OpenShift Container Platform 4.8

Architecture

An overview of the architecture for OpenShift Container Platform

Last Updated: 2021-10-14

OpenShift Container Platform 4.8 Architecture

An overview of the architecture for OpenShift Container Platform

Legal Notice

Abstract

This document provides an overview of the platform and application architecture in OpenShift Container Platform.

Table of Contents

CHAPTER 1. OPENSHIFT CONTAINER PLATFORM ARCHITECTURE

1.1. INTRODUCTION TO OPENSHIFT CONTAINER PLATFORM

OpenShift Container Platform is a platform for developing and running containerized applications. It is designed to allow applications and the data centers that support them to expand from just a few machines and applications to thousands of machines that serve millions of clients.

With its foundation in Kubernetes, OpenShift Container Platform incorporates the same technology that serves as the engine for massive telecommunications, streaming video, gaming, banking, and other applications. Its implementation in open Red Hat technologies lets you extend your containerized applications beyond a single cloud to on-premise and multi-cloud environments.

1.1.1. About Kubernetes

Although container images and the containers that run from them are the primary building blocks for modern application development, to run them at scale requires a reliable and flexible distribution system. Kubernetes is the defacto standard for orchestrating containers.

Kubernetes is an open source container orchestration engine for automating deployment, scaling, and management of containerized applications. The general concept of Kubernetes is fairly simple:

- Start with one or more worker nodes to run the container workloads.
- Manage the deployment of those workloads from one or more control plane nodes (also known as the master nodes).
- Wrap containers in a deployment unit called a pod. Using pods provides extra metadata with the container and offers the ability to group several containers in a single deployment entity.
- Create special kinds of assets. For example, services are represented by a set of pods and a policy that defines how they are accessed. This policy allows containers to connect to the services that they need even if they do not have the specific IP addresses for the services. Replication controllers are another special asset that indicates how many pod replicas are required to run at a time. You can use this capability to automatically scale your application to adapt to its current demand.

In only a few years, Kubernetes has seen massive cloud and on-premise adoption. The open source development model allows many people to extend Kubernetes by implementing different technologies for components such as networking, storage, and authentication.

1.1.2. The benefits of containerized applications

Using containerized applications offers many advantages over using traditional deployment methods. Where applications were once expected to be installed on operating systems that included all their dependencies, containers let an application carry their dependencies with them. Creating containerized applications offers many benefits.

1.1.2.1. Operating system benefits

Containers use small, dedicated Linux operating systems without a kernel. Their file system, networking, cgroups, process tables, and namespaces are separate from the host Linux system, but the containers can integrate with the hosts seamlessly when necessary. Being based on Linux allows containers to use

all the advantages that come with the open source development model of rapid innovation.

Because each container uses a dedicated operating system, you can deploy applications that require conflicting software dependencies on the same host. Each container carries its own dependent software and manages its own interfaces, such as networking and file systems, so applications never need to compete for those assets.

1.1.2.2. Deployment and scaling benefits

If you employ rolling upgrades between major releases of your application, you can continuously improve your applications without downtime and still maintain compatibility with the current release.

You can also deploy and test a new version of an application alongside the existing version. If the container passes your tests, simply deploy more new containers and remove the old ones.

Since all the software dependencies for an application are resolved within the container itself, you can use a standardized operating system on each host in your data center. You do not need to configure a specific operating system for each application host. When your data center needs more capacity, you can deploy another generic host system.

Similarly, scaling containerized applications is simple. OpenShift Container Platform offers a simple, standard way of scaling any containerized service. For example, if you build applications as a set of microservices rather than large, monolithic applications, you can scale the individual microservices individually to meet demand. This capability allows you to scale only the required services instead of the entire application, which can allow you to meet application demands while using minimal resources.

1.1.3. OpenShift Container Platform overview

OpenShift Container Platform provides enterprise-ready enhancements to Kubernetes, including the following enhancements:

- Hybrid cloud deployments. You can deploy OpenShift Container Platform clusters to a variety of public cloud platforms or in your data center.
- Integrated Red Hat technology. Major components in OpenShift Container Platform come from Red Hat Enterprise Linux (RHEL) and related Red Hat technologies. OpenShift Container Platform benefits from the intense testing and certification initiatives for Red Hat's enterprise quality software.
- Open source development model. Development is completed in the open, and the source code is available from public software repositories. This open collaboration fosters rapid innovation and development.

Although Kubernetes excels at managing your applications, it does not specify or manage platform-level requirements or deployment processes. Powerful and flexible platform management tools and processes are important benefits that OpenShift Container Platform 4.8 offers. The following sections describe some unique features and benefits of OpenShift Container Platform.

1.1.3.1. Custom operating system

OpenShift Container Platform uses Red Hat Enterprise Linux CoreOS (RHCOS), a container-oriented operating system that is specifically designed for running containerized applications from OpenShift Container Platform and works with new tools to provide fast installation, Operator-based management, and simplified upgrades.

RHCOS includes:

- Ignition, which OpenShift Container Platform uses as a firstboot system configuration for initially bringing up and configuring machines.
- CRI-O, a Kubernetes native container runtime implementation that integrates closely with the operating system to deliver an efficient and optimized Kubernetes experience. CRI-O provides facilities for running, stopping, and restarting containers. It fully replaces the Docker Container Engine, which was used in OpenShift Container Platform 3.
- Kubelet, the primary node agent for Kubernetes that is responsible for launching and monitoring containers.

In OpenShift Container Platform 4.8, you must use RHCOS for all control plane machines, but you can use Red Hat Enterprise Linux (RHEL) as the operating system for compute machines, which are also known as worker machines. If you choose to use RHEL workers, you must perform more system maintenance than if you use RHCOS for all of the cluster machines.

1.1.3.2. Simplified installation and update process

With OpenShift Container Platform 4.8, if you have an account with the right permissions, you can deploy a production cluster in supported clouds by running a single command and providing a few values. You can also customize your cloud installation or install your cluster in your data center if you use a supported platform.

For clusters that use RHCOS for all machines, updating, or upgrading, OpenShift Container Platform is a simple, highly-automated process. Because OpenShift Container Platform completely controls the systems and services that run on each machine, including the operating system itself, from a central control plane, upgrades are designed to become automatic events. If your cluster contains RHEL worker machines, the control plane benefits from the streamlined update process, but you must perform more tasks to upgrade the RHEL machines.

1.1.3.3. Other key features

Operators are both the fundamental unit of the OpenShift Container Platform 4.8 code base and a convenient way to deploy applications and software components for your applications to use. In OpenShift Container Platform, Operators serve as the platform foundation and remove the need for manual upgrades of operating systems and control plane applications. OpenShift Container Platform Operators such as the Cluster Version Operator and Machine Config Operator allow simplified, cluster-wide management of those critical components.

Operator Lifecycle Manager (OLM) and the OperatorHub provide facilities for storing and distributing Operators to people developing and deploying applications.

The Red Hat Quay Container Registry is a Quay.io container registry that serves most of the container images and Operators to OpenShift Container Platform clusters. Quay.io is a public registry version of Red Hat Quay that stores millions of images and tags.

Other enhancements to Kubernetes in OpenShift Container Platform include improvements in software defined networking (SDN), authentication, log aggregation, monitoring, and routing. OpenShift Container Platform also offers a comprehensive web console and the custom OpenShift CLI (**oc**) interface.

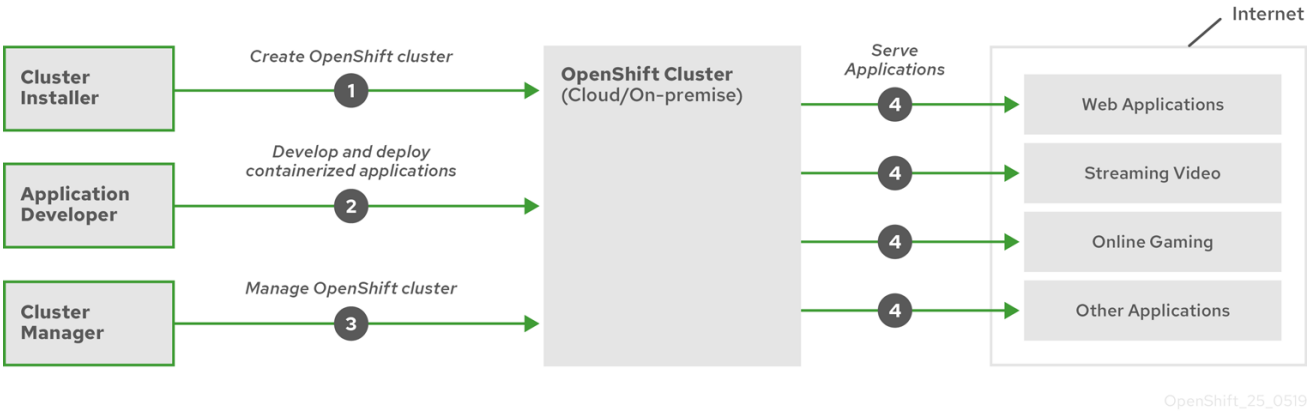
1.1.3.4. OpenShift Container Platform lifecycle

The following figure illustrates the basic OpenShift Container Platform lifecycle:

- Creating an OpenShift Container Platform cluster

- Managing the cluster
- Developing and deploying applications
- Scaling up applications

Figure 1.1. High level OpenShift Container Platform overview



1.1.4. Internet access for OpenShift Container Platform

In OpenShift Container Platform 4.8, you require access to the internet to install your cluster.

You must have internet access to:

- Access the Red Hat OpenShift Cluster Manager page to download the installation program and perform subscription management. If the cluster has internet access and you do not disable Telemetry, that service automatically entitles your cluster.
- Access Quay.io to obtain the packages that are required to install your cluster.
- Obtain the packages that are required to perform cluster updates.

IMPORTANT

If your cluster cannot have direct internet access, you can perform a restricted network installation on some types of infrastructure that you provision. During that process, you download the content that is required and use it to populate a mirror registry with the packages that you need to install a cluster and generate the installation program. With some installation types, the environment that you install your cluster in will not require internet access. Before you update the cluster, you update the content of the mirror registry.

CHAPTER 2. INSTALLATION AND UPDATE

2.1. OPENSHIFT CONTAINER PLATFORM INSTALLATION OVERVIEW

The OpenShift Container Platform installation program offers you flexibility. You can use the installation program to deploy a cluster on infrastructure that the installation program provisions and the cluster maintains or deploy a cluster on infrastructure that you prepare and maintain.

These two basic types of OpenShift Container Platform clusters are frequently called installer-provisioned infrastructure clusters and user-provisioned infrastructure clusters.

Both types of clusters have the following characteristics:

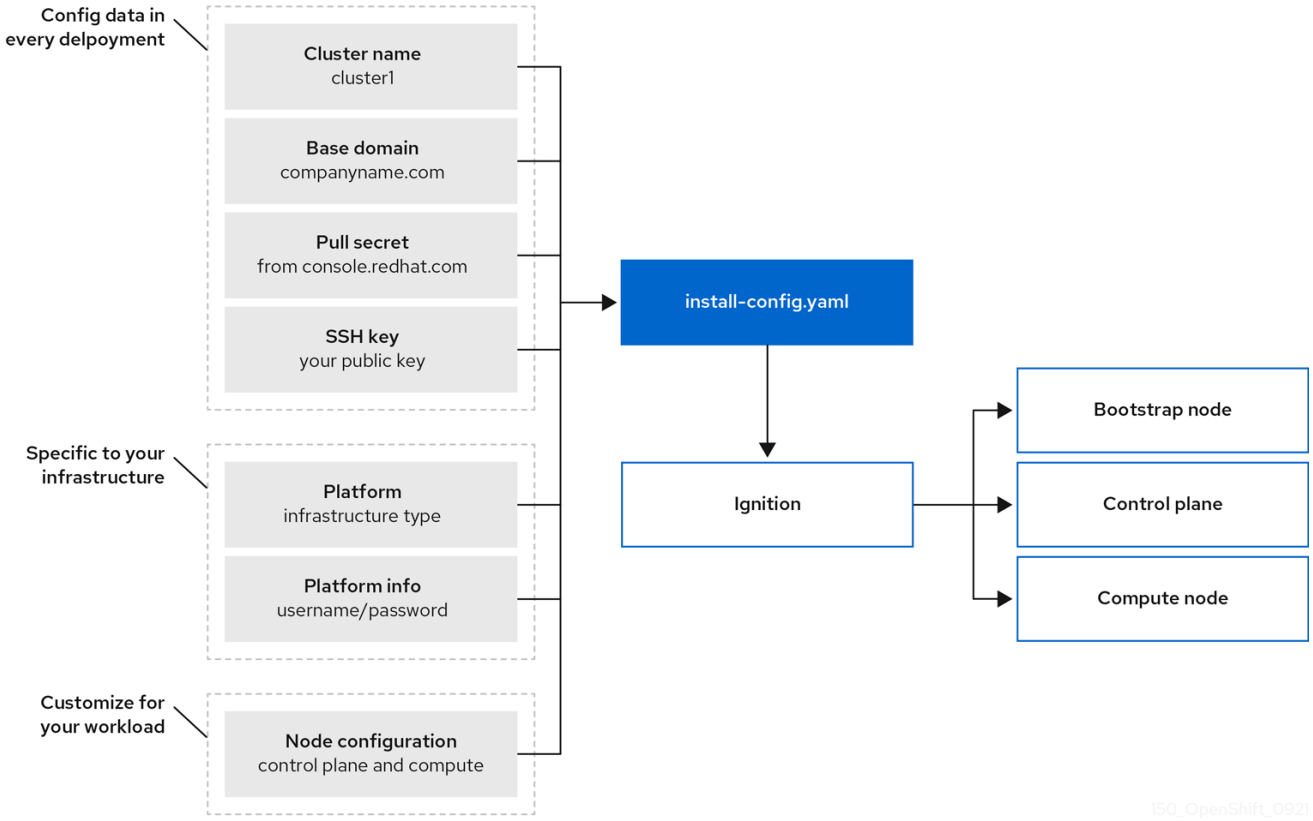
- Highly available infrastructure with no single points of failure is available by default
- Administrators maintain control over what updates are applied and when

You use the same installation program to deploy both types of clusters. The main assets generated by the installation program are the Ignition config files for the bootstrap, master, and worker machines. With these three configurations and correctly configured infrastructure, you can start an OpenShift Container Platform cluster.

The OpenShift Container Platform installation program uses a set of targets and dependencies to manage cluster installation. The installation program has a set of targets that it must achieve, and each target has a set of dependencies. Because each target is only concerned with its own dependencies, the installation program can act to achieve multiple targets in parallel. The ultimate target is a running cluster. By meeting dependencies instead of running commands, the installation program is able to recognize and use existing components instead of running the commands to create them again.

The following diagram shows a subset of the installation targets and dependencies:

Figure 2.1. OpenShift Container Platform installation targets and dependencies



After installation, each cluster machine uses Red Hat Enterprise Linux CoreOS (RHCOS) as the operating system. RHCOS is the immutable container host version of Red Hat Enterprise Linux (RHEL) and features a RHEL kernel with SELinux enabled by default. It includes the **kubelet**, which is the Kubernetes node agent, and the CRI-O container runtime, which is optimized for Kubernetes.

Every control plane machine in an OpenShift Container Platform 4.8 cluster must use RHCOS, which includes a critical first-boot provisioning tool called Ignition. This tool enables the cluster to configure the machines. Operating system updates are delivered as an Atomic OSTree repository that is embedded in a container image that is rolled out across the cluster by an Operator. Actual operating system changes are made in-place on each machine as an atomic operation by using rpm-ostree. Together, these technologies enable OpenShift Container Platform to manage the operating system like it manages any other application on the cluster, via in-place upgrades that keep the entire platform up-to-date. These in-place updates can reduce the burden on operations teams.

If you use RHCOS as the operating system for all cluster machines, the cluster manages all aspects of its components and machines, including the operating system. Because of this, only the installation program and the Machine Config Operator can change machines. The installation program uses Ignition config files to set the exact state of each machine, and the Machine Config Operator completes more changes to the machines, such as the application of new certificates or keys, after installation.

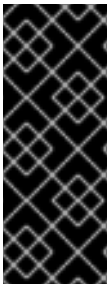
2.1.1. Supported platforms for OpenShift Container Platform clusters

In OpenShift Container Platform 4.8, you can install a cluster that uses installer-provisioned infrastructure on the following platforms:

- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)

- Microsoft Azure
- Red Hat OpenStack Platform (RHOSP) version 13 and 16
    - The latest OpenShift Container Platform release supports both the latest RHOSP long-life release and intermediate release. For complete RHOSP release compatibility, see the OpenShift Container Platform on RHOSP support matrix.
- Red Hat Virtualization (RHV)
- VMware vSphere
- VMware Cloud (VMC) on AWS
- Bare metal

For these clusters, all machines, including the computer that you run the installation process on, must have direct internet access to pull images for platform containers and provide telemetry data to Red Hat.



IMPORTANT

After installation, the following changes are not supported:

- Mixing cloud provider platforms
- Mixing cloud provider components, such as using a persistent storage framework from a differing platform than what the cluster is installed on

In OpenShift Container Platform 4.8, you can install a cluster that uses user-provisioned infrastructure on the following platforms:

- AWS
- Azure
- GCP
- RHOSP
- RHV
- VMware vSphere
- VMware Cloud on AWS
- Bare metal
- IBM Z or LinuxONE
- IBM Power Systems

With installations on user-provisioned infrastructure, each machine can have full internet access, you can place your cluster behind a proxy, or you can perform a *restricted network installation*. In a restricted network installation, you can download the images that are required to install a cluster, place them in a

mirror registry, and use that data to install your cluster. While you require internet access to pull images for platform containers, with a restricted network installation on vSphere or bare metal infrastructure, your cluster machines do not require direct internet access.

The OpenShift Container Platform 4.x Tested Integrations page contains details about integration testing for different platforms.

2.1.2. Installation process

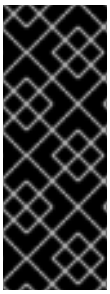
When you install an OpenShift Container Platform cluster, you download the installation program from the appropriate Infrastructure Provider page on the Red Hat OpenShift Cluster Manager site. This site manages:

- REST API for accounts
- Registry tokens, which are the pull secrets that you use to obtain the required components
- Cluster registration, which associates the cluster identity to your Red Hat account to facilitate the gathering of usage metrics

In OpenShift Container Platform 4.8, the installation program is a Go binary file that performs a series of file transformations on a set of assets. The way you interact with the installation program differs depending on your installation type.

- For clusters with installer-provisioned infrastructure, you delegate the infrastructure bootstrapping and provisioning to the installation program instead of doing it yourself. The installation program creates all of the networking, machines, and operating systems that are required to support the cluster.
- If you provision and manage the infrastructure for your cluster, you must provide all of the cluster infrastructure and resources, including the bootstrap machine, networking, load balancing, storage, and individual cluster machines.

You use three sets of files during installation: an installation configuration file that is named **install-config.yaml**, Kubernetes manifests, and Ignition config files for your machine types.



IMPORTANT

It is possible to modify Kubernetes and the Ignition config files that control the underlying RHCOS operating system during installation. However, no validation is available to confirm the suitability of any modifications that you make to these objects. If you modify these objects, you might render your cluster non-functional. Because of this risk, modifying Kubernetes and Ignition config files is not supported unless you are following documented procedures or are instructed to do so by Red Hat support.

The installation configuration file is transformed into Kubernetes manifests, and then the manifests are wrapped into Ignition config files. The installation program uses these Ignition config files to create the cluster.

The installation configuration files are all pruned when you run the installation program, so be sure to back up all configuration files that you want to use again.

IMPORTANT

You cannot modify the parameters that you set during installation, but you can modify many cluster attributes after installation.

The installation process with installer-provisioned infrastructure
The default installation type uses installer-provisioned infrastructure. By default, the installation program acts as an installation wizard, prompting you for values that it cannot determine on its own and providing reasonable default values for the remaining parameters. You can also customize the installation process to support advanced infrastructure scenarios. The installation program provisions the underlying infrastructure for the cluster.

You can install either a standard cluster or a customized cluster. With a standard cluster, you provide minimum details that are required to install the cluster. With a customized cluster, you can specify more details about the platform, such as the number of machines that the control plane uses, the type of virtual machine that the cluster deploys, or the CIDR range for the Kubernetes service network.

If possible, use this feature to avoid having to provision and maintain the cluster infrastructure. In all other environments, you use the installation program to generate the assets that you require to provision your cluster infrastructure.

With installer-provisioned infrastructure clusters, OpenShift Container Platform manages all aspects of the cluster, including the operating system itself. Each machine boots with a configuration that references resources hosted in the cluster that it joins. This configuration allows the cluster to manage itself as updates are applied.

The installation process with user-provisioned infrastructure
You can also install OpenShift Container Platform on infrastructure that you provide. You use the installation program to generate the assets that you require to provision the cluster infrastructure, create the cluster infrastructure, and then deploy the cluster to the infrastructure that you provided.

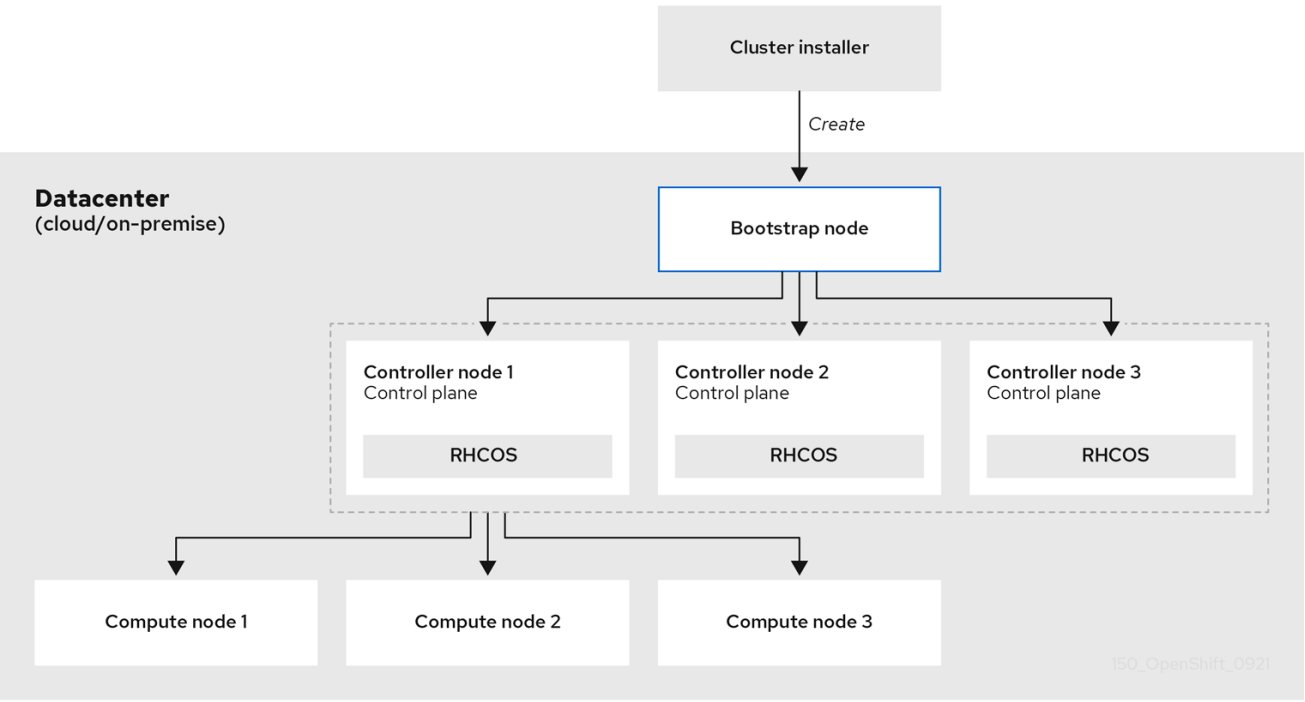
If you do not use infrastructure that the installation program provisioned, you must manage and maintain the cluster resources yourself, including:

- The underlying infrastructure for the control plane and compute machines that make up the cluster
- Load balancers
- Cluster networking, including the DNS records and required subnets
- Storage for the cluster infrastructure and applications

If your cluster uses user-provisioned infrastructure, you have the option of adding RHEL compute machines to your cluster.

Installation process details
Because each machine in the cluster requires information about the cluster when it is provisioned, OpenShift Container Platform uses a temporary *bootstrap* machine during initial configuration to provide the required information to the permanent control plane. It boots by using an Ignition config file that describes how to create the cluster. The bootstrap machine creates the control plane machines (also known as the master machines) that make up the control plane. The control plane machines then create the compute machines, which are also known as worker machines. The following figure illustrates this process:

Figure 2.2. Creating the bootstrap, control plane, and compute machines



After the cluster machines initialize, the bootstrap machine is destroyed. All clusters use the bootstrap process to initialize the cluster, but if you provision the infrastructure for your cluster, you must complete many of the steps manually.

IMPORTANT

The Ignition config files that the installation program generates contain certificates that expire after 24 hours, which are then renewed at that time. If the cluster is shut down before renewing the certificates and the cluster is later restarted after the 24 hours have elapsed, the cluster automatically recovers the expired certificates. The exception is that you must manually approve the pending **node-bootstrapper** certificate signing requests (CSRs) to recover kubelet certificates. See the documentation for *Recovering from expired control plane certificates* for more information.

Bootstrapping a cluster involves the following steps:

1. The bootstrap machine boots and starts hosting the remote resources required for the control plane machines to boot. (Requires manual intervention if you provision the infrastructure)
2. The bootstrap machine starts a single-node etcd cluster and a temporary Kubernetes control plane.
3. The control plane machines fetch the remote resources from the bootstrap machine and finish booting. (Requires manual intervention if you provision the infrastructure)
4. The temporary control plane schedules the production control plane to the production control plane machines.
5. The Cluster Version Operator (CVO) comes online and installs the etcd Operator. The etcd Operator scales up etcd on all control plane nodes.
6. The temporary control plane shuts down and passes control to the production control plane.

7. The bootstrap machine injects OpenShift Container Platform components into the production control plane.

8. The installation program shuts down the bootstrap machine. (Requires manual intervention if you provision the infrastructure)

9. The control plane sets up the compute nodes.

10. The control plane installs additional services in the form of a set of Operators.

The result of this bootstrapping process is a fully running OpenShift Container Platform cluster. The cluster then downloads and configures remaining components needed for the day-to-day operation, including the creation of compute machines in supported environments.

Installation scope

The scope of the OpenShift Container Platform installation program is intentionally narrow. It is designed for simplicity and ensured success. You can complete many more configuration tasks after installation completes.

Additional resources

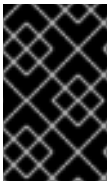
- See Available cluster customizations for details about OpenShift Container Platform configuration resources.

2.2. ABOUT THE OPENSHIFT UPDATE SERVICE

The OpenShift Update Service (OSUS) provides over-the-air updates to OpenShift Container Platform, including Red Hat Enterprise Linux CoreOS (RHCOS). It provides a graph, or diagram, that contains the *vertices* of component Operators and the *edges* that connect them. The edges in the graph show which versions you can safely update to. The vertices are update payloads that specify the intended state of the managed cluster components.

The Cluster Version Operator (CVO) in your cluster checks with the OpenShift Update Service to see the valid updates and update paths based on current component versions and information in the graph. When you request an update, the CVO uses the release image for that update to upgrade your cluster. The release artifacts are hosted in Quay as container images.

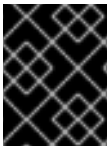
To allow the OpenShift Update Service to provide only compatible updates, a release verification pipeline drives automation. Each release artifact is verified for compatibility with supported cloud platforms and system architectures, as well as other component packages. After the pipeline confirms the suitability of a release, the OpenShift Update Service notifies you that it is available.



IMPORTANT

The OpenShift Update Service displays all recommended updates for your current cluster. If an upgrade path is not recommended by the OpenShift Update Service, it might be because of a known issue with the update or the target release.

Two controllers run during continuous update mode. The first controller continuously updates the payload manifests, applies the manifests to the cluster, and outputs the controlled rollout status of the Operators to indicate whether they are available, upgrading, or failed. The second controller polls the OpenShift Update Service to determine if updates are available.



IMPORTANT

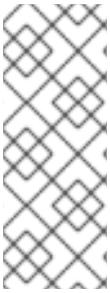
Only upgrading to a newer version is supported. Reverting or rolling back your cluster to a previous version is not supported. If your upgrade fails, contact Red Hat support.

During the upgrade process, the Machine Config Operator (MCO) applies the new configuration to your cluster machines. The MCO cordons the number of nodes as specified by the **maxUnavailable** field on the machine configuration pool and marks them as unavailable. By default, this value is set to **1**. The MCO then applies the new configuration and reboots the machine.

If you use Red Hat Enterprise Linux (RHEL) machines as workers, the MCO does not update the kubelet because you must update the OpenShift API on the machines first.

With the specification for the new version applied to the old kubelet, the RHEL machine cannot return to the **Ready** state. You cannot complete the update until the machines are available. However, the maximum number of unavailable nodes is set to ensure that normal cluster operations can continue with that number of machines out of service.

The OpenShift Update Service is composed of an Operator and one or more application instances.



NOTE

During the upgrade process, nodes in the cluster might become temporarily unavailable. The **MachineHealthCheck** might identify such nodes as unhealthy and reboot them. To avoid rebooting such nodes, remove any **MachineHealthCheck** resource that you have deployed before updating the cluster. However, a MachineHealthCheck resource that is deployed by default (such as **machine-api-termination-handler**) cannot be removed and will be recreated.

2.3. SUPPORT POLICY FOR UNMANAGED OPERATORS

The *management state* of an Operator determines whether an Operator is actively managing the resources for its related component in the cluster as designed. If an Operator is set to an *unmanaged* state, it does not respond to changes in configuration nor does it receive updates.

While this can be helpful in non-production clusters or during debugging, Operators in an unmanaged state are unsupported and the cluster administrator assumes full control of the individual component configurations and upgrades.

An Operator can be set to an unmanaged state using the following methods:

- **Individual Operator configuration**
  Individual Operators have a **managementState** parameter in their configuration. This can be accessed in different ways, depending on the Operator. For example, the Red Hat OpenShift Logging Operator accomplishes this by modifying a custom resource (CR) that it manages, while the Cluster Samples Operator uses a cluster-wide configuration resource.

  Changing the **managementState** parameter to **Unmanaged** means that the Operator is not actively managing its resources and will take no action related to the related component. Some Operators might not support this management state as it might damage the cluster and require manual recovery.

WARNING

Changing individual Operators to the **Unmanaged** state renders that particular component and functionality unsupported. Reported issues must be reproduced in **Managed** state for support to proceed.

- **Cluster Version Operator (CVO) overrides**
  The **spec.overrides** parameter can be added to the CVO's configuration to allow administrators to provide a list of overrides to the CVO's behavior for a component. Setting the **spec.overrides[].unmanaged** parameter to **true** for a component blocks cluster upgrades and alerts the administrator after a CVO override has been set:

  > Disabling ownership via cluster version overrides prevents upgrades. Please remove overrides before continuing.

WARNING

Setting a CVO override puts the entire cluster in an unsupported state. Reported issues must be reproduced after removing any overrides for support to proceed.

2.4. NEXT STEPS

- Selecting a cluster installation method and preparing it for users

CHAPTER 3. THE OPENSHIFT CONTAINER PLATFORM CONTROL PLANE

3.1. UNDERSTANDING THE OPENSHIFT CONTAINER PLATFORM CONTROL PLANE

The control plane, which is composed of control plane machines (also known as the master machines), manages the OpenShift Container Platform cluster. The control plane machines manage workloads on the compute machines, which are also known as worker machines. The cluster itself manages all upgrades to the machines by the actions of the Cluster Version Operator, the Machine Config Operator, and a set of individual Operators.

3.1.1. Node configuration management with machine config pools

Machines that run control plane components or user workloads are divided into groups based on the types of resources they handle. These groups of machines are called machine config pools (MCP). Each MCP manages a set of nodes and its corresponding machine configs. The role of the node determines which MCP it belongs to; the MCP governs nodes based on its assigned node role label. Nodes in an MCP have the same configuration; this means nodes can be scaled up and torn down in response to increased or decreased workloads.

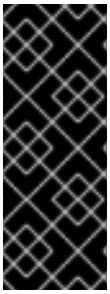
By default, there are two MCPs created by the cluster when it is installed: **master** and **worker**. Each default MCP has a defined configuration applied by the Machine Config Operator (MCO), which is responsible for managing MCPs and facilitating MCP upgrades. You can create additional MCPs, or custom pools, to manage nodes that have custom use cases that extend outside of the default node types.

Custom pools are pools that inherit their configurations from the worker pool. They use any machine config targeted for the worker pool, but add the ability to deploy changes only targeted at the custom pool. Since a custom pool inherits its configuration from the worker pool, any change to the worker pool is applied to the custom pool as well. Custom pools that do not inherit their configurations from the worker pool are not supported by the MCO.

NOTE

A node can only be included in one MCP. If a node has multiple labels that correspond to several MCPs, like **worker,infra**, it is managed by the infra custom pool, not the worker pool. Custom pools take priority on selecting nodes to manage based on node labels; nodes that do not belong to a custom pool are managed by the worker pool.

It is recommended to have a custom pool for every node role you want to manage in your cluster. For example, if you create infra nodes to handle infra workloads, it is recommended to create a custom infra MCP to group those nodes together. If you apply an **infra** role label to a worker node so it has the **worker,infra** dual label, but do not have a custom infra MCP, the MCO considers it a worker node. If you remove the **worker** label from a node and apply the **infra** label without grouping it in a custom pool, the node is not recognized by the MCO and is unmanaged by the cluster.



IMPORTANT

Any node labeled with the **infra** role that is only running infra workloads is not counted toward the total number of subscriptions. The MCP managing an infra node is mutually exclusive from how the cluster determines subscription charges; tagging a node with the appropriate **infra** role and using taints to prevent user workloads from being scheduled on that node are the only requirements for avoiding subscription charges for infra workloads.

The MCO applies updates for pools independently; for example, if there is an update that affects all pools, nodes from each pool update in parallel with each other. If you add a custom pool, nodes from that pool also attempt to update concurrently with the master and worker nodes.

3.1.2. Machine roles in OpenShift Container Platform

OpenShift Container Platform assigns hosts different roles. These roles define the function of the machine within the cluster. The cluster contains definitions for the standard master and worker role types.

NOTE

The cluster also contains the definition for the bootstrap role. Because the bootstrap machine is used only during cluster installation, its function is explained in the cluster installation documentation.

3.1.2.1. Cluster workers

In a Kubernetes cluster, the worker nodes are where the actual workloads requested by Kubernetes users run and are managed. The worker nodes advertise their capacity and the scheduler, which is part of the master services, determines on which nodes to start containers and pods. Important services run on each worker node, including CRI-O, which is the container engine, Kubelet, which is the service that accepts and fulfills requests for running and stopping container workloads, and a service proxy, which manages communication for pods across workers.

In OpenShift Container Platform, machine sets control the worker machines. Machines with the worker role drive compute workloads that are governed by a specific machine pool that autoscales them. Because OpenShift Container Platform has the capacity to support multiple machine types, the worker machines are classed as *compute* machines. In this release, the terms *worker machine* and *compute machine* are used interchangeably because the only default type of compute machine is the worker machine. In future versions of OpenShift Container Platform, different types of compute machines, such as infrastructure machines, might be used by default.

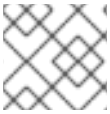
NOTE

Machine sets are groupings of machine resources under the **machine-api** namespace. Machine sets are configurations that are designed to start new machines on a specific cloud provider. Conversely, machine config pools (MCPs) are part of the Machine Config Operator (MCO) namespace. An MCP is used to group machines together so the MCO can manage their configurations and facilitate their upgrades.

3.1.2.2. Cluster masters

In a Kubernetes cluster, the control plane nodes (also known as the master nodes) run services that are required to control the Kubernetes cluster. In OpenShift Container Platform, the control plane machines

are the control plane. They contain more than just the Kubernetes services for managing the OpenShift Container Platform cluster. Because all of the machines with the control plane role are control plane machines, the terms *master* and *control plane* are used interchangeably to describe them. Instead of being grouped into a machine set, control plane machines are defined by a series of standalone machine API resources. Extra controls apply to control plane machines to prevent you from deleting all control plane machines and breaking your cluster.



NOTE

Exactly three control plane nodes must be used for all production deployments.

Services that fall under the Kubernetes category on the master include the Kubernetes API server, etcd, the Kubernetes controller manager, and the Kubernetes scheduler.

Table 3.1. Kubernetes services that run on the control plane

Component	Description
Kubernetes API server	The Kubernetes API server validates and configures the data for pods, services, and replication controllers. It also provides a focal point for the shared state of the cluster.
etcd	etcd stores the persistent master state while other components watch etcd for changes to bring themselves into the specified state.
Kubernetes controller manager	The Kubernetes controller manager watches etcd for changes to objects such as replication, namespace, and service account controller objects, and then uses the API to enforce the specified state. Several such processes create a cluster with one active leader at a time.
Kubernetes scheduler	The Kubernetes scheduler watches for newly created pods without an assigned node and selects the best node to host the pod.

There are also OpenShift services that run on the control plane, which include the OpenShift API server, OpenShift controller manager, OpenShift OAuth API server, and OpenShift OAuth server.

Table 3.2. OpenShift services that run on the control plane

Component	Description
OpenShift API server	The OpenShift API server validates and configures the data for OpenShift resources, such as projects, routes, and templates. The OpenShift API server is managed by the OpenShift API Server Operator.
OpenShift controller manager	The OpenShift controller manager watches etcd for changes to OpenShift objects, such as project, route, and template controller objects, and then uses the API to enforce the specified state. The OpenShift controller manager is managed by the OpenShift Controller Manager Operator.

Component	Description
OpenShift OAuth API server	The OpenShift OAuth API server validates and configures the data to authenticate to OpenShift Container Platform, such as users, groups, and OAuth tokens. The OpenShift OAuth API server is managed by the Cluster Authentication Operator.
OpenShift OAuth server	Users request tokens from the OpenShift OAuth server to authenticate themselves to the API. The OpenShift OAuth server is managed by the Cluster Authentication Operator.

Some of these services on the control plane machines run as systemd services, while others run as static pods.

Systemd services are appropriate for services that you need to always come up on that particular system shortly after it starts. For control plane machines, those include sshd, which allows remote login. It also includes services such as:

- The CRI-O container engine (crio), which runs and manages the containers. OpenShift Container Platform 4.8 uses CRI-O instead of the Docker Container Engine.
- Kubelet (kubelet), which accepts requests for managing containers on the machine from master services.

CRI-O and Kubelet must run directly on the host as systemd services because they need to be running before you can run other containers.

The **installer-*** and **revision-pruner-*** control plane pods must run with root permissions because they write to the **/etc/kubernetes** directory, which is owned by the root user. These pods are in the following namespaces:

- **openshift-etcd**
- **openshift-kube-apiserver**
- **openshift-kube-controller-manager**
- **openshift-kube-scheduler**

3.1.3. Operators in OpenShift Container Platform

In OpenShift Container Platform, Operators are the preferred method of packaging, deploying, and managing services on the control plane. They also provide advantages to applications that users run. Operators integrate with Kubernetes APIs and CLI tools such as **kubectl** and **oc** commands. They provide the means of watching over an application, performing health checks, managing over-the-air updates, and ensuring that the applications remain in your specified state.

Because CRI-O and the Kubelet run on every node, almost every other cluster function can be managed on the control plane by using Operators. Operators are among the most important components of OpenShift Container Platform 4.8. Components that are added to the control plane by using Operators

include critical networking and credential services.

The Operator that manages the other Operators in an OpenShift Container Platform cluster is the Cluster Version Operator.

OpenShift Container Platform 4.8 uses different classes of Operators to perform cluster operations and run services on the cluster for your applications to use.

3.1.3.1. Platform Operators in OpenShift Container Platform

In OpenShift Container Platform 4.8, all cluster functions are divided into a series of platform Operators. Platform Operators manage a particular area of cluster functionality, such as cluster-wide application logging, management of the Kubernetes control plane, or the machine provisioning system.

Each Operator provides you with a simple API for determining cluster functionality. The Operator hides the details of managing the lifecycle of that component. Operators can manage a single component or tens of components, but the end goal is always to reduce operational burden by automating common actions. Operators also offer a more granular configuration experience. You configure each component by modifying the API that the Operator exposes instead of modifying a global configuration file.

3.1.3.2. Operators managed by OLM

The Cluster Operator Lifecycle Management (OLM) component manages Operators that are available for use in applications. It does not manage the Operators that comprise OpenShift Container Platform. OLM is a framework that manages Kubernetes-native applications as Operators. Instead of managing Kubernetes manifests, it manages Kubernetes Operators. OLM manages two classes of Operators, Red Hat Operators and certified Operators.

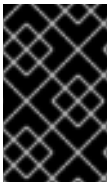
Some Red Hat Operators drive the cluster functions, like the scheduler and problem detectors. Others are provided for you to manage yourself and use in your applications, like etcd. OpenShift Container Platform also offers certified Operators, which the community built and maintains. These certified Operators provide an API layer to traditional applications so you can manage the application through Kubernetes constructs.

3.1.3.3. About the OpenShift Update Service

The OpenShift Update Service (OSUS) provides over-the-air updates to OpenShift Container Platform, including Red Hat Enterprise Linux CoreOS (RHCOS). It provides a graph, or diagram, that contains the *vertices* of component Operators and the *edges* that connect them. The edges in the graph show which versions you can safely update to. The vertices are update payloads that specify the intended state of the managed cluster components.

The Cluster Version Operator (CVO) in your cluster checks with the OpenShift Update Service to see the valid updates and update paths based on current component versions and information in the graph. When you request an update, the CVO uses the release image for that update to upgrade your cluster. The release artifacts are hosted in Quay as container images.

To allow the OpenShift Update Service to provide only compatible updates, a release verification pipeline drives automation. Each release artifact is verified for compatibility with supported cloud platforms and system architectures, as well as other component packages. After the pipeline confirms the suitability of a release, the OpenShift Update Service notifies you that it is available.



IMPORTANT

The OpenShift Update Service displays all recommended updates for your current cluster. If an upgrade path is not recommended by the OpenShift Update Service, it might be because of a known issue with the update or the target release.

Two controllers run during continuous update mode. The first controller continuously updates the payload manifests, applies the manifests to the cluster, and outputs the controlled rollout status of the Operators to indicate whether they are available, upgrading, or failed. The second controller polls the OpenShift Update Service to determine if updates are available.

IMPORTANT

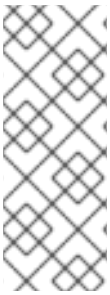
Only upgrading to a newer version is supported. Reverting or rolling back your cluster to a previous version is not supported. If your upgrade fails, contact Red Hat support.

During the upgrade process, the Machine Config Operator (MCO) applies the new configuration to your cluster machines. The MCO cordons the number of nodes as specified by the **maxUnavailable** field on the machine configuration pool and marks them as unavailable. By default, this value is set to **1**. The MCO then applies the new configuration and reboots the machine.

If you use Red Hat Enterprise Linux (RHEL) machines as workers, the MCO does not update the kubelet because you must update the OpenShift API on the machines first.

With the specification for the new version applied to the old kubelet, the RHEL machine cannot return to the **Ready** state. You cannot complete the update until the machines are available. However, the maximum number of unavailable nodes is set to ensure that normal cluster operations can continue with that number of machines out of service.

The OpenShift Update Service is composed of an Operator and one or more application instances.



NOTE

During the upgrade process, nodes in the cluster might become temporarily unavailable. The **MachineHealthCheck** might identify such nodes as unhealthy and reboot them. To avoid rebooting such nodes, remove any **MachineHealthCheck** resource that you have deployed before updating the cluster. However, a MachineHealthCheck resource that is deployed by default (such as **machine-api-termination-handler**) cannot be removed and will be recreated.

3.1.3.4. Understanding the Machine Config Operator

OpenShift Container Platform 4.8 integrates both operating system and cluster management. Because the cluster manages its own updates, including updates to Red Hat Enterprise Linux CoreOS (RHCOS) on cluster nodes, OpenShift Container Platform provides an opinionated lifecycle management experience that simplifies the orchestration of node upgrades.

OpenShift Container Platform employs three daemon sets and controllers to simplify node management. These daemon sets orchestrate operating system updates and configuration changes to the hosts by using standard Kubernetes-style constructs. They include:

- The **machine-config-controller**, which coordinates machine upgrades from the control plane. It monitors all of the cluster nodes and orchestrates their configuration updates.
- The **machine-config-daemon** daemon set, which runs on each node in the cluster and updates

a machine to configuration as defined by machine config and as instructed by the MachineConfigController. When the node detects a change, it drains off its pods, applies the update, and reboots. These changes come in the form of Ignition configuration files that apply the specified machine configuration and control kubelet configuration. The update itself is delivered in a container. This process is key to the success of managing OpenShift Container Platform and RHCOS updates together.

- The **machine-config-server** daemon set, which provides the Ignition config files to control plane nodes as they join the cluster.

The machine configuration is a subset of the Ignition configuration. The **machine-config-daemon** reads the machine configuration to see if it needs to do an OSTree update or if it must apply a series of systemd kubelet file changes, configuration changes, or other changes to the operating system or OpenShift Container Platform configuration.

When you perform node management operations, you create or modify a **KubeletConfig** custom resource (CR).

IMPORTANT

When changes are made to a machine configuration, the Machine Config Operator (MCO) automatically reboots all corresponding nodes in order for the changes to take effect.

To prevent the nodes from automatically rebooting after machine configuration changes, before making the changes, you must pause the autoreboot process by setting the **spec.paused** field to **true** in the corresponding machine config pool. When paused, machine configuration changes are not applied until you set the **spec.paused** field to **false** and the nodes have rebooted into the new configuration.

The following modifications do not trigger a node reboot:

- When the MCO detects any of the following changes, it applies the update without draining or rebooting the node:
  - Changes to the SSH key in the **spec.config.passwd.users.sshAuthorizedKeys** parameter of a machine config.
  - Changes to the global pull secret or pull secret in the **openshift-config** namespace.
  - Automatic rotation of the **/etc/kubernetes/kubelet-ca.crt** certificate authority (CA) by the Kubernetes API Server Operator.
- When the MCO detects changes to the **/etc/containers/registries.conf** file, such as adding or editing an **ImageContentSourcePolicy** object, it drains the corresponding nodes, applies the changes, and uncordons the nodes.

Additional information

For information on preventing the control plane machines from rebooting after the Machine Config Operator makes changes to the machine config, see Disabling Machine Config Operator from automatically rebooting.

CHAPTER 4. UNDERSTANDING OPENSHIFT CONTAINER PLATFORM DEVELOPMENT

To fully leverage the capability of containers when developing and running enterprise-quality applications, ensure your environment is supported by tools that allow containers to be:

- Created as discrete microservices that can be connected to other containerized, and non-containerized, services. For example, you might want to join your application with a database or attach a monitoring application to it.
- Resilient, so if a server crashes or needs to go down for maintenance or to be decommissioned, containers can start on another machine.
- Automated to pick up code changes automatically and then start and deploy new versions of themselves.
- Scaled up, or replicated, to have more instances serving clients as demand increases and then spun down to fewer instances as demand declines.
- Run in different ways, depending on the type of application. For example, one application might run once a month to produce a report and then exit. Another application might need to run constantly and be highly available to clients.
- Managed so you can watch the state of your application and react when something goes wrong.

Containers' widespread acceptance, and the resulting requirements for tools and methods to make them enterprise-ready, resulted in many options for them.

The rest of this section explains options for assets you can create when you build and deploy containerized Kubernetes applications in OpenShift Container Platform. It also describes which approaches you might use for different kinds of applications and development requirements.

4.1. ABOUT DEVELOPING CONTAINERIZED APPLICATIONS

You can approach application development with containers in many ways, and different approaches might be more appropriate for different situations. To illustrate some of this variety, the series of approaches that is presented starts with developing a single container and ultimately deploys that container as a mission-critical application for a large enterprise. These approaches show different tools, formats, and methods that you can employ with containerized application development. This topic describes:

- Building a simple container and storing it in a registry
- Creating a Kubernetes manifest and saving it to a Git repository
- Making an Operator to share your application with others

4.2. BUILDING A SIMPLE CONTAINER

You have an idea for an application and you want to containerize it.

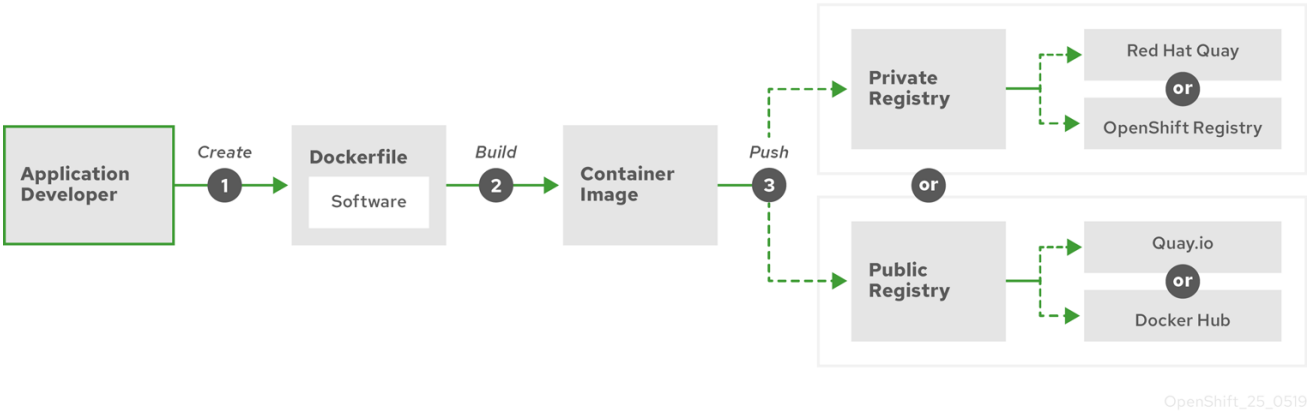
First you require a tool for building a container, like buildah or docker, and a file that describes what goes in your container, which is typically a Dockerfile.

Next, you require a location to push the resulting container image so you can pull it to run anywhere you want it to run. This location is a container registry.

Some examples of each of these components are installed by default on most Linux operating systems, except for the Dockerfile, which you provide yourself.

The following diagram displays the process of building and pushing an image:

Figure 4.1. Create a simple containerized application and push it to a registry



If you use a computer that runs Red Hat Enterprise Linux (RHEL) as the operating system, the process of creating a containerized application requires the following steps:

1. Install container build tools: RHEL contains a set of tools that includes podman, buildah, and skopeo that you use to build and manage containers.

2. Create a Dockerfile to combine base image and software: Information about building your container goes into a file that is named **Dockerfile**. In that file, you identify the base image you build from, the software packages you install, and the software you copy into the container. You also identify parameter values like network ports that you expose outside the container and volumes that you mount inside the container. Put your Dockerfile and the software you want to containerize in a directory on your RHEL system.

3. Run buildah or docker build: Run the **buildah build-using-dockerfile** or the **docker build** command to pull your chosen base image to the local system and create a container image that is stored locally. You can also build container images without a Dockerfile by using buildah.

4. Tag and push to a registry: Add a tag to your new container image that identifies the location of the registry in which you want to store and share your container. Then push that image to the registry by running the **podman push** or **docker push** command.

5. Pull and run the image: From any system that has a container client tool, such as podman or docker, run a command that identifies your new image. For example, run the **podman run <image_name>** or **docker run <image_name>** command. Here **<image_name>** is the name of your new container image, which resembles **quay.io/myrepo/myapp:latest**. The registry might require credentials to push and pull images.

For more details on the process of building container images, pushing them to registries, and running them, see Custom image builds with Buildah .

4.2.1. Container build tool options

Building and managing containers with buildah, podman, and skopeo results in industry standard container images that include features specifically tuned for deploying containers in OpenShift Container Platform or other Kubernetes environments. These tools are daemonless and can run without root privileges, requiring less overhead to run them.

IMPORTANT

Support for Docker Container Engine as a container runtime is deprecated in Kubernetes 1.20 and will be removed in a future release. However, Docker-produced images will continue to work in your cluster with all runtimes, including CRI-O. For more information, see the Kubernetes blog announcement .

When you ultimately run your containers in OpenShift Container Platform, you use the CRI-O container engine. CRI-O runs on every worker and control plane machine (also known as the master machine) in an OpenShift Container Platform cluster, but CRI-O is not yet supported as a standalone runtime outside of OpenShift Container Platform.

4.2.2. Base image options

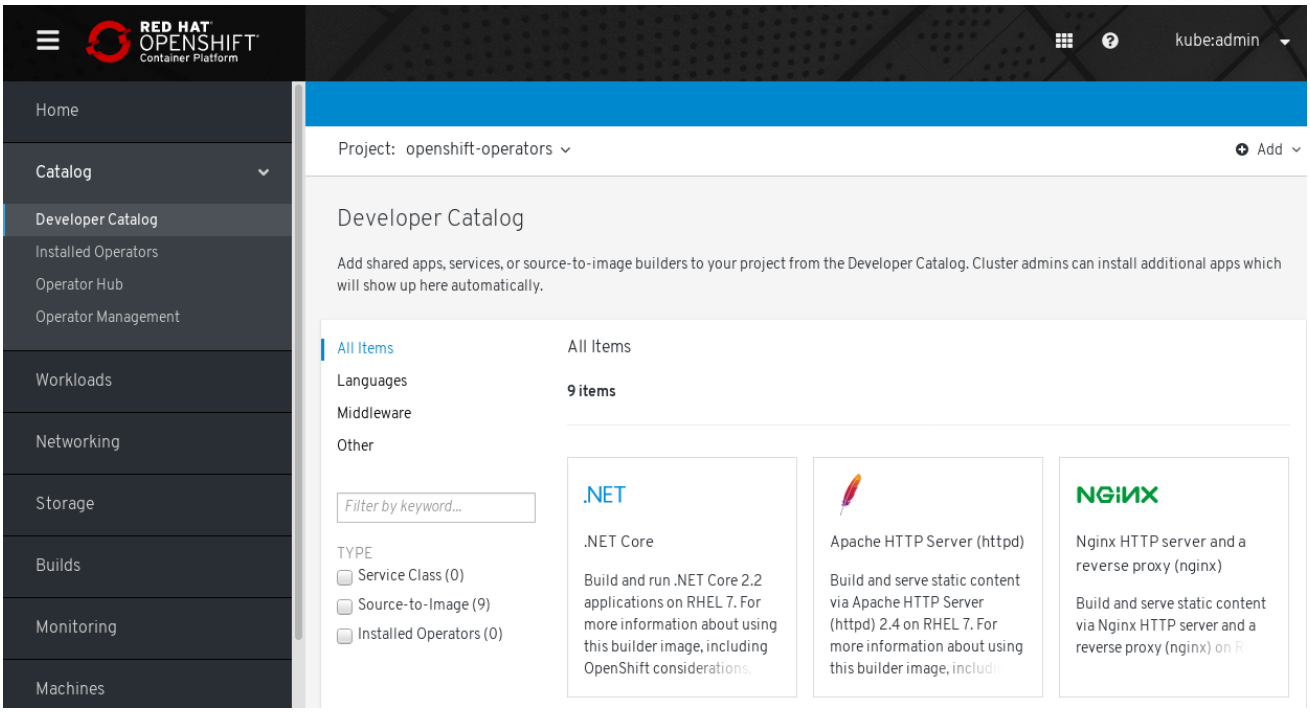
The base image you choose to build your application on contains a set of software that resembles a Linux system to your application. When you build your own image, your software is placed into that file system and sees that file system as though it were looking at its operating system. Choosing this base image has major impact on how secure, efficient and upgradeable your container is in the future.

Red Hat provides a new set of base images referred to as Red Hat Universal Base Images (UBI). These images are based on Red Hat Enterprise Linux and are similar to base images that Red Hat has offered in the past, with one major difference: they are freely redistributable without a Red Hat subscription. As a result, you can build your application on UBI images without having to worry about how they are shared or the need to create different images for different environments.

These UBI images have standard, init, and minimal versions. You can also use the Red Hat Software Collections images as a foundation for applications that rely on specific runtime environments such as Node.js, Perl, or Python. Special versions of some of these runtime base images are referred to as Source-to-Image (S2I) images. With S2I images, you can insert your code into a base image environment that is ready to run that code.

S2I images are available for you to use directly from the OpenShift Container Platform web UI by selecting **Catalog → Developer Catalog**, as shown in the following figure:

Figure 4.2. Choose S2I base images for apps that need specific runtimes



4.2.3. Registry options

Container registries are where you store container images so you can share them with others and make them available to the platform where they ultimately run. You can select large, public container registries that offer free accounts or a premium version that offer more storage and special features. You can also install your own registry that can be exclusive to your organization or selectively shared with others.

To get Red Hat images and certified partner images, you can draw from the Red Hat Registry. The Red Hat Registry is represented by two locations: **registry.access.redhat.com**, which is unauthenticated and deprecated, and **registry.redhat.io**, which requires authentication. You can learn about the Red Hat and partner images in the Red Hat Registry from the Container images section of the Red Hat Ecosystem Catalog. Besides listing Red Hat container images, it also shows extensive information about the contents and quality of those images, including health scores that are based on applied security updates.

Large, public registries include Docker Hub and Quay.io. The Quay.io registry is owned and managed by Red Hat. Many of the components used in OpenShift Container Platform are stored in Quay.io, including container images and the Operators that are used to deploy OpenShift Container Platform itself. Quay.io also offers the means of storing other types of content, including Helm charts.

If you want your own, private container registry, OpenShift Container Platform itself includes a private container registry that is installed with OpenShift Container Platform and runs on its cluster. Red Hat also offers a private version of the Quay.io registry called Red Hat Quay. Red Hat Quay includes geo replication, Git build triggers, Clair image scanning, and many other features.

All of the registries mentioned here can require credentials to download images from those registries. Some of those credentials are presented on a cluster-wide basis from OpenShift Container Platform, while other credentials can be assigned to individuals.

4.3. CREATING A KUBERNETES MANIFEST FOR OPENSHIFT CONTAINER PLATFORM

While the container image is the basic building block for a containerized application, more information is required to manage and deploy that application in a Kubernetes environment such as OpenShift Container Platform. The typical next steps after you create an image are to:

- Understand the different resources you work with in Kubernetes manifests
- Make some decisions about what kind of an application you are running
- Gather supporting components
- Create a manifest and store that manifest in a Git repository so you can store it in a source versioning system, audit it, track it, promote and deploy it to the next environment, roll it back to earlier versions, if necessary, and share it with others

4.3.1. About Kubernetes pods and services

While the container image is the basic unit with docker, the basic units that Kubernetes works with are called pods. Pods represent the next step in building out an application. A pod can contain one or more than one container. The key is that the pod is the single unit that you deploy, scale, and manage.

Scalability and namespaces are probably the main items to consider when determining what goes in a pod. For ease of deployment, you might want to deploy a container in a pod and include its own logging and monitoring container in the pod. Later, when you run the pod and need to scale up an additional instance, those other containers are scaled up with it. For namespaces, containers in a pod share the same network interfaces, shared storage volumes, and resource limitations, such as memory and CPU, which makes it easier to manage the contents of the pod as a single unit. Containers in a pod can also communicate with each other by using standard inter-process communications, such as System V semaphores or POSIX shared memory.

While individual pods represent a scalable unit in Kubernetes, a service provides a means of grouping together a set of pods to create a complete, stable application that can complete tasks such as load balancing. A service is also more permanent than a pod because the service remains available from the same IP address until you delete it. When the service is in use, it is requested by name and the OpenShift Container Platform cluster resolves that name into the IP addresses and ports where you can reach the pods that compose the service.

By their nature, containerized applications are separated from the operating systems where they run and, by extension, their users. Part of your Kubernetes manifest describes how to expose the application to internal and external networks by defining network policies that allow fine-grained control over communication with your containerized applications. To connect incoming requests for HTTP, HTTPS, and other services from outside your cluster to services inside your cluster, you can use an **Ingress** resource.

If your container requires on-disk storage instead of database storage, which might be provided through a service, you can add volumes to your manifests to make that storage available to your pods. You can configure the manifests to create persistent volumes (PVs) or dynamically create volumes that are added to your **Pod** definitions.

After you define a group of pods that compose your application, you can define those pods in **Deployment** and **DeploymentConfig** objects.

4.3.2. Application types

Next, consider how your application type influences how to run it.

Kubernetes defines different types of workloads that are appropriate for different kinds of applications. To determine the appropriate workload for your application, consider if the application is:

- Meant to run to completion and be done. An example is an application that starts up to produce a report and exits when the report is complete. The application might not run again then for a month. Suitable OpenShift Container Platform objects for these types of applications include **Job** and **CronJob** objects.

- Expected to run continuously. For long-running applications, you can write a deployment.

- Required to be highly available. If your application requires high availability, then you want to size your deployment to have more than one instance. A **Deployment** or **DeploymentConfig** object can incorporate a replica set for that type of application. With replica sets, pods run across multiple nodes to make sure the application is always available, even if a worker goes down.

- Need to run on every node. Some types of Kubernetes applications are intended to run in the cluster itself on every master or worker node. DNS and monitoring applications are examples of applications that need to run continuously on every node. You can run this type of application as a daemon set. You can also run a daemon set on a subset of nodes, based on node labels.

- Require life-cycle management. When you want to hand off your application so that others can use it, consider creating an Operator. Operators let you build in intelligence, so it can handle things like backups and upgrades automatically. Coupled with the Operator Lifecycle Manager (OLM), cluster managers can expose Operators to selected namespaces so that users in the cluster can run them.

- Have identity or numbering requirements. An application might have identity requirements or numbering requirements. For example, you might be required to run exactly three instances of the application and to name the instances **0**, **1**, and **2**. A stateful set is suitable for this application. Stateful sets are most useful for applications that require independent storage, such as databases and zookeeper clusters.

4.3.3. Available supporting components

The application you write might need supporting components, like a database or a logging component. To fulfill that need, you might be able to obtain the required component from the following Catalogs that are available in the OpenShift Container Platform web console:

- OperatorHub, which is available in each OpenShift Container Platform 4.8 cluster. The OperatorHub makes Operators available from Red Hat, certified Red Hat partners, and community members to the cluster operator. The cluster operator can make those Operators available in all or selected namespaces in the cluster, so developers can launch them and configure them with their applications.

- Templates, which are useful for a one-off type of application, where the lifecycle of a component is not important after it is installed. A template provides an easy way to get started developing a Kubernetes application with minimal overhead. A template can be a list of resource definitions, which could be **Deployment**, **Service**, **Route**, or other objects. If you want to change names or resources, you can set these values as parameters in the template.

You can configure the supporting Operators and templates to the specific needs of your development team and then make them available in the namespaces in which your developers work. Many people add shared templates to the **openshift** namespace because it is accessible from all other namespaces.

4.3.4. Applying the manifest

Kubernetes manifests let you create a more complete picture of the components that make up your Kubernetes applications. You write these manifests as YAML files and deploy them by applying them to the cluster, for example, by running the **oc apply** command.

4.3.5. Next steps

At this point, consider ways to automate your container development process. Ideally, you have some sort of CI pipeline that builds the images and pushes them to a registry. In particular, a GitOps pipeline integrates your container development with the Git repositories that you use to store the software that is required to build your applications.

The workflow to this point might look like:

- Day 1: You write some YAML. You then run the **oc apply** command to apply that YAML to the cluster and test that it works.
- Day 2: You put your YAML container configuration file into your own Git repository. From there, people who want to install that app, or help you improve it, can pull down the YAML and apply it to their cluster to run the app.
- Day 3: Consider writing an Operator for your application.

4.4. DEVELOP FOR OPERATORS

Packaging and deploying your application as an Operator might be preferred if you make your application available for others to run. As noted earlier, Operators add a lifecycle component to your application that acknowledges that the job of running an application is not complete as soon as it is installed.

When you create an application as an Operator, you can build in your own knowledge of how to run and maintain the application. You can build in features for upgrading the application, backing it up, scaling it, or keeping track of its state. If you configure the application correctly, maintenance tasks, like updating the Operator, can happen automatically and invisibly to the Operator's users.

An example of a useful Operator is one that is set up to automatically back up data at particular times. Having an Operator manage an application's backup at set times can save a system administrator from remembering to do it.

Any application maintenance that has traditionally been completed manually, like backing up data or rotating certificates, can be completed automatically with an Operator.

CHAPTER 5. RED HAT ENTERPRISE LINUX COREOS (RHCOS)

5.1. ABOUT RHCOS

Red Hat Enterprise Linux CoreOS (RHCOS) represents the next generation of single-purpose container operating system technology by providing the quality standards of Red Hat Enterprise Linux (RHEL) with automated, remote upgrade features.

RHCOS is supported only as a component of OpenShift Container Platform 4.8 for all OpenShift Container Platform machines. RHCOS is the only supported operating system for OpenShift Container Platform control plane, or master, machines. While RHCOS is the default operating system for all cluster machines, you can create compute machines, which are also known as worker machines, that use RHEL as their operating system. There are two general ways RHCOS is deployed in OpenShift Container Platform 4.8:

- If you install your cluster on infrastructure that the cluster provisions, RHCOS images are downloaded to the target platform during installation, and suitable Ignition config files, which control the RHCOS configuration, are used to deploy the machines.

- If you install your cluster on infrastructure that you manage, you must follow the installation documentation to obtain the RHCOS images, generate Ignition config files, and use the Ignition config files to provision your machines.

5.1.1. Key RHCOS features

The following list describes key features of the RHCOS operating system:

- **Based on RHEL**: The underlying operating system consists primarily of RHEL components. The same quality, security, and control measures that support RHEL also support RHCOS. For example, RHCOS software is in RPM packages, and each RHCOS system starts up with a RHEL kernel and a set of services that are managed by the systemd init system.

- **Controlled immutability**: Although it contains RHEL components, RHCOS is designed to be managed more tightly than a default RHEL installation. Management is performed remotely from the OpenShift Container Platform cluster. When you set up your RHCOS machines, you can modify only a few system settings. This controlled immutability allows OpenShift Container Platform to store the latest state of RHCOS systems in the cluster so it is always able to create additional machines and perform updates based on the latest RHCOS configurations.

- **CRI-O container runtime**: Although RHCOS contains features for running the OCI- and libcontainer-formatted containers that Docker requires, it incorporates the CRI-O container engine instead of the Docker container engine. By focusing on features needed by Kubernetes platforms, such as OpenShift Container Platform, CRI-O can offer specific compatibility with different Kubernetes versions. CRI-O also offers a smaller footprint and reduced attack surface than is possible with container engines that offer a larger feature set. At the moment, CRI-O is the only engine available within OpenShift Container Platform clusters.

- **Set of container tools**: For tasks such as building, copying, and otherwise managing containers, RHCOS replaces the Docker CLI tool with a compatible set of container tools. The podman CLI tool supports many container runtime features, such as running, starting, stopping, listing, and removing containers and container images. The skopeo CLI tool can copy, authenticate, and sign images. You can use the **crictl** CLI tool to work with containers and pods from the CRI-O container engine. While direct use of these tools in RHCOS is discouraged, you can use them for debugging purposes.

- **rpm-ostree upgrades**: RHCOS features transactional upgrades using the **rpm-ostree** system. Updates are delivered by means of container images and are part of the OpenShift Container Platform update process. When deployed, the container image is pulled, extracted, and written to disk, then the bootloader is modified to boot into the new version. The machine will reboot into the update in a rolling manner to ensure cluster capacity is minimally impacted.

- **bootupd firmware and bootloader updater**: Package managers and hybrid systems such as **rpm-ostree** do not update the firmware or the bootloader. With **bootupd**, RHCOS users have access to a cross-distribution, system-agnostic update tool that manages firmware and boot updates in UEFI and legacy BIOS boot modes that run on modern architectures, such as x86_64, ppc64le, and aarch64.
  For information about how to install **bootupd**, see the documentation for *Updating the bootloader using bootupd* for more information.

- **Updated through the Machine Config Operator**: In OpenShift Container Platform, the Machine Config Operator handles operating system upgrades. Instead of upgrading individual packages, as is done with **yum** upgrades, **rpm-ostree** delivers upgrades of the OS as an atomic unit. The new OS deployment is staged during upgrades and goes into effect on the next reboot. If something goes wrong with the upgrade, a single rollback and reboot returns the system to the previous state. RHCOS upgrades in OpenShift Container Platform are performed during cluster updates.

For RHCOS systems, the layout of the **rpm-ostree** file system has the following characteristics:

- /**usr** is where the operating system binaries and libraries are stored and is read-only. We do not support altering this.

- /**etc**, /**boot**, /**var** are writable on the system but only intended to be altered by the Machine Config Operator.

- /**var**/**lib**/**containers** is the graph storage location for storing container images.

5.1.2. Choosing how to configure RHCOS

RHCOS is designed to deploy on an OpenShift Container Platform cluster with a minimal amount of user configuration. In its most basic form, this consists of:

- Starting with a provisioned infrastructure, such as on AWS, or provisioning the infrastructure yourself.

- Supplying a few pieces of information, such as credentials and cluster name, in an **install-config.yaml** file when running **openshift-install**.

Because RHCOS systems in OpenShift Container Platform are designed to be fully managed from the OpenShift Container Platform cluster after that, directly changing an RHCOS machine is discouraged. Although limited direct access to RHCOS machines cluster can be accomplished for debugging purposes, you should not directly configure RHCOS systems. Instead, if you need to add or change features on your OpenShift Container Platform nodes, consider making changes in the following ways:

- **Kubernetes workload objects, such as DaemonSet and Deployment**: If you need to add services or other user-level features to your cluster, consider adding them as Kubernetes workload objects. Keeping those features outside of specific node configurations is the best way to reduce the risk of breaking the cluster on subsequent upgrades.

- **Day-2 customizations**: If possible, bring up a cluster without making any customizations to cluster nodes and make necessary node changes after the cluster is up. Those changes are

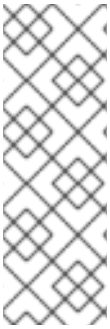
easier to track later and less likely to break updates. Creating machine configs or modifying Operator custom resources are ways of making these customizations.

- **Day-1 customizations**: For customizations that you must implement when the cluster first comes up, there are ways of modifying your cluster so changes are implemented on first boot. Day-1 customizations can be done through Ignition configs and manifest files during **openshift-install** or by adding boot options during ISO installs provisioned by the user.

Here are examples of customizations you could do on day 1:

- **Kernel arguments**: If particular kernel features or tuning is needed on nodes when the cluster first boots.
- **Disk encryption**: If your security needs require that the root file system on the nodes are encrypted, such as with FIPS support.
- **Kernel modules**: If a particular hardware device, such as a network card or video card, does not have a usable module available by default in the Linux kernel.
- **Chronyd**: If you want to provide specific clock settings to your nodes, such as the location of time servers.

To accomplish these tasks, you can augment the **openshift-install** process to include additional objects such as **MachineConfig** objects. Those procedures that result in creating machine configs can be passed to the Machine Config Operator after the cluster is up.



NOTE

The Ignition config files that the installation program generates contain certificates that expire after 24 hours, which are then renewed at that time. If the cluster is shut down before renewing the certificates and the cluster is later restarted after the 24 hours have elapsed, the cluster automatically recovers the expired certificates. The exception is that you must manually approve the pending **node-bootstrapper** certificate signing requests (CSRs) to recover kubelet certificates. See the documentation for *Recovering from expired control plane certificates* for more information.

5.1.3. Choosing how to deploy RHCOS

Differences between RHCOS installations for OpenShift Container Platform are based on whether you are deploying on an infrastructure provisioned by the installer or by the user:

- **Installer-provisioned**: Some cloud environments offer pre-configured infrastructures that allow you to bring up an OpenShift Container Platform cluster with minimal configuration. For these types of installations, you can supply Ignition configs that place content on each node so it is there when the cluster first boots.
- **User-provisioned**: If you are provisioning your own infrastructure, you have more flexibility in how you add content to a RHCOS node. For example, you could add kernel arguments when you boot the RHCOS ISO installer to install each system. However, in most cases where configuration is required on the operating system itself, it is best to provide that configuration through an Ignition config.

The Ignition facility runs only when the RHCOS system is first set up. After that, Ignition configs can be supplied later using the machine config.

5.1.4. About Ignition

Ignition is the utility that is used by RHCOS to manipulate disks during initial configuration. It completes common disk tasks, including partitioning disks, formatting partitions, writing files, and configuring users. On first boot, Ignition reads its configuration from the installation media or the location that you specify and applies the configuration to the machines.

Whether you are installing your cluster or adding machines to it, Ignition always performs the initial configuration of the OpenShift Container Platform cluster machines. Most of the actual system setup happens on each machine itself. For each machine, Ignition takes the RHCOS image and boots the RHCOS kernel. Options on the kernel command line identify the type of deployment and the location of the Ignition-enabled initial RAM disk (initramfs).

5.1.4.1. How Ignition works

To create machines by using Ignition, you need Ignition config files. The OpenShift Container Platform installation program creates the Ignition config files that you need to deploy your cluster. These files are based on the information that you provide to the installation program directly or through an **install-config.yaml** file.

The way that Ignition configures machines is similar to how tools like cloud-init or Linux Anaconda kickstart configure systems, but with some important differences:

- Ignition runs from an initial RAM disk that is separate from the system you are installing to. Because of that, Ignition can repartition disks, set up file systems, and perform other changes to the machine's permanent file system. In contrast, cloud-init runs as part of a machine init system when the system boots, so making foundational changes to things like disk partitions cannot be done as easily. With cloud-init, it is also difficult to reconfigure the boot process while you are in the middle of the node boot process.

- Ignition is meant to initialize systems, not change existing systems. After a machine initializes and the kernel is running from the installed system, the Machine Config Operator from the OpenShift Container Platform cluster completes all future machine configuration.

- Instead of completing a defined set of actions, Ignition implements a declarative configuration. It checks that all partitions, files, services, and other items are in place before the new machine starts. It then makes the changes, like copying files to disk that are necessary for the new machine to meet the specified configuration.

- After Ignition finishes configuring a machine, the kernel keeps running but discards the initial RAM disk and pivots to the installed system on disk. All of the new system services and other features start without requiring a system reboot.

- Because Ignition confirms that all new machines meet the declared configuration, you cannot have a partially configured machine. If a machine setup fails, the initialization process does not finish, and Ignition does not start the new machine. Your cluster will never contain partially configured machines. If Ignition cannot complete, the machine is not added to the cluster. You must add a new machine instead. This behavior prevents the difficult case of debugging a machine when the results of a failed configuration task are not known until something that depended on it fails at a later date.

- If there is a problem with an Ignition config that causes the setup of a machine to fail, Ignition will not try to use the same config to set up another machine. For example, a failure could result from an Ignition config made up of a parent and child config that both want to create the same file. A failure in such a case would prevent that Ignition config from being used again to set up an other machines until the problem is resolved.

- If you have multiple Ignition config files, you get a union of that set of configs. Because Ignition

is declarative, conflicts between the configs could cause Ignition to fail to set up the machine. The order of information in those files does not matter. Ignition will sort and implement each setting in ways that make the most sense. For example, if a file needs a directory several levels deep, if another file needs a directory along that path, the later file is created first. Ignition sorts and creates all files, directories, and links by depth.

- Because Ignition can start with a completely empty hard disk, it can do something cloud-init cannot do: set up systems on bare metal from scratch using features such as PXE boot. In the bare metal case, the Ignition config is injected into the boot partition so that Ignition can find it and configure the system correctly.

5.1.4.2. The Ignition sequence

The Ignition process for an RHCOS machine in an OpenShift Container Platform cluster involves the following steps:

- The machine gets its Ignition config file. Control plane machines (also known as the master machines) get their Ignition config files from the bootstrap machine, and worker machines get Ignition config files from a master.
- Ignition creates disk partitions, file systems, directories, and links on the machine. It supports RAID arrays but does not support LVM volumes.
- Ignition mounts the root of the permanent file system to the **/sysroot** directory in the initramfs and starts working in that **/sysroot** directory.
- Ignition configures all defined file systems and sets them up to mount appropriately at runtime.
- Ignition runs **systemd** temporary files to populate required files in the **/var** directory.
- Ignition runs the Ignition config files to set up users, systemd unit files, and other configuration files.
- Ignition unmounts all components in the permanent system that were mounted in the initramfs.
- Ignition starts up the init process of the new machine, which in turn starts up all other services on the machine that run during system boot.

At the end of this process, the machine is ready to join the cluster and does not require a reboot.

5.2. VIEWING IGNITION CONFIGURATION FILES

To see the Ignition config file used to deploy the bootstrap machine, run the following command:

```
$ openshift-install create ignition-configs --dir $HOME/testconfig
```

After you answer a few questions, the **bootstrap.ign**, **master.ign**, and **worker.ign** files appear in the directory you entered.

To see the contents of the **bootstrap.ign** file, pipe it through the **jq** filter. Here's a snippet from that file:

```
$ cat $HOME/testconfig/bootstrap.ign | jq
{
  "ignition": {
    "version": "3.2.0"
```

```
  },
  "passwd": {
    "users": [
      {
        "name": "core",
        "sshAuthorizedKeys": [
          "ssh-rsa AAAAB3NzaC1yc...."
        ]
      }
    ]
  },
  "storage": {
    "files": [
      {
        "overwrite": false,
        "path": "/etc/motd",
        "user": {
          "name": "root"
        },
        "append": [
          {
            "source": "data:text/plain;charset=utf-
8;base64,VGhpcyBpcyB0aGUgYm9vdHN0cmFwIG5vZGU7IGl0IHdpbGwgYmUgZGVzdHJveWVkIHdo
ZW4gdGhlIG1hc3RlciBpcyBmdWxseSB1cC4KClRoZSBwcmltYXJ5IHNlcnZpY2VzIGFyZSByZWxlYXNlLWltYWdlLnNlcnZpY2UgZm9sbG93ZWQgYnkgYm9vdGt1YmUuc2VydmljZS4gVG8gd2F0Y2ggdGhlaXIgc3RhdHVzLCBydW4gZS5nLgoKICBqb3VybmFsY3RsIC1iIC1mIC11IHJlbGVhc2UtaW1hZ2Uuc2VydmljZSAtdSBib290a3ViZS5zZXJ2aWNlCg=="
          }
        ],
        "mode": 420
      },
...
```

To decode the contents of a file listed in the **bootstrap.ign** file, pipe the base64-encoded data string representing the contents of that file to the **base64 -d** command. Here's an example using the contents of the /**etc**/**motd** file added to the bootstrap machine from the output shown above:

```
$ echo
VGhpcyBpcyB0aGUgYm9vdHN0cmFwIG5vZGU7IGl0IHdpbGwgYmUgZGVzdHJveWVkIHdoZW4gdG
hlIG1hc3RlciBpcyBmdWxseSB1cC4KClRoZSBwcmltYXJ5IHNlcnZpY2VzIGFyZSByZWxlYXNlLWltYWdl
LnNlcnZpY2UgZm9sbG93ZWQgYnkgYm9vdGt1YmUuc2VydmljZS4gVG8gd2F0Y2ggdGhlaXIgc3Rhd
HVzLCBydW4gZS5nLgoKICBqb3VybmFsY3RsIC1iIC1mIC11IHJlbGVhc2UtaW1hZ2Uuc2VydmljZSAtd
SBib290a3ViZS5zZXJ2aWNlCg== | base64 --decode
```

Example output

```
This is the bootstrap node; it will be destroyed when the master is fully up.

The primary services are release-image.service followed by bootkube.service. To watch their status,
run e.g.

  journalctl -b -f -u release-image.service -u bootkube.service
```

Repeat those commands on the **master.ign** and **worker.ign** files to see the source of Ignition config files for each of those machine types. You should see a line like the following for the **worker.ign**, identifying how it gets its Ignition config from the bootstrap machine:

```
"source": "https://api.myign.develcluster.example.com:22623/config/worker",
```

Here are a few things you can learn from the **bootstrap.ign** file:

- Format: The format of the file is defined in the Ignition config spec. Files of the same format are used later by the MCO to merge changes into a machine's configuration.
- Contents: Because the bootstrap machine serves the Ignition configs for other machines, both master and worker machine Ignition config information is stored in the **bootstrap.ign**, along with the bootstrap machine's configuration.
- Size: The file is more than 1300 lines long, with path to various types of resources.
- The content of each file that will be copied to the machine is actually encoded into data URLs, which tends to make the content a bit clumsy to read. (Use the **jq** and **base64** commands shown previously to make the content more readable.)
- Configuration: The different sections of the Ignition config file are generally meant to contain files that are just dropped into a machine's file system, rather than commands to modify existing files. For example, instead of having a section on NFS that configures that service, you would just add an NFS configuration file, which would then be started by the init process when the system comes up.
- users: A user named **core** is created, with your SSH key assigned to that user. This allows you to log in to the cluster with that user name and your credentials.
- storage: The storage section identifies files that are added to each machine. A few notable files include **/root/.docker/config.json** (which provides credentials your cluster needs to pull from container image registries) and a bunch of manifest files in **/opt/openshift/manifests** that are used to configure your cluster.
- systemd: The **systemd** section holds content used to create **systemd** unit files. Those files are used to start up services at boot time, as well as manage those services on running systems.
- Primitives: Ignition also exposes low-level primitives that other tools can build on.

5.3. CHANGING IGNITION CONFIGS AFTER INSTALLATION

Machine config pools manage a cluster of nodes and their corresponding machine configs. Machine configs contain configuration information for a cluster. To list all machine config pools that are known:

```
$ oc get machineconfigpools
```

Example output

```
NAME   CONFIG                                UPDATED UPDATING DEGRADED
master master-1638c1aea398413bb918e76632f20799 False   False    False
worker worker-2feef4f8288936489a5a832ca8efe953 False   False    False
```

To list all machine configs:

```
$ oc get machineconfig
```

Example output

```
NAME                                               GENERATEDBYCONTROLLER   IGNITIONVERSION   CREATED
OSIMAGEURL

00-master                                          4.0.0-0.150.0.0-dirty   3.2.0             16m
00-master-ssh                                      4.0.0-0.150.0.0-dirty                     16m
00-worker                                          4.0.0-0.150.0.0-dirty   3.2.0             16m
00-worker-ssh                                      4.0.0-0.150.0.0-dirty                     16m
01-master-kubelet                                  4.0.0-0.150.0.0-dirty   3.2.0             16m
01-worker-kubelet                                  4.0.0-0.150.0.0-dirty   3.2.0             16m
master-1638c1aea398413bb918e76632f20799   4.0.0-0.150.0.0-dirty   3.2.0             16m
worker-2feef4f8288936489a5a832ca8efe953   4.0.0-0.150.0.0-dirty   3.2.0             16m
```

The Machine Config Operator acts somewhat differently than Ignition when it comes to applying these machine configs. The machine configs are read in order (from 00* to 99*). Labels inside the machine configs identify the type of node each is for (master or worker). If the same file appears in multiple machine config files, the last one wins. So, for example, any file that appears in a 99* file would replace the same file that appeared in a 00* file. The input **MachineConfig** objects are unioned into a "rendered" **MachineConfig** object, which will be used as a target by the operator and is the value you can see in the machine config pool.

To see what files are being managed from a machine config, look for "Path:" inside a particular **MachineConfig** object. For example:

```
$ oc describe machineconfigs 01-worker-container-runtime | grep Path:
```

Example output

```
      Path:         /etc/containers/registries.conf
      Path:         /etc/containers/storage.conf
      Path:         /etc/crio/crio.conf
```

Be sure to give the machine config file a later name (such as 10-worker-container-runtime). Keep in mind that the content of each file is in URL-style data. Then apply the new machine config to the cluster.

CHAPTER 6. ADMISSION PLUG-INS

6.1. ABOUT ADMISSION PLUG-INS

Admission plug-ins are used to help regulate how OpenShift Container Platform 4.8 functions. Admission plug-ins intercept requests to the master API to validate resource requests and ensure policies are adhered to, after the request is authenticated and authorized. For example, they are commonly used to enforce security policy, resource limitations or configuration requirements.

Admission plug-ins run in sequence as an admission chain. If any admission plug-in in the sequence rejects a request, the whole chain is aborted and an error is returned.

OpenShift Container Platform has a default set of admission plug-ins enabled for each resource type. These are required for proper functioning of the cluster. Admission plug-ins ignore resources that they are not responsible for.

In addition to the defaults, the admission chain can be extended dynamically through webhook admission plug-ins that call out to custom webhook servers. There are two types of webhook admission plug-ins: a mutating admission plug-in and a validating admission plug-in. The mutating admission plug-in runs first and can both modify resources and validate requests. The validating admission plug-in validates requests and runs after the mutating admission plug-in so that modifications triggered by the mutating admission plug-in can also be validated.

Calling webhook servers through a mutating admission plug-in can produce side effects on resources related to the target object. In such situations, you must take steps to validate that the end result is as expected.

WARNING

Dynamic admission should be used cautiously because it impacts cluster control plane operations. When calling webhook servers through webhook admission plug-ins in OpenShift Container Platform 4.8, ensure that you have read the documentation fully and tested for side effects of mutations. Include steps to restore resources back to their original state prior to mutation, in the event that a request does not pass through the entire admission chain.

6.2. DEFAULT ADMISSION PLUG-INS

Default validating and admission plug-ins are enabled in OpenShift Container Platform 4.8. These default plug-ins contribute to fundamental control plane functionality, such as ingress policy, cluster resource limit override and quota policy. The following lists contain the default admission plug-ins:

Example 6.1. Validating admission plug-ins

- **LimitRanger**
- **ServiceAccount**
- **PodNodeSelector**

- **Priority**
- **PodTolerationRestriction**
- **OwnerReferencesPermissionEnforcement**
- **PersistentVolumeClaimResize**
- **RuntimeClass**
- **CertificateApproval**
- **CertificateSigning**
- **CertificateSubjectRestriction**
- **autoscaling.openshift.io/ManagementCPUsOverride**
- **authorization.openshift.io/RestrictSubjectBindings**
- **scheduling.openshift.io/OriginPodNodeEnvironment**
- **network.openshift.io/ExternalIPRanger**
- **network.openshift.io/RestrictedEndpointsAdmission**
- **image.openshift.io/ImagePolicy**
- **security.openshift.io/SecurityContextConstraint**
- **security.openshift.io/SCCExecRestrictions**
- **route.openshift.io/IngressAdmission**
- **config.openshift.io/ValidateAPIServer**
- **config.openshift.io/ValidateAuthentication**
- **config.openshift.io/ValidateFeatureGate**
- **config.openshift.io/ValidateConsole**
- **operator.openshift.io/ValidateDNS**
- **config.openshift.io/ValidateImage**
- **config.openshift.io/ValidateOAuth**
- **config.openshift.io/ValidateProject**
- **config.openshift.io/DenyDeleteClusterConfiguration**
- **config.openshift.io/ValidateScheduler**
- **quota.openshift.io/ValidateClusterResourceQuota**
- **security.openshift.io/ValidateSecurityContextConstraints**

- **authorization.openshift.io/ValidateRoleBindingRestriction**
- **config.openshift.io/ValidateNetwork**
- **operator.openshift.io/ValidateKubeControllerManager**
- **ValidatingAdmissionWebhook**
- **ResourceQuota**
- **quota.openshift.io/ClusterResourceQuota**

Example 6.2. Mutating admission plug-ins

- **NamespaceLifecycle**
- **LimitRanger**
- **ServiceAccount**
- **NodeRestriction**
- **TaintNodesByCondition**
- **PodNodeSelector**
- **Priority**
- **DefaultTolerationSeconds**
- **PodTolerationRestriction**
- **PersistentVolumeLabel**
- **DefaultStorageClass**
- **StorageObjectInUseProtection**
- **RuntimeClass**
- **DefaultIngressClass**
- **autoscaling.openshift.io/ManagementCPUsOverride**
- **scheduling.openshift.io/OriginPodNodeEnvironment**
- **image.openshift.io/ImagePolicy**
- **security.openshift.io/SecurityContextConstraint**
- **security.openshift.io/DefaultSecurityContextConstraints**
- **MutatingAdmissionWebhook**

6.3. WEBHOOK ADMISSION PLUG-INS

In addition to OpenShift Container Platform default admission plug-ins, dynamic admission can be implemented through webhook admission plug-ins that call webhook servers, to extend the functionality of the admission chain. Webhook servers are called over HTTP at defined endpoints.

There are two types of webhook admission plug-ins in OpenShift Container Platform:

- During the admission process, the *mutating admission plug-in* can perform tasks, such as injecting affinity labels.
- At the end of the admission process, the *validating admission plug-in* can be used to make sure an object is configured properly, for example ensuring affinity labels are as expected. If the validation passes, OpenShift Container Platform schedules the object as configured.

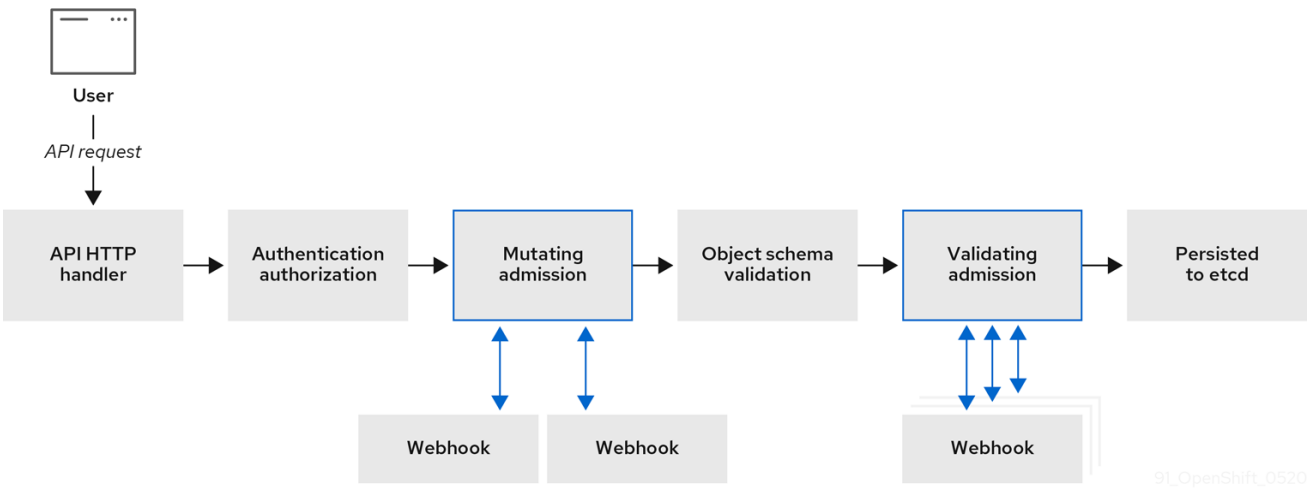
When an API request comes in, mutating or validating admission plug-ins use the list of external webhooks in the configuration and call them in parallel:

- If all of the webhooks approve the request, the admission chain continues.
- If any of the webhooks deny the request, the admission request is denied and the reason for doing so is based on the first denial.
- If more than one webhook denies the admission request, only the first denial reason is returned to the user.
- If an error is encountered when calling a webhook, the request is either denied or the webhook is ignored depending on the error policy set. If the error policy is set to **Ignore**, the request is unconditionally accepted in the event of a failure. If the policy is set to **Fail**, failed requests are denied. Using **Ignore** can result in unpredictable behavior for all clients.

Communication between the webhook admission plug-in and the webhook server must use TLS. Generate a CA certificate and use the certificate to sign the server certificate that is used by your webhook admission server. The PEM-encoded CA certificate is supplied to the webhook admission plug-in using a mechanism, such as service serving certificate secrets.

The following diagram illustrates the sequential admission chain process within which multiple webhook servers are called.

Figure 6.1. API admission chain with mutating and validating admission plug-ins



An example webhook admission plug-in use case is where all pods must have a common set of labels. In this example, the mutating admission plug-in can inject labels and the validating admission plug-in can check that labels are as expected. OpenShift Container Platform would subsequently schedule pods that include required labels and reject those that do not.

Some common webhook admission plug-in use cases include:

- Namespace reservation.
- Limiting custom network resources managed by the SR-IOV network device plug-in.
- Defining tolerations that enable taints to qualify which pods should be scheduled on a node.
- Pod priority class validation.

6.4. TYPES OF WEBHOOK ADMISSION PLUG-INS

Cluster administrators can call out to webhook servers through the mutating admission plug-in or the validating admission plug-in in the API server admission chain.

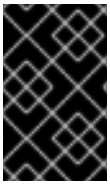
6.4.1. Mutating admission plug-in

The mutating admission plug-in is invoked during the mutation phase of the admission process, which allows modification of resource content before it is persisted. One example webhook that can be called through the mutating admission plug-in is the Pod Node Selector feature, which uses an annotation on a namespace to find a label selector and add it to the pod specification.

Sample mutating admission plug-in configuration

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingWebhookConfiguration (1)
metadata:
  name: <webhook_name> (2)
webhooks:
- name: <webhook_name> (3)
  clientConfig: (4)
    service:
      namespace: default (5)
      name: kubernetes (6)
      path: <webhook_url> (7)
    caBundle: <ca_signing_certificate> (8)
  rules: (9)
  - operations: (10)
    - <operation>
    apiGroups:
    - ""
    apiVersions:
    - "*"
    resources:
    - <resource>
  failurePolicy: <policy> (11)
  sideEffects: None
```

1. Specifies a mutating admission plug-in configuration.
2. The name for the **MutatingWebhookConfiguration** object. Replace **<webhook_name>** with the appropriate value.
3. The name of the webhook to call. Replace **<webhook_name>** with the appropriate value.
4. Information about how to connect to, trust, and send data to the webhook server.
5. The namespace where the front-end service is created.
6. The name of the front-end service.
7. The webhook URL used for admission requests. Replace **<webhook_url>** with the appropriate value.
8. A PEM-encoded CA certificate that signs the server certificate that is used by the webhook server. Replace **<ca_signing_certificate>** with the appropriate certificate in base64 format.
9. Rules that define when the API server should use this webhook admission plug-in.
10. One or more operations that trigger the API server to call this webhook admission plug-in. Possible values are **create**, **update**, **delete** or **connect**. Replace **<operation>** and **<resource>** with the appropriate values.
11. Specifies how the policy should proceed if the webhook server is unavailable. Replace **<policy>** with either **Ignore** (to unconditionally accept the request in the event of a failure) or **Fail** (to deny the failed request). Using **Ignore** can result in unpredictable behavior for all clients.



IMPORTANT

In OpenShift Container Platform 4.8, objects created by users or control loops through a mutating admission plug-in might return unexpected results, especially if values set in an initial request are overwritten, which is not recommended.

6.4.2. Validating admission plug-in

A validating admission plug-in is invoked during the validation phase of the admission process. This phase allows the enforcement of invariants on particular API resources to ensure that the resource does not change again. The Pod Node Selector is also an example of a webhook which is called by the validating admission plug-in, to ensure that all **nodeSelector** fields are constrained by the node selector restrictions on the namespace.

Sample validating admission plug-in configuration

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: ValidatingWebhookConfiguration (1)
metadata:
  name: <webhook_name> (2)
webhooks:
- name: <webhook_name> (3)
  clientConfig: (4)
    service:
      namespace: default (5)
```

```
    name: kubernetes 6
    path: <webhook_url> 7
  caBundle: <ca_signing_certificate> 8
rules: 9
- operations: 10
  - <operation>
  apiGroups:
  - ""
  apiVersions:
  - "*"
  resources:
  - <resource>
failurePolicy: <policy> 11
sideEffects: Unknown
```

1. Specifies a validating admission plug-in configuration.
2. The name for the **ValidatingWebhookConfiguration** object. Replace **<webhook_name>** with the appropriate value.
3. The name of the webhook to call. Replace **<webhook_name>** with the appropriate value.
4. Information about how to connect to, trust, and send data to the webhook server.
5. The namespace where the front-end service is created.
6. The name of the front-end service.
7. The webhook URL used for admission requests. Replace **<webhook_url>** with the appropriate value.
8. A PEM-encoded CA certificate that signs the server certificate that is used by the webhook server. Replace **<ca_signing_certificate>** with the appropriate certificate in base64 format.
9. Rules that define when the API server should use this webhook admission plug-in.
10. One or more operations that trigger the API server to call this webhook admission plug-in. Possible values are **create**, **update**, **delete** or **connect**. Replace **<operation>** and **<resource>** with the appropriate values.
11. Specifies how the policy should proceed if the webhook server is unavailable. Replace **<policy>** with either **Ignore** (to unconditionally accept the request in the event of a failure) or **Fail** (to deny the failed request). Using **Ignore** can result in unpredictable behavior for all clients.

6.5. CONFIGURING DYNAMIC ADMISSION

This procedure outlines high-level steps to configure dynamic admission. The functionality of the admission chain is extended by configuring a webhook admission plug-in to call out to a webhook server.

The webhook server is also configured as an aggregated API server. This allows other OpenShift Container Platform components to communicate with the webhook using internal credentials and facilitates testing using the **oc** command. Additionally, this enables role based access control (RBAC) into the webhook and prevents token information from other API servers from being disclosed to the webhook.

Prerequisites

- An OpenShift Container Platform account with cluster administrator access.
- The OpenShift Container Platform CLI (**oc**) installed.
- A published webhook server container image.

Procedure

1. Build a webhook server container image and make it available to the cluster using an image registry.
2. Create a local CA key and certificate and use them to sign the webhook server's certificate signing request (CSR).
3. Create a new project for webhook resources:

```
$ oc new-project my-webhook-namespace 1
```

1 Note that the webhook server might expect a specific name.

4. Define RBAC rules for the aggregated API service in a file called **rbac.yaml**:

```
apiVersion: v1
kind: List
items:

- apiVersion: rbac.authorization.k8s.io/v1 1
  kind: ClusterRoleBinding
  metadata:
    name: auth-delegator-my-webhook-namespace
  roleRef:
    kind: ClusterRole
    apiGroup: rbac.authorization.k8s.io
    name: system:auth-delegator
  subjects:
  - kind: ServiceAccount
    namespace: my-webhook-namespace
    name: server

- apiVersion: rbac.authorization.k8s.io/v1 2
  kind: ClusterRole
  metadata:
    annotations:
    name: system:openshift:online:my-webhook-server
  rules:
  - apiGroups:
    - online.openshift.io
    resources:
    - namespacereservations 3
    verbs:
    - get
    - list
    - watch
```

```
- apiVersion: rbac.authorization.k8s.io/v1 4
  kind: ClusterRole
  metadata:
    name: system:openshift:online:my-webhook-requester
  rules:
  - apiGroups:
    - admission.online.openshift.io
    resources:
    - namespacereservations 5
    verbs:
    - create

- apiVersion: rbac.authorization.k8s.io/v1 6
  kind: ClusterRoleBinding
  metadata:
    name: my-webhook-server-my-webhook-namespace
  roleRef:
    kind: ClusterRole
    apiGroup: rbac.authorization.k8s.io
    name: system:openshift:online:my-webhook-server
  subjects:
  - kind: ServiceAccount
    namespace: my-webhook-namespace
    name: server

- apiVersion: rbac.authorization.k8s.io/v1 7
  kind: RoleBinding
  metadata:
    namespace: kube-system
    name: extension-server-authentication-reader-my-webhook-namespace
  roleRef:
    kind: Role
    apiGroup: rbac.authorization.k8s.io
    name: extension-apiserver-authentication-reader
  subjects:
  - kind: ServiceAccount
    namespace: my-webhook-namespace
    name: server

- apiVersion: rbac.authorization.k8s.io/v1 8
  kind: ClusterRole
  metadata:
    name: my-cluster-role
  rules:
  - apiGroups:
    - admissionregistration.k8s.io
    resources:
    - validatingwebhookconfigurations
    - mutatingwebhookconfigurations
    verbs:
    - get
    - list
    - watch
  - apiGroups:
```

```
    - ""
    resources:
    - namespaces
    verbs:
    - get
    - list
    - watch

- apiVersion: rbac.authorization.k8s.io/v1
  kind: ClusterRoleBinding
  metadata:
    name: my-cluster-role
  roleRef:
    kind: ClusterRole
    apiGroup: rbac.authorization.k8s.io
    name: my-cluster-role
  subjects:
  - kind: ServiceAccount
    namespace: my-webhook-namespace
    name: server
```

1. Delegates authentication and authorization to the webhook server API.
2. Allows the webhook server to access cluster resources.
3. Points to resources. This example points to the **namespacereservations** resource.
4. Enables the aggregated API server to create admission reviews.
5. Points to resources. This example points to the **namespacereservations** resource.
6. Enables the webhook server to access cluster resources.
7. Role binding to read the configuration for terminating authentication.
8. Default cluster role and cluster role bindings for an aggregated API server.

5. Apply those RBAC rules to the cluster:

```
$ oc auth reconcile -f rbac.yaml
```

6. Create a YAML file called **webhook-daemonset.yaml** that is used to deploy a webhook as a daemon set server in a namespace:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  namespace: my-webhook-namespace
  name: server
  labels:
    server: "true"
spec:
  selector:
    matchLabels:
      server: "true"
```

```
  template:
    metadata:
      name: server
      labels:
        server: "true"
    spec:
      serviceAccountName: server
      containers:
      - name: my-webhook-container 1
        image: <image_registry_username>/<image_path>:<tag> 2
        imagePullPolicy: IfNotPresent
        command:
        - <container_commands> 3
        ports:
        - containerPort: 8443 4
        volumeMounts:
        - mountPath: /var/serving-cert
          name: serving-cert
        readinessProbe:
          httpGet:
            path: /healthz
            port: 8443 5
            scheme: HTTPS
      volumes:
      - name: serving-cert
        secret:
          defaultMode: 420
          secretName: server-serving-cert
```

1. Note that the webhook server might expect a specific container name.
2. Points to a webhook server container image. Replace **<image_registry_username>/<image_path>:<tag>** with the appropriate value.
3. Specifies webhook container run commands. Replace **<container_commands>** with the appropriate value.
4. Defines the target port within pods. This example uses port 8443.
5. Specifies the port used by the readiness probe. This example uses port 8443.

7. Deploy the daemon set:

```
$ oc apply -f webhook-daemonset.yaml
```

8. Define a secret for the service serving certificate signer, within a YAML file called **webhook-secret.yaml**:

```
apiVersion: v1
kind: Secret
metadata:
  namespace: my-webhook-namespace
  name: server-serving-cert
type: kubernetes.io/tls
```

```
data:
  tls.crt: <server_certificate> 1
  tls.key: <server_key> 2
```

1. References the signed webhook server certificate. Replace **<server_certificate>** with the appropriate certificate in base64 format.
2. References the signed webhook server key. Replace **<server_key>** with the appropriate key in base64 format.

9. Create the secret:

```
$ oc apply -f webhook-secret.yaml
```

10. Define a service account and service, within a YAML file called **webhook-service.yaml**:

```
apiVersion: v1
kind: List
items:

- apiVersion: v1
  kind: ServiceAccount
  metadata:
    namespace: my-webhook-namespace
    name: server

- apiVersion: v1
  kind: Service
  metadata:
    namespace: my-webhook-namespace
    name: server
    annotations:
      service.alpha.openshift.io/serving-cert-secret-name: server-serving-cert
  spec:
    selector:
      server: "true"
    ports:
    - port: 443 1
      targetPort: 8443 2
```

1. Defines the port that the service listens on. This example uses port 443.
2. Defines the target port within pods that the service forwards connections to. This example uses port 8443.

11. Expose the webhook server within the cluster:

```
$ oc apply -f webhook-service.yaml
```

12. Define a custom resource definition for the webhook server, in a file called **webhook-crd.yaml**:

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
```

```
metadata:
  name: namespacereservations.online.openshift.io (1)
spec:
  group: online.openshift.io (2)
  version: v1alpha1 (3)
  scope: Cluster (4)
  names:
    plural: namespacereservations (5)
    singular: namespacereservation (6)
    kind: NamespaceReservation (7)
```

1. Reflects **CustomResourceDefinition spec** values and is in the format **<plural>.<group>**. This example uses the **namespacereservations** resource.
2. REST API group name.
3. REST API version name.
4. Accepted values are **Namespaced** or **Cluster**.
5. Plural name to be included in URL.
6. Alias seen in **oc** output.
7. The reference for resource manifests.

13. Apply the custom resource definition:

```
$ oc apply -f webhook-crd.yaml
```

14. Configure the webhook server also as an aggregated API server, within a file called **webhook-api-service.yaml**:

```
apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  name: v1beta1.admission.online.openshift.io
spec:
  caBundle: <ca_signing_certificate> (1)
  group: admission.online.openshift.io
  groupPriorityMinimum: 1000
  versionPriority: 15
  service:
    name: server
    namespace: my-webhook-namespace
  version: v1beta1
```

1. A PEM-encoded CA certificate that signs the server certificate that is used by the webhook server. Replace **<ca_signing_certificate>** with the appropriate certificate in base64 format.

15. Deploy the aggregated API service:

```
$ oc apply -f webhook-api-service.yaml
```

16. Define the webhook admission plug-in configuration within a file called **webhook-config.yaml**. This example uses the validating admission plug-in:

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: ValidatingWebhookConfiguration
metadata:
 name: namespacereservations.admission.online.openshift.io (1)
webhooks:
- name: namespacereservations.admission.online.openshift.io (2)
 clientConfig:
  service: (3)
   namespace: default
   name: kubernetes
   path: /apis/admission.online.openshift.io/v1beta1/namespacereservations (4)
  caBundle: <ca_signing_certificate> (5)
 rules:
 - operations:
  - CREATE
  apiGroups:
  - project.openshift.io
  apiVersions:
  - "*"
  resources:
  - projectrequests
 - operations:
  - CREATE
  apiGroups:
  - ""
  apiVersions:
  - "*"
  resources:
  - namespaces
 failurePolicy: Fail
```

(1) Name for the **ValidatingWebhookConfiguration** object. This example uses the **namespacereservations** resource.

(2) Name of the webhook to call. This example uses the **namespacereservations** resource.

(3) Enables access to the webhook server through the aggregated API.

(4) The webhook URL used for admission requests. This example uses the **namespacereservation** resource.

(5) A PEM-encoded CA certificate that signs the server certificate that is used by the webhook server. Replace **<ca_signing_certificate>** with the appropriate certificate in base64 format.

17. Deploy the webhook:

```
$ oc apply -f webhook-config.yaml
```

18. Verify that the webhook is functioning as expected. For example, if you have configured dynamic admission to reserve specific namespaces, confirm that requests to create those namespaces are rejected and that requests to create non-reserved namespaces succeed.

6.6. ADDITIONAL RESOURCES

- Limiting custom network resources managed by the SR-IOV network device plug-in
- Defining tolerations that enable taints to qualify which pods should be scheduled on a node
- Pod priority class validation