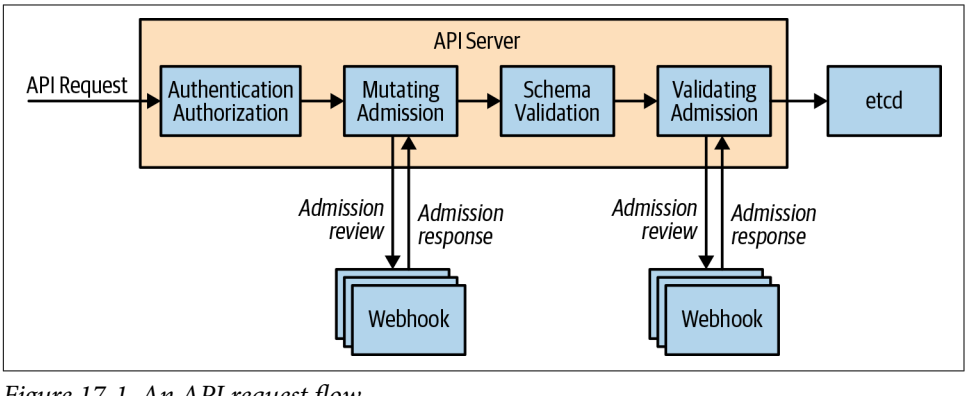



*Figure 17-1. An API request flow*

## Admission Control

Have you ever wondered how namespaces are automatically created when you define a resource in a namespace that doesn't already exist? Maybe you've wondered how a default storage class is selected? These changes are powered by a little-known feature

called *admission controllers*. In this section, we take a look at how you can use admission controllers to implement Kubernetes best practices on the server side on behalf of the user and how we can utilize admission control to govern how a Kubernetes cluster is used.

## What Are They?

Admission controllers sit in the path of the Kubernetes API server request flow and receive requests following the authentication and authorization phases. They are used to either validate or mutate (or both) the request object before saving it to storage. The difference between validating and mutating admission controllers is that mutating can modify the request object they admit, whereas validating cannot.

## Why Are They Important?

Given that admission controllers sit in the path of all API server requests, you can use them in a variety of different ways. Most commonly, admission controller usage can be grouped into the following three groups:

*Policy and governance*
: Admission controllers allow policy to be enforced in order to meet business requirements; for example:

  - Only internal cloud load balancers can be used when in the `dev` namespace.
  - All containers in a pod must have resource limits.
  - Add predefined standard labels or annotations to all resources in order to make them discoverable to existing tools.
  - All Ingress resources only use HTTPS. For more details on how to use admission webhooks in this context, see Chapter 11.

*Security*
: You can use admission controllers to enforce a consistent security posture across your cluster. A canonical example is the PodSecurityPolicy admission controller, which enables controls on security-sensitive fields of the pod specification, for example, denying privileged containers or usage of specific paths from the host filesystem. You can enforce more granular or custom security rules using admission webhooks.

*Resource management*
: Admission controllers allow you to validate in order to provide best practices for your cluster users, for example:

  - Ensure all ingress fully qualified domain names (FQDN) fall within a specific suffix.

- Ensure ingress FQDNs don't overlap.
- All containers in a pod must have resource limits.

## Admission Controller Types

There are two classes of admission controllers: *standard* and *dynamic*. Standard admission controllers are compiled into the API server and are shipped as plug-ins with each Kubernetes release; they need to be configured when the API server is started. Dynamic controllers, on the other hand, are configurable at runtime and are developed outside the core Kubernetes codebase. The only type of dynamic admission control is admission webhooks, which receive admission requests via HTTP callbacks.

Kubernetes ships with more than 30 admission controllers, which are enabled via the following flag on the Kubernetes API server:

```
--enable-admission-plugins
```

Many of the features that ship with Kubernetes depend on the enablement of specific standard admission controllers and, as such, there is a recommended set of defaults:

```
--enable-admission-
plugins=NamespaceLifecycle,LimitRanger,ServiceAccount,DefaultStorage-
Class,DefaultTolerationSeconds,MutatingAdmissionWebhook,ValidatingAdmissionWebho
ok,Priority,ResourceQuota,PodSecurityPolicy
```

You can find the list of Kubernetes admission controllers and their functionality in the Kubernetes documentation.

You might have noticed the following from the list of recommended admission controllers to enable: "MutatingAdmissionWebhook,ValidatingAdmissionWebhook." These standard admission controllers don't implement any admission logic themselves; rather, they are used to configure a webhook endpoint running in-cluster to forward the admission request object.

## Configuring Admission Webhooks

As previously mentioned, one of the main advantages of admission webhooks is that they are dynamically configurable. It is important that you understand how to effectively configure admission webhooks because there are implications and trade-offs when it comes to consistency and failure modes.

The snippet that follows is a ValidatingWebhookConfiguration resource manifest. This manifest is used to define a validating admission webhook. The snippet provides detailed descriptions on the function of each field:

```
apiVersion: admissionregistration.k8s.io/v1beta1
  kind: ValidatingWebhookConfiguration
```

```
  metadata:
    name: ## Resource name
  webhooks:
  - name: ## Admission webhook name, which will be shown to the user when any
admission reviews are denied
    clientConfig:
      service:
        namespace: ## The namespace where the admission webhook pod resides
        name: ## The service name that is used to connect to the admission
          webhook
       path: ## The webhook URL
      caBundle: ## The PEM encoded CA bundle which will be used to validate the
webhook's server certificate
    rules: ## Describes what operations on what resources/subresources the API
server must send to this webhook
    - operations:
      - ## The specific operation that triggers the API server to send to this
webhook (e.g., create, update, delete, connect)
      apiGroups:
      - ""
      apiVersions:
      - "*"
      resources:
      - ## Specific resources by name (e.g., deployments, services, ingresses)
    failurePolicy: ## Defines how to handle access issues or unrecognized
errors, and must be Ignore or Fail
```

For completeness, let's take a look at a MutatingWebhookConfiguration resource manifest. This manifest defines a mutating admission webhook. The snippet provides detailed descriptions on the function of each field:

```
apiVersion: admissionregistration.k8s.io/v1beta1
  kind: MutatingWebhookConfiguration
  metadata:
    name: ## Resource name
  webhooks:
  - name: ## Admission webhook name, which will be shown to the user when any
admission reviews are denied
    clientConfig:
      service:
        namespace: ## The namespace where the admission webhook pod resides
        name: ## The service name that is used to connect to the admission web
hook
       path: ## The webhook URL
      caBundle: ## The PEM encoded CA bundle which will be used to validate the
webhook's server certificate
    rules: ## Describes what operations on what resources/subresources the API
server must send to this webhook
    - operations:
      - ## The specific operation that triggers the API server to send to this
webhook (e.g., create, update, delete, connect)
      apiGroups:
```

```
      - ""
      apiVersions:
      - "*"
      resources:
      - ## Specific resources by name (e.g., deployments, services, ingresses)
    failurePolicy: ## Defines how to handle access issues or unrecognized
errors, and must be Ignore or Fail
```

You might have noticed that both resources are identical, with the exception of the `kind` field. There is one difference on the backend, however: MutatingWebhookConfiguration allows the admission webhook to return a modified request object, whereas ValidatingWebhookConfiguration does not. Even still, it is acceptable to define a MutatingWebhookConfiguration and simply validate; there are security considerations that come into play, and you should consider following the *least-privilege rule*.

It is also likely that you thought to yourself, "What happens if I define a ValidatingWebhookConfiguration or MutatingWebhookConfiguration with the resource field under the rule object to be either ValidatingWebhookConfiguration or MutatingWebhookConfiguration?" The good news is that ValidatingAdmissionWebhooks and MutatingAdmissionWebhooks are never called on admission requests for ValidatingWebhookConfiguration and MutatingWebhookConfiguration objects. This is for good reason: you don't want to accidentally put the cluster in an unrecoverable state.

## Admission Control Best Practices

Now that we've covered the power of admission controllers, here are our best practices to help you make the most of using them:

- Admission plug-in ordering doesn't matter. In earlier versions of Kubernetes, the ordering of the admission plug-ins was specific to the processing order; hence it mattered. In current supported Kubernetes versions, the ordering of the admission plug-ins as specified as API server flags via `--enable-admission-plugins` no longer matters. Ordering does, however, play a small role when it comes to admission webhooks, so it's important to understand the request flow in this case. Request admittance or rejection operates as a logical AND, meaning if any of the admission webhooks reject a request, the entire request is rejected and an error is sent back to the user. It's also important to note that mutating admission controllers are always run prior to running validating admission controllers. If you think about it, this makes good sense: you probably don't want to validate

objects that you are going to subsequently modify. Figure 17-2 illustrates a request flow via admission webhooks.

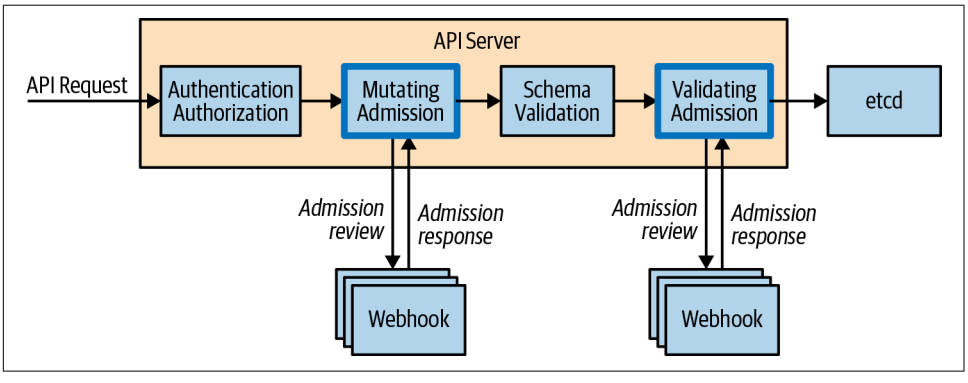



*Figure 17-2. An API request flow via admission webhooks*

- Don't mutate the same fields. Configuring multiple mutating admission webhooks also presents challenges. There is no way to order the request flow through multiple mutating admission webhooks, so it's important to not have mutating admission controllers modify the same fields, because this can result in unexpected results. In the case where you have multiple mutating admission webhooks, we generally recommend configuring validating admission webhooks to confirm that the final resource manifest is what you expect post-mutation because it's guaranteed to be run following mutating webhooks.
- Fail open/fail closed. You might recall seeing the `failurePolicy` field as part of both the mutating and validating webhook configuration resources. This field defines how the API server should proceed in the case where the admission webhooks have access issues or encounter unrecognized errors. You can set this field to either `Ignore` or `Fail`. `Ignore` essentially fails to open, meaning that processing of the request will continue, whereas `Fail` denies the entire request. This might seem obvious, but the implications in both cases require consideration. Ignoring a critical admission webhook could result in policy that the business relies on not being applied to a resource without the user knowing.

  One potential solution to protect against this would be to raise an alert when the API server logs that it cannot reach a given admission webhook. `Fail` can be even more devastating by denying all requests if the admission webhook is experiencing issues. To protect against this you can scope the rules to ensure that only specific resource requests are set to the admission webhook. As a tenet, you should never have any rules that apply to all resources in the cluster.
- If you have written your own admission webhook, it's important to remember that user/system requests can be directly affected by the time it takes for your

admission webhook to make a decision and respond. All admission webhook calls are configured with a 30-second timeout, after which time the `failurePolicy` takes effect. Even if it takes several seconds for your admission webhook to make an admit/deny decision, it can severely affect user experience when working with the cluster. Avoid having complex logic or relying on external systems such as databases in order to process the admit/deny logic.

- Scoping admission webhooks. There is an optional field that allows you to scope the namespaces in which the admission webhooks operate on via the `NamespaceSelector` field. This field defaults to empty, which matches everything, but can be used to match namespace labels via the use of the `matchLabels` field. We recommend that you always use this field because it allows for an explicit opt-in per namespace.
- The `kube-system` namespace is a reserved namespace that's common across all Kubernetes clusters. It's where all system-level services operate. We recommend never running admission webhooks against the resources in this namespace specifically, and you can achieve this by using the `NamespaceSelector` field and simply not matching the `kube-system` namespace. You should also consider it on any system-level namespaces that are required for cluster operation.
- Lock down admission webhook configurations with RBAC. Now that you know about all the fields in the admission webhook configuration, you have probably thought of a really simple way to break access to a cluster. It goes without saying that the creation of both a MutatingWebhookConfiguration and ValidatingWebhookConfiguration is a root-level operation on the cluster and must be locked down appropriately using RBAC. Failure to do so can result in a broken cluster or, even worse, an injection attack on your application workloads.
- Don't send sensitive data. Admission webhooks are essentially black boxes that accept AdmissionRequests and output AdmissionResponses. How they store and manipulate the request is opaque to the user. It's important to think about what request payloads you are sending to the admission webhook. In the case of Kubernetes secrets or ConfigMaps, they might contain sensitive information and require strong guarantees about how that information is stored and shared. Sharing these resources with an admission webhook can leak sensitive information, which is why you should scope your resource rules to the minimum resource needed to validate and/or mutate.

# Authorization

We often think about authorization in the context of answering the following question: "Is this user able to perform these actions on these resources?" In Kubernetes, the authorization of each request is performed after authentication but before admission. In this section, we explore how you can configure different authorization

modules and better understand how you can create the appropriate policy to serve the needs of your cluster. Figure 17-3 illustrates where authorization sits in the request flow.

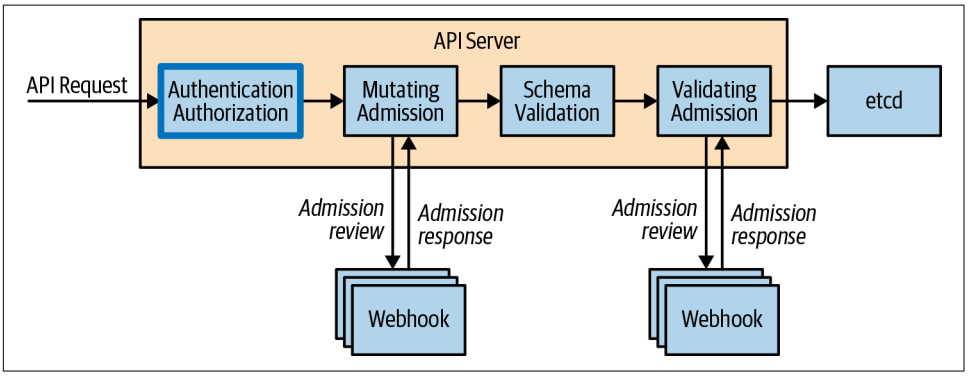



*Figure 17-3. API request flow via authorization modules*

## Authorization Modules

Authorization modules are responsible for either granting or denying permission to access. They determine whether to grant access based on policy that must be explicitly defined; otherwise all requests will be implicitly denied.

As of version 1.15, Kubernetes ships with the following authorization modules out of the box:

*Attribute-Based Access Control (ABAC)*
: Allows authorization policy to be configured via local files

*RBAC*
: Allows authorization policy to be configured via the Kubernetes API (refer to Chapter 4)

*Webhook*
: Allows the authorization of a request to be handled via a remote REST endpoint

*Node*
: Specialized authorization module that authorizes requests from kubelets

The modules are configured by the cluster administrator via the following flag on the API server: `--authorization-mode`. Multiple modules can be configured and are checked in order. Unlike admission controllers, if a single authorization module admits the request, the request can proceed. Only for the case in which all modules deny the request will an error be returned to the user.

### ABAC

Let's take a look at a policy definition in the context of using the ABAC authorization module. The following grants user Mary read-only access to a pod in the `kube-system` namespace:

```
apiVersion: abac.authorization.kubernetes.io/v1beta1
kind: Policy
spec:
  user: mary
  resource: pods
  readonly: true
  namespace: kube-system
```

If Mary were to make the following request, it would be denied because Mary doesn't have access to get pods in the `demo-app` namespace:

```
apiVersion: authorization.k8s.io/v1beta1
kind: SubjectAccessReview
spec:
  resourceAttributes:
    verb: get
    resource: pods
    namespace: demo-app
```

This example introduced a new API group, `authorization.k8s.io`. This set of APIs exposes API server authorization to external services and has the following APIs, which are great for debugging:

*SelfSubjectAccessReview*
: Access review for the current user

*SubjectAccessReview*
: Like SelfSubjectAccessReview but for any user

*LocalSubjectAccessReview*
: Like SubjectAccessReview but namespace specific

*SelfSubjectRulesReview*
: Returns a list of actions a user can perform in a given namespace

The really cool part is that you can query these APIs by creating resources as you typically would. Let's actually take the previous example and test this for ourselves using the SelfSubjectAccessReview. The status field in the output indicates that this request is allowed:

```
$ cat << EOF | kubectl create -f - -o yaml
apiVersion: authorization.k8s.io/v1beta1
kind: SelfSubjectAccessReview
spec:
  resourceAttributes:
```

```
    verb: get
    resource: pods
    namespace: demo-app
EOF
apiVersion: authorization.k8s.io/v1beta1
kind: SelfSubjectAccessReview
metadata:
  creationTimestamp: null
spec:
  resourceAttributes:
    namespace: kube-system
    resource: pods
    verb: get
status:
  allowed: true
```

In fact, Kubernetes ships with tooling built into `kubectl` to make this even easier. The `kubectl auth can-i` command operates by querying the same API as the previous example:

```
$ kubectl auth can-i get pods --namespace demo-app
yes
```

With administrator credentials, you can also run the same command to check actions as another user:

```
$ kubectl auth can-i get pods --namespace demo-app --as mary
yes
```

#### RBAC

Kubernetes role-based access control is covered in depth in Chapter 4.

#### Webhook

Using the webhook authorization module allows a cluster administrator to configure an external REST endpoint to delegate the authorization process to. This would run off cluster and be reachable via URL. The configuration of the REST endpoint is found in a file on the master filesystem and configured on the API server via `--authorization-webhook-config-file=SOME_FILENAME`. After you've configured it, the API server will send SubjectAccessReview objects as part of the request body to the authorization webhook application, which processes and returns the object with the status field complete.

## Authorization Best Practices

Consider the following best practices before making changes to the authorization modules configured on your cluster:

- Given that the ABAC policies need to be placed on the filesystem of each master node and kept synchronized, we generally recommend *against* using ABAC in multimaster clusters. The same can be said for the webhook module because the configuration is based on a file and a corresponding flag being present. Furthermore, changes to these policies in the files require a restart of the API server to take effect, which is effectively a control-plane outage in a single master cluster or inconsistent configuration in a multimaster cluster. Given these details, we recommend using only the RBAC module for user authorization because the rules are configured and stored in Kubernetes itself.
- Webhook modules, although powerful, are potentially very dangerous. Given that every request is subject to the authorization process, a failure of a webhook service would be devastating for a cluster. Therefore, we generally recommend not using external authorization modules unless you completely vet and are comfortable with your cluster failure modes if the webhook service becomes unreachable or unavailable.

## Summary

In this chapter, we covered the foundational topics of admission and authorization and covered best practices. Put these skills to use by determining the best admission and authorization configuration that allows you to customize the controls and policies needed for the life of your cluster.

# CHAPTER 18
# Conclusion

The primary strength of Kubernetes is its modularity and generality. Nearly every kind of application that you might want to deploy you can fit within Kubernetes, and no matter what kind of adjustments or tuning you need to make to your system, they're generally possible.

Of course, this modularity and generality come at a cost, and that cost is a reasonable amount of complexity. Understanding how the APIs and components of Kubernetes work is critical to successfully unlocking the power of Kubernetes to make your application development, management, and deployment easier and more reliable.

Likewise, understanding how to link Kubernetes up with a wide variety of external systems and practices as varied as an on-premises database and a Continuous Delivery system is critical to efficiently making use of Kubernetes in the real world.

Throughout this book we have worked to provide concrete real-world experience on specific topics that you will likely encounter whether you are a newcomer to Kubernetes or an experienced administrator. Regardless of whether you are facing a new area in which you need to become an expert, or you simply want a refresher about how others have addressed a familiar problem, hopefully, the chapters in this book have enabled you to learn from our experience. We also hope that in this learning, you gain the skills and confidence to use Kubernetes to its fullest capabilities. Thank you and we look forward to seeing you out in the real world!

# Index

## D



## E



## F



## G

## H



## I



## J



## K

## L



## M



## N

## S

## T



## U

## About the Authors

**Brendan Burns** is a distinguished engineer at Microsoft Azure and cofounder of the Kubernetes open source project. He's been building cloud applications for more than a decade.

**Eddie Villalba** is a software engineer with Microsoft's Commercial Software Engineering division, focusing on open source cloud and Kubernetes. He's helped many real-world users adopt Kubernetes for their applications.

**Dave Strebel** is a global cloud native architect at Microsoft Azure focusing on open source cloud and Kubernetes. He's deeply involved in the Kubernetes open source project, helping with the Kubernetes release team and leading SIG-Azure.

**Lachlan Evenson** is a principal program manager on the container compute team at Microsoft Azure. He's helped numerous people onboard to Kubernetes through both hands-on teaching and conference talks.

## Colophon

The animal on the cover of *Kubernetes Best Practices* is an Old World mallard duck (*Anas platyrhynchos*), a kind of dabbling duck that feeds on the surface of water rather than diving for food. Species of Anas are typically separated by their ranges and behavioral cues; however, mallards frequently interbreed with other species, which has introduced some fully fertile hybrids.

Mallard ducklings are precocial and capable of swimming as soon as they hatch. Juveniles begin flying between three and four months of age. They reach full maturity at 14 months and have an average life expectancy of 3 years.

The mallard is a medium-sized duck that is just slightly heavier than most dabbling ducks. Adults average 23 inches long with a wingspan of 36 inches, and weigh 2.5 pounds. Ducklings have yellow and black plumage. At around six months of age, males and females can be distinguished visually as their coloring changes. Males have green head feathers, a white collar, purple-brown breast, gray-brown wings, and a yellowish-orange bill. Females are mottled brown, which is the color of most female dabbling ducks.

Mallards have a wide range of habitats across both northern and southern hemispheres. They are found in fresh- and salt-water wetlands, from lakes to rivers to seashores. Northern mallards are migratory, and winter father south. The mallard diet is highly variable, and includes plants, seeds, roots, gastropods, invertebrates, and crustaceans.

Brood parasites will target mallard nests. These are species of other birds who may lay their eggs in the mallard nest. If the eggs resemble those of the mallard, the mallard will accept them and raise the hatchlings with their own.

Mallards must contend with a wide variety of predators, most notably foxes and birds of prey such as falcons and eagles. They have also been preyed upon by catfish and pike. Crows, swans, and geese have all been known to attack the ducks over territorial disputes. Unihemispheric sleep (or sleeping with one eye open), which allows one hemisphere of the brain to sleep while the other is awake, was first noted in mallards. It is common among aquatic birds as a predation-avoidance behavior.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Jose Marzan, based on a black and white engraving from *The Animal World*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.