OpenShift Dedicated 4

Architecture

Architecture overview.

OpenShift Dedicated 4 Architecture

Architecture overview.

Legal Notice

Abstract

OpenShift Dedicated is a cloud-based Kubernetes container platform. The foundation of OpenShift Dedicated is based on Kubernetes and therefore shares the same technology. To learn more about OpenShift Dedicated and Kubernetes, see product architecture.

Table of Contents

CHAPTER 1. ARCHITECTURE OVERVIEW

OpenShift Dedicated is a cloud-based Kubernetes container platform. The foundation of OpenShift Dedicated is based on Kubernetes and therefore shares the same technology. To learn more about OpenShift Dedicated and Kubernetes, see product architecture.

1.1. GLOSSARY OF COMMON TERMS FOR OPENSHIFT DEDICATED ARCHITECTURE

This glossary defines common terms that are used in the architecture content.

access policies

A set of roles that dictate how users, applications, and entities within a cluster interact with one another. An access policy increases cluster security.

admission plugins

Admission plugins enforce security policies, resource limitations, or configuration requirements.

authentication

To control access to an OpenShift Dedicated cluster, an administrator with the **dedicated-admin** role can configure user authentication to ensure only approved users access the cluster. To interact with an OpenShift Dedicated cluster, you must authenticate with the OpenShift Dedicated API. You can authenticate by providing an OAuth access token or an X.509 client certificate in your requests to the OpenShift Dedicated API.

bootstrap

A temporary machine that runs minimal Kubernetes and deploys the OpenShift Dedicated control plane.

certificate signing requests (CSRs)

A resource requests a denoted signer to sign a certificate. This request might get approved or denied.

Cluster Version Operator (CVO)

An Operator that checks with the OpenShift Dedicated Update Service to see the valid updates and update paths based on current component versions and information in the graph.

compute nodes

Nodes that are responsible for executing workloads for cluster users. Compute nodes are also known as worker nodes.

configuration drift

A situation where the configuration on a node does not match what the machine config specifies.

containers

Lightweight and executable images that consist of software and all of its dependencies. Because containers virtualize the operating system, you can run containers anywhere, such as data centers, public or private clouds, and local hosts.

container orchestration engine

Software that automates the deployment, management, scaling, and networking of containers.

container workloads

Applications that are packaged and deployed in containers.

control groups (cgroups)

Partitions sets of processes into groups to manage and limit the resources processes consume.

control plane

A container orchestration layer that exposes the API and interfaces to define, deploy, and manage the life cycle of containers. Control planes are also known as control plane machines.

CRI-O

A Kubernetes native container runtime implementation that integrates with the operating system to deliver an efficient Kubernetes experience.

deployment

A Kubernetes resource object that maintains the life cycle of an application.

Dockerfile

A text file that contains the user commands to perform on a terminal to assemble the image.

hybrid cloud deployments

Deployments that deliver a consistent platform across bare metal, virtual, private, and public cloud environments. This offers speed, agility, and portability.

Ignition

A utility that RHCOS uses to manipulate disks during initial configuration. It completes common disk tasks, including partitioning disks, formatting partitions, writing files, and configuring users.

installer-provisioned infrastructure

The installation program deploys and configures the infrastructure that the cluster runs on.

kubelet

A primary node agent that runs on each node in the cluster to ensure that containers are running in a pod.

kubernetes manifest

Specifications of a Kubernetes API object in a JSON or YAML format. A configuration file can include deployments, config maps, secrets, daemon sets.

Machine Config Daemon (MCD)

A daemon that regularly checks the nodes for configuration drift.

Machine Config Operator (MCO)

An Operator that applies the new configuration to your cluster machines.

machine config pools (MCP)

A group of machines, such as control plane components or user workloads, that are based on the resources that they handle.

metadata

Additional information about cluster deployment artifacts.

microservices

An approach to writing software. Applications can be separated into the smallest components, independent from each other by using microservices.

mirror registry

A registry that holds the mirror of OpenShift Dedicated images.

monolithic applications

Applications that are self-contained, built, and packaged as a single piece.

namespaces

A namespace isolates specific system resources that are visible to all processes. Inside a namespace, only processes that are members of that namespace can see those resources.

networking

Network information of OpenShift Dedicated cluster.

node

A worker machine in the OpenShift Dedicated cluster. A node is either a virtual machine (VM) or a physical machine.

OpenShift CLI (oc)

A command line tool to run OpenShift Dedicated commands on the terminal.

OpenShift Update Service (OSUS)

For clusters with internet access, Red Hat Enterprise Linux (RHEL) provides over-the-air updates by using an OpenShift update service as a hosted service located behind public APIs.

OpenShift image registry

A registry provided by OpenShift Dedicated to manage images.

Operator

The preferred method of packaging, deploying, and managing a Kubernetes application in an OpenShift Dedicated cluster. An Operator takes human operational knowledge and encodes it into software that is packaged and shared with customers.

OperatorHub

A platform that contains various OpenShift Dedicated Operators to install.

Operator Lifecycle Manager (OLM)

OLM helps you to install, update, and manage the lifecycle of Kubernetes native applications. OLM is an open source toolkit designed to manage Operators in an effective, automated, and scalable way.

OSTree

An upgrade system for Linux-based operating systems that performs atomic upgrades of complete file system trees. OSTree tracks meaningful changes to the file system tree using an addressable object store, and is designed to complement existing package management systems.

over-the-air (OTA) updates

The OpenShift Dedicated Update Service (OSUS) provides over-the-air updates to OpenShift Dedicated, including Red Hat Enterprise Linux CoreOS (RHCOS).

pod

One or more containers with shared resources, such as volume and IP addresses, running in your OpenShift Dedicated cluster. A pod is the smallest compute unit defined, deployed, and managed.

private registry

OpenShift Dedicated can use any server implementing the container image registry API as a source of the image which allows the developers to push and pull their private container images.

public registry

OpenShift Dedicated can use any server implementing the container image registry API as a source of the image which allows the developers to push and pull their public container images.

RHEL OpenShift Dedicated Cluster Manager

A managed service where you can install, modify, operate, and upgrade your OpenShift Dedicated clusters.

RHEL Quay Container Registry

A Quay.io container registry that serves most of the container images and Operators to OpenShift Dedicated clusters.

replication controllers

An asset that indicates how many pod replicas are required to run at a time.

role-based access control (RBAC)

A key security control to ensure that cluster users and workloads have only access to resources required to execute their roles.

route

Routes expose a service to allow for network access to pods from users and applications outside the OpenShift Dedicated instance.

scaling

The increasing or decreasing of resource capacity.

service

A service exposes a running application on a set of pods.

Source-to-Image (S2I) image

An image created based on the programming language of the application source code in OpenShift Dedicated to deploy applications.

storage

OpenShift Dedicated supports many types of storage for cloud providers. You can manage container storage for persistent and non-persistent data in an OpenShift Dedicated cluster.

Telemetry

A component to collect information such as size, health, and status of OpenShift Dedicated.

template

A template describes a set of objects that can be parameterized and processed to produce a list of objects for creation by OpenShift Dedicated.

web console

A user interface (UI) to manage OpenShift Dedicated.

worker node

Nodes that are responsible for executing workloads for cluster users. Worker nodes are also known as compute nodes.

Additional resources

- For more information on storage, see OpenShift Dedicated storage.

- For more information on authentication, see OpenShift Dedicated authentication.

- For more information on Operator Lifecycle Manager (OLM), see OLM.

- For more information on logging, see About Logging.

1.2. UNDERSTANDING HOW OPENSHIFT DEDICATED DIFFERS FROM OPENSHIFT CONTAINER PLATFORM

OpenShift Dedicated uses the same code base as OpenShift Container Platform but is installed in an opinionated way to be optimized for performance, scalability, and security. OpenShift Dedicated is a fully managed service; therefore, many of the OpenShift Dedicated components and settings that you manually set up in OpenShift Container Platform are set up for you by default.

Review the following differences between OpenShift Dedicated and a standard installation of OpenShift Container Platform on your own infrastructure:

OpenShift Container Platform	OpenShift Dedicated
The customer installs and configures OpenShift Container Platform.	OpenShift Dedicated is installed through Red Hat OpenShift Cluster Manager and in a standardized way that is optimized for performance, scalability, and security.
Customers can choose their computing resources.	OpenShift Dedicated is hosted and managed in a public cloud (Amazon Web Services or Google Cloud Platform) either owned by Red Hat or provided by the customer.
Customers have top-level administrative access to the infrastructure.	Customers have a built-in administrator group (**dedicated-admin**), though the top-level administration access is available when cloud accounts are provided by the customer.
Customers can use all supported features and configuration settings available in OpenShift Container Platform.	Some OpenShift Container Platform features and configuration settings might not be available or changeable in OpenShift Dedicated.
You set up control plane components such as the API server and etcd on machines that get the **control** role. You can modify the control plane components, but are responsible for backing up, restoring, and making control plane data highly available.	Red Hat sets up the control plane and manages the control plane components for you. The control plane is highly available.
You are responsible for updating the underlying infrastructure for the control plane and worker nodes. You can use the OpenShift web console to update OpenShift Container Platform versions.	Red Hat automatically notifies the customer when updates are available. You can manually or automatically schedule updates in OpenShift Cluster Manager.
Support is provided based on the terms of your Red Hat subscription or cloud provider.	Engineered, operated, and supported by Red Hat with a 99.95% uptime SLA and 24x7 coverage. For details, see Red Hat Enterprise Agreement Appendix 4 (Online Subscription Services).

1.3. ABOUT THE CONTROL PLANE

The control plane manages the worker nodes and the pods in your cluster. You can configure nodes with the use of machine config pools (MCPs). MCPs are groups of machines, such as control plane components or user workloads, that are based on the resources that they handle. OpenShift Dedicated assigns different roles to hosts. These roles define the function of a machine in a cluster. The cluster contains definitions for the standard control plane and worker role types.

You can use Operators to package, deploy, and manage services on the control plane. Operators are important components in OpenShift Dedicated because they provide the following services:

- Perform health checks

- Provide ways to watch applications

- Manage over-the-air updates

- Ensure applications stay in the specified state

1.4. ABOUT CONTAINERIZED APPLICATIONS FOR DEVELOPERS

As a developer, you can use different tools, methods, and formats to develop your containerized application based on your unique requirements, for example:

- Use various build-tool, base-image, and registry options to build a simple container application.

- Use supporting components such as OperatorHub and templates to develop your application.

- Package and deploy your application as an Operator.

You can also create a Kubernetes manifest and store it in a Git repository. Kubernetes works on basic units called pods. A pod is a single instance of a running process in your cluster. Pods can contain one or more containers. You can create a service by grouping a set of pods and their access policies. Services provide permanent internal IP addresses and host names for other applications to use as pods are created and destroyed. Kubernetes defines workloads based on the type of your application.

1.5. ABOUT ADMISSION PLUGINS

You can use admission plugins to regulate how OpenShift Dedicated functions. After a resource request is authenticated and authorized, admission plugins intercept the resource request to the master API to validate resource requests and to ensure that scaling policies are adhered to. Admission plugins are used to enforce security policies, resource limitations, configuration requirements, and other settings.

CHAPTER 2. OPENSHIFT DEDICATED ARCHITECTURE

2.1. INTRODUCTION TO OPENSHIFT DEDICATED

OpenShift Dedicated 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 Dedicated 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.

Red Hat OpenShift

Control plane nodes (xN)

OpenShift Services

Networking components

Kubernetes Services

Scheduler

Controller manager

API server

Cluster Version Operator

etcd

Compute nodes (xN)

Observability

Logging

Kibana

Elastisearch

Grafana Loki

Monitoring

Prometheus

Grafana

Alertmanager

Networking

SDN

NTO

DNS

Router

OpenShift Lifecycle Manager

Integrated Image Registry

Machine management

Kubelet

Compute | Network | Storage

Developers — SCM (Git) → CI/CD

Admins — Existing Automation Toolsets

596_OpenShift_0324

2.1.1. 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.

- 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.

2.1.2. 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.

2.1.2.1. 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.

2.1.2.2. 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 Dedicated 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.

2.1.3. OpenShift Dedicated overview

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

- Integrated Red Hat technology. Major components in OpenShift Dedicated come from Red Hat Enterprise Linux (RHEL) and related Red Hat technologies. OpenShift Dedicated 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 Dedicated 4 offers. The following sections describe some unique features and benefits of OpenShift Dedicated.

2.1.3.1. Custom operating system

OpenShift Dedicated uses Red Hat Enterprise Linux CoreOS (RHCOS) as the operating system for all control plane and worker nodes.

RHCOS includes:

- Ignition, which OpenShift Dedicated 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 Dedicated 3.

- Kubelet, the primary node agent for Kubernetes that is responsible for launching and monitoring containers.

2.1.3.2. Simplified update process

Updating, or upgrading, OpenShift Dedicated is a simple, highly-automated process. Because OpenShift Dedicated 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.

2.1.3.3. Other key features

Operators are both the fundamental unit of the OpenShift Dedicated 4 code base and a convenient way to deploy applications and software components for your applications to use. In OpenShift Dedicated, Operators serve as the platform foundation and remove the need for manual upgrades of operating systems and control plane applications. OpenShift Dedicated 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 Dedicated 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 Dedicated include improvements in software defined networking (SDN), authentication, log aggregation, monitoring, and routing. OpenShift Dedicated also offers a comprehensive web console and the custom OpenShift CLI (**oc**) interface.

CHAPTER 3. CONTROL PLANE ARCHITECTURE

The *control plane*, which is composed of control plane machines, manages the OpenShift Dedicated 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 (CVO), and a set of individual Operators.

3.1. MACHINE ROLES IN OPENSHIFT DEDICATED

OpenShift Dedicated 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.

3.1.1. Cluster workers

In a Kubernetes cluster, worker nodes run and manage the actual workloads requested by Kubernetes users. The worker nodes advertise their capacity and the scheduler, which is a control plane service, determines on which nodes to start pods and containers. The following important services run on each worker node:

- CRI-O, which is the container engine.

- kubelet, which is the service that accepts and fulfills requests for running and stopping container workloads.

- A service proxy, which manages communication for pods across workers.

- The runC or crun low-level container runtime, which creates and runs containers.

> **NOTE**
>
> For information about how to enable crun instead of the default runC, see the documentation for creating a **ContainerRuntimeConfig** CR.

In OpenShift Dedicated, compute machine sets control the compute machines, which are assigned the **worker** machine role. Machines with the **worker** role drive compute workloads that are governed by a specific machine pool that autoscales them. Because OpenShift Dedicated has the capacity to support multiple machine types, the machines with the **worker** role 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 Dedicated, different types of compute machines, such as infrastructure machines, might be used by default.

> **NOTE**
>
> Compute machine sets are groupings of compute machine resources under the **machine-api** namespace. Compute machine sets are configurations that are designed to start new compute 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. Cluster control planes

In a Kubernetes cluster, the *master* nodes run services that are required to control the Kubernetes cluster. In OpenShift Dedicated, the control plane is comprised of control plane machines that have a

master machine role. They contain more than just the Kubernetes services for managing the OpenShift Dedicated cluster.

For most OpenShift Dedicated clusters, control plane machines are defined by a series of standalone machine API resources. Control planes are managed with control plane machine sets. Extra controls apply to control plane machines to prevent you from deleting all of the control plane machines and breaking your cluster.

> **NOTE**
>
> Single availability zone clusters and multiple availability zone clusters require a minimum of three control plane nodes.

Services that fall under the Kubernetes category on the control plane 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 control plane 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.

Component	Description
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.
OpenShift OAuth API server	The OpenShift OAuth API server validates and configures the data to authenticate to OpenShift Dedicated, 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 Dedicated 4 uses CRI-O instead of the Docker Container Engine.

- Kubelet (kubelet), which accepts requests for managing containers on the machine from control plane 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**

3.2. OPERATORS IN OPENSHIFT DEDICATED

Operators are among the most important components of OpenShift Dedicated. Operators are the preferred method of packaging, deploying, and managing services on the control plane. They can 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 monitoring applications, performing health checks, managing over-the-air (OTA) updates, and ensuring that applications remain in your specified state.

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.

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. Components that are added to the control plane by using Operators include critical networking and credential services.

While both follow similar Operator concepts and goals, Operators in OpenShift Dedicated are managed by two different systems, depending on their purpose:

- Cluster Operators, which are managed by the Cluster Version Operator (CVO), are installed by default to perform cluster functions.

- Optional add-on Operators, which are managed by Operator Lifecycle Manager (OLM), can be made accessible for users to run in their applications.

3.2.1. Add-on Operators

Operator Lifecycle Manager (OLM) and OperatorHub are default components in OpenShift Dedicated that help manage Kubernetes-native applications as Operators. Together they provide the system for discovering, installing, and managing the optional add-on Operators available on the cluster.

Using OperatorHub in the OpenShift Dedicated web console, administrators with the **dedicated-admin** role and authorized users can select Operators to install from catalogs of Operators. After installing an Operator from OperatorHub, it can be made available globally or in specific namespaces to run in user applications.

Default catalog sources are available that include Red Hat Operators, certified Operators, and community Operators. Administrators with the **dedicated-admin** role can also add their own custom catalog sources, which can contain a custom set of Operators.
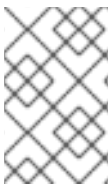
> **NOTE**
>
> All Operators listed in the Operator Hub marketplace should be available for installation. These Operators are considered customer workloads, and are not monitored by Red Hat Site Reliability Engineering (SRE).

Developers can use the Operator SDK to help author custom Operators that take advantage of OLM features, as well. Their Operator can then be bundled and added to a custom catalog source, which can be added to a cluster and made available to users.

> **NOTE**
>
> OLM does not manage the cluster Operators that comprise the OpenShift Dedicated architecture.

Additional resources

- For more details on running add-on Operators in OpenShift Dedicated, see the *Operators* guide sections on Operator Lifecycle Manager (OLM) and OperatorHub.

- For more details on the Operator SDK, see Developing Operators.

3.3. OVERVIEW OF ETCD

etcd is a consistent, distributed key-value store that holds small amounts of data that can fit entirely in memory. Although etcd is a core component of many projects, it is the primary data store for Kubernetes, which is the standard system for container orchestration.

3.3.1. Benefits of using etcd

By using etcd, you can benefit in several ways:

- Maintain consistent uptime for your cloud-native applications, and keep them working even if individual servers fail

- Store and replicate all cluster states for Kubernetes

- Distribute configuration data to provide redundancy and resiliency for the configuration of nodes

3.3.2. How etcd works

To ensure a reliable approach to cluster configuration and management, etcd uses the etcd Operator. The Operator simplifies the use of etcd on a Kubernetes container platform like OpenShift Dedicated. With the etcd Operator, you can create or delete etcd members, resize clusters, perform backups, and upgrade etcd.

The etcd Operator observes, analyzes, and acts:

1. It observes the cluster state by using the Kubernetes API.

2. It analyzes differences between the current state and the state that you want.

3. It fixes the differences through the etcd cluster management APIs, the Kubernetes API, or both.

etcd holds the cluster state, which is constantly updated. This state is continuously persisted, which leads to a high number of small changes at high frequency. As a result, Red Hat Site Reliability Engineering (SRE) backs the etcd cluster member with fast, low-latency I/O.

CHAPTER 4. NVIDIA GPU ARCHITECTURE OVERVIEW

NVIDIA supports the use of graphics processing unit (GPU) resources on OpenShift Dedicated. OpenShift Dedicated is a security-focused and hardened Kubernetes platform developed and supported by Red Hat for deploying and managing Kubernetes clusters at scale. OpenShift Dedicated includes enhancements to Kubernetes so that users can easily configure and use NVIDIA GPU resources to accelerate workloads.

The NVIDIA GPU Operator leverages the Operator framework within OpenShift Dedicated to manage the full lifecycle of NVIDIA software components required to run GPU-accelerated workloads.

These components include the NVIDIA drivers (to enable CUDA), the Kubernetes device plugin for GPUs, the NVIDIA Container Toolkit, automatic node tagging using GPU feature discovery (GFD), DCGM-based monitoring, and others.

> **NOTE**
>
> The NVIDIA GPU Operator is only supported by NVIDIA. For more information about obtaining support from NVIDIA, see Obtaining Support from NVIDIA .

4.1. NVIDIA GPU PREREQUISITES

- A working OpenShift cluster with at least one GPU worker node.

- Access to the OpenShift cluster as a **cluster-admin** to perform the required steps.

- OpenShift CLI (**oc**) is installed.

- The node feature discovery (NFD) Operator is installed and a **nodefeaturediscovery** instance is created.

4.2. GPUS AND OSD

You can deploy OpenShift Dedicated on NVIDIA GPU instance types.

It is important that this compute instance is a GPU-accelerated compute instance and that the GPU type matches the list of supported GPUs from NVIDIA AI Enterprise. For example, T4, V100, and A100 are part of this list.

You can choose one of the following methods to access the containerized GPUs:

- GPU passthrough to access and use GPU hardware within a virtual machine (VM).

- GPU (vGPU) time slicing when the entire GPU is not required.

Additional resources

- Red Hat Openshift in the Cloud

4.3. GPU SHARING METHODS

Red Hat and NVIDIA have developed GPU concurrency and sharing mechanisms to simplify GPU-accelerated computing on an enterprise-level OpenShift Dedicated cluster.

Applications typically have different compute requirements that can leave GPUs underutilized. Providing the right amount of compute resources for each workload is critical to reduce deployment cost and maximize GPU utilization.

Concurrency mechanisms for improving GPU utilization exist that range from programming model APIs to system software and hardware partitioning, including virtualization. The following list shows the GPU concurrency mechanisms:

- Compute Unified Device Architecture (CUDA) streams

- Time-slicing

- CUDA Multi-Process Service (MPS)

- Multi-instance GPU (MIG)

- Virtualization with vGPU

Additional resources

- Improving GPU Utilization

4.3.1. CUDA streams

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by NVIDIA for general computing on GPUs.

A stream is a sequence of operations that executes in issue-order on the GPU. CUDA commands are typically executed sequentially in a default stream and a task does not start until a preceding task has completed.

Asynchronous processing of operations across different streams allows for parallel execution of tasks. A task issued in one stream runs before, during, or after another task is issued into another stream. This allows the GPU to run multiple tasks simultaneously in no prescribed order, leading to improved performance.

Additional resources

- Asynchronous Concurrent Execution

4.3.2. Time-slicing

GPU time-slicing interleaves workloads scheduled on overloaded GPUs when you are running multiple CUDA applications.

You can enable time-slicing of GPUs on Kubernetes by defining a set of replicas for a GPU, each of which can be independently distributed to a pod to run workloads on. Unlike multi-instance GPU (MIG), there is no memory or fault isolation between replicas, but for some workloads this is better than not sharing at all. Internally, GPU time-slicing is used to multiplex workloads from replicas of the same underlying GPU.

You can apply a cluster-wide default configuration for time-slicing. You can also apply node-specific configurations. For example, you can apply a time-slicing configuration only to nodes with Tesla T4 GPUs and not modify nodes with other GPU models.

You can combine these two approaches by applying a cluster-wide default configuration and then labeling nodes to give those nodes a node-specific configuration.

4.3.3. CUDA Multi-Process Service

CUDA Multi-Process Service (MPS) allows a single GPU to use multiple CUDA processes. The processes run in parallel on the GPU, eliminating saturation of the GPU compute resources. MPS also enables concurrent execution, or overlapping, of kernel operations and memory copying from different processes to enhance utilization.

Additional resources

- CUDA MPS

4.3.4. Multi-instance GPU

Using Multi-instance GPU (MIG), you can split GPU compute units and memory into multiple MIG instances. Each of these instances represents a standalone GPU device from a system perspective and can be connected to any application, container, or virtual machine running on the node. The software that uses the GPU treats each of these MIG instances as an individual GPU.

MIG is useful when you have an application that does not require the full power of an entire GPU. The MIG feature of the new NVIDIA Ampere architecture enables you to split your hardware resources into multiple GPU instances, each of which is available to the operating system as an independent CUDA-enabled GPU.

NVIDIA GPU Operator version 1.7.0 and higher provides MIG support for the A100 and A30 Ampere cards. These GPU instances are designed to support up to seven multiple independent CUDA applications so that they operate completely isolated with dedicated hardware resources.

Additional resources

- NVIDIA Multi-Instance GPU User Guide

4.3.5. Virtualization with vGPU

Virtual machines (VMs) can directly access a single physical GPU using NVIDIA vGPU. You can create virtual GPUs that can be shared by VMs across the enterprise and accessed by other devices.

This capability combines the power of GPU performance with the management and security benefits provided by vGPU. Additional benefits provided by vGPU includes proactive management and monitoring for your VM environment, workload balancing for mixed VDI and compute workloads, and resource sharing across multiple VMs.

Additional resources

- Virtual GPUs

4.4. NVIDIA GPU FEATURES FOR OPENSHIFT DEDICATED

NVIDIA Container Toolkit

NVIDIA Container Toolkit enables you to create and run GPU-accelerated containers. The toolkit includes a container runtime library and utilities to automatically configure containers to use NVIDIA GPUs.

NVIDIA AI Enterprise

NVIDIA AI Enterprise is an end-to-end, cloud-native suite of AI and data analytics software optimized, certified, and supported with NVIDIA-Certified systems.

NVIDIA AI Enterprise includes support for Red Hat OpenShift Dedicated. The following installation methods are supported:

- OpenShift Dedicated on bare metal or VMware vSphere with GPU Passthrough.

- OpenShift Dedicated on VMware vSphere with NVIDIA vGPU.

GPU Feature Discovery

NVIDIA GPU Feature Discovery for Kubernetes is a software component that enables you to automatically generate labels for the GPUs available on a node. GPU Feature Discovery uses node feature discovery (NFD) to perform this labeling.

The Node Feature Discovery Operator (NFD) manages the discovery of hardware features and configurations in an OpenShift Container Platform cluster by labeling nodes with hardware-specific information. NFD labels the host with node-specific attributes, such as PCI cards, kernel, OS version, and so on.

You can find the NFD Operator in the Operator Hub by searching for "Node Feature Discovery".

NVIDIA GPU Operator with OpenShift Virtualization

Up until this point, the GPU Operator only provisioned worker nodes to run GPU-accelerated containers. Now, the GPU Operator can also be used to provision worker nodes for running GPU-accelerated virtual machines (VMs).

You can configure the GPU Operator to deploy different software components to worker nodes depending on which GPU workload is configured to run on those nodes.

GPU Monitoring dashboard

You can install a monitoring dashboard to display GPU usage information on the cluster **Observe** page in the OpenShift Dedicated web console. GPU utilization information includes the number of available GPUs, power consumption (in watts), temperature (in degrees Celsius), utilization (in percent), and other metrics for each GPU.

Additional resources

- NVIDIA-Certified Systems

- NVIDIA AI Enterprise

- NVIDIA Container Toolkit

- Enabling the GPU Monitoring Dashboard

- MIG Support in OpenShift Container Platform

- Time-slicing NVIDIA GPUs in OpenShift

- Deploy GPU Operators in a disconnected or airgapped environment

CHAPTER 5. UNDERSTANDING OPENSHIFT DEDICATED 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 Dedicated. It also describes which approaches you might use for different kinds of applications and development requirements.

5.1. 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

5.2. 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 5.1. Create a simple containerized application and push it to a registry
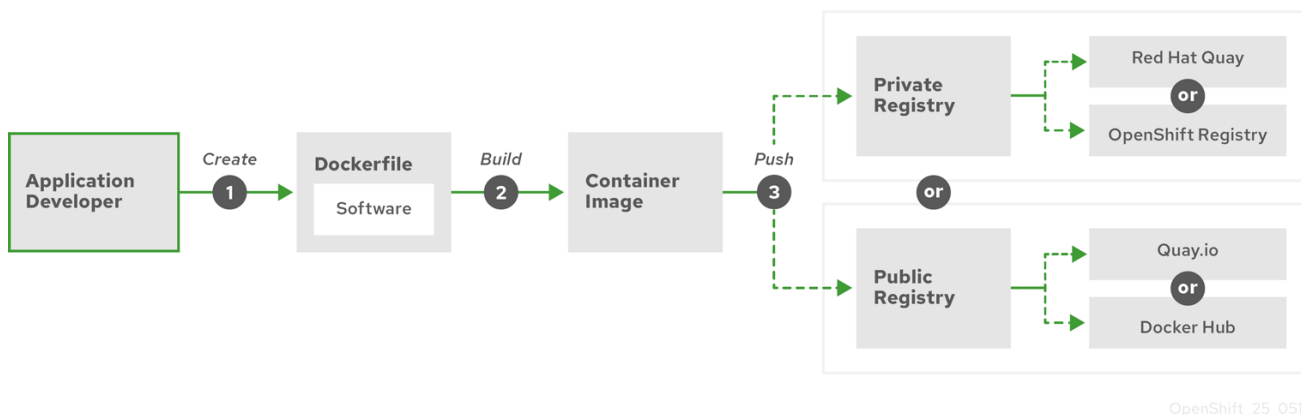
If you use a computer that runs Red Hat Enterprise Linux (RHEL) as the operating system, the process of creating a containerized application requires the following steps:

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.

5.2.1. 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 Dedicated 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 Dedicated, you use the CRI-O container engine. CRI-O runs on every worker and control plane machine in an OpenShift Dedicated cluster, but CRI-O is not yet supported as a standalone runtime outside of OpenShift Dedicated.

5.2.2. 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 Dedicated web UI. In the Developer perspective, navigate to the **+Add** view and in the **Developer Catalog** tile, view all of the available services in the Developer Catalog.

Figure 5.2. Choose S2I base images for apps that need specific runtimes
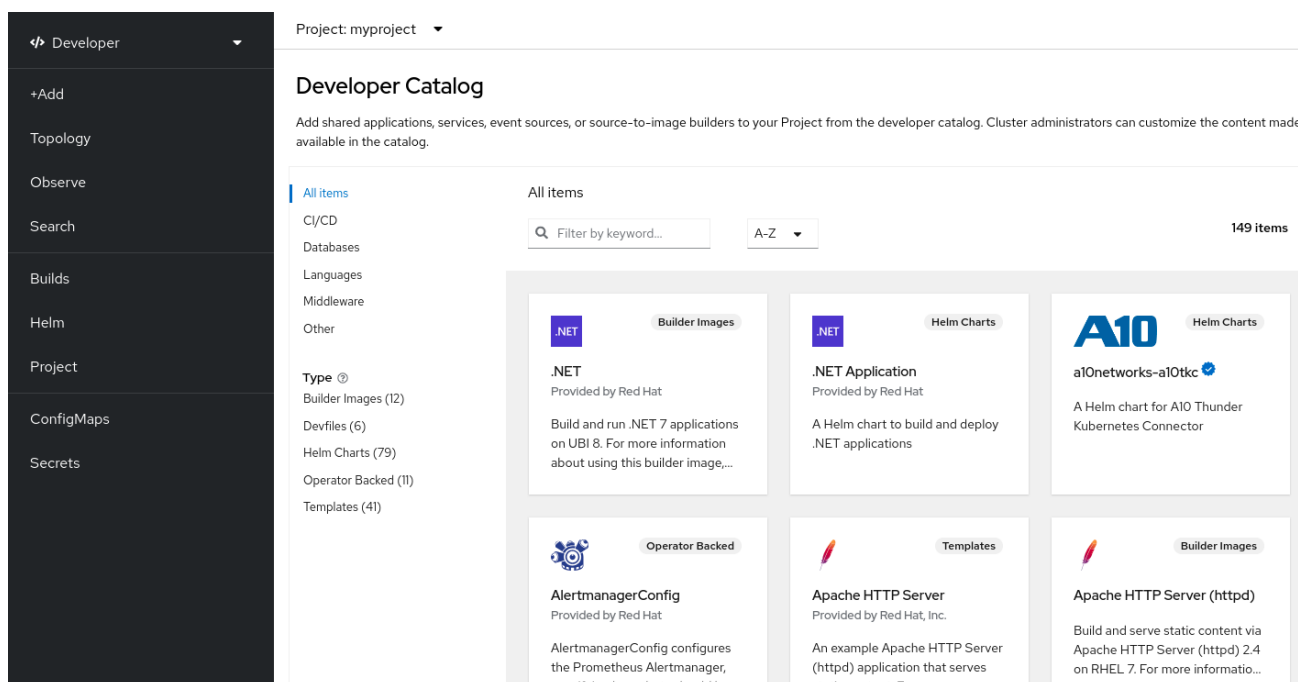
5.2.3. Registry options

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 Dedicated are stored in Quay.io, including container images and the Operators that are used to deploy OpenShift Dedicated 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 Dedicated itself includes a private container registry that is installed with OpenShift Dedicated 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 Dedicated, while other credentials can be assigned to individuals.

5.3. CREATING A KUBERNETES MANIFEST FOR OPENSHIFT DEDICATED

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 Dedicated. 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

5.3.1. 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 Dedicated 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.

5.3.2. 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 Dedicated 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.

5.3.3. 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 Dedicated web console:

- OperatorHub, which is available in each OpenShift Dedicated 4 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.

5.3.4. 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.

5.3.5. 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.

5.4. 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 6. ADMISSION PLUGINS

Admission plugins are used to help regulate how OpenShift Dedicated functions.

6.1. ABOUT ADMISSION PLUGINS

Admission plugins intercept requests to the master API to validate resource requests. After a request is authenticated and authorized, the admission plugins ensure that any associated policies are followed. For example, they are commonly used to enforce security policy, resource limitations or configuration requirements.

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

OpenShift Dedicated has a default set of admission plugins enabled for each resource type. These are required for proper functioning of the cluster. Admission plugins ignore resources that they are not responsible for.

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

Calling webhook servers through a mutating admission plugin 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 plugins in OpenShift Dedicated 4, 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 PLUGINS

Default validating and admission plugins are enabled in OpenShift Dedicated 4. These default plugins contribute to fundamental control plane functionality, such as ingress policy, cluster resource limit override and quota policy.
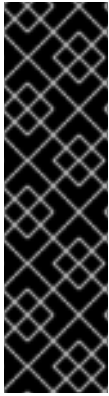
> **IMPORTANT**
>
> Do not run workloads in or share access to default projects. Default projects are reserved for running core cluster components.
>
> The following default projects are considered highly privileged: **default**, **kube-public**, **kube-system**, **openshift**, **openshift-infra**, **openshift-node**, and other system-created projects that have the **openshift.io/run-level** label set to **0** or **1**. Functionality that relies on admission plugins, such as pod security admission, security context constraints, cluster resource quotas, and image reference resolution, does not work in highly privileged projects.

The following lists contain the default admission plugins:

> Example 6.1. Validating admission plugins
>
> - **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 plugins

- **NamespaceLifecycle**

- **LimitRanger**

- **ServiceAccount**

- **NodeRestriction**

- **TaintNodesByCondition**

- **PodNodeSelector**

- **Priority**

- **DefaultTolerationSeconds**

- **PodTolerationRestriction**

- **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 PLUGINS

In addition to OpenShift Dedicated default admission plugins, dynamic admission can be implemented through webhook admission plugins 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 plugins in OpenShift Dedicated:

- During the admission process, the *mutating admission plugin* can perform tasks, such as injecting affinity labels.

- At the end of the admission process, the *validating admission plugin* can be used to make sure an object is configured properly, for example ensuring affinity labels are as expected. If the validation passes, OpenShift Dedicated schedules the object as configured.

When an API request comes in, mutating or validating admission plugins 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.

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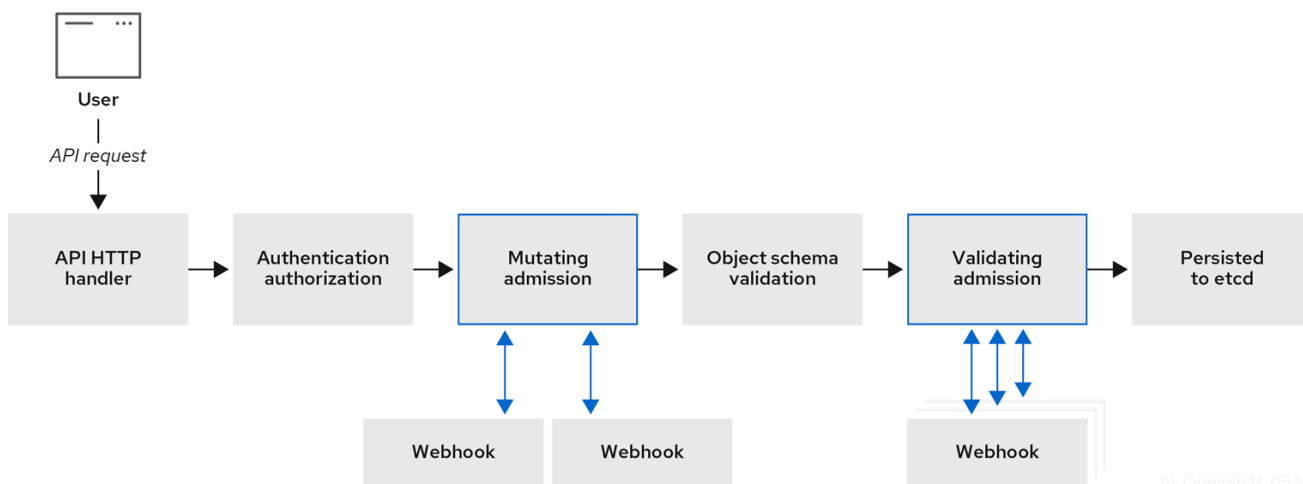
An example webhook admission plugin use case is where all pods must have a common set of labels. In this example, the mutating admission plugin can inject labels and the validating admission plugin can check that labels are as expected. OpenShift Dedicated would subsequently schedule pods that include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

Some common webhook admission plugin use cases include:

- Namespace reservation.

- Limiting custom network resources managed by the SR-IOV network device plugin.

- Pod priority class validation.

NOTE

The maximum default webhook timeout value in OpenShift Dedicated is 13 seconds, and it cannot be changed.

6.4. TYPES OF WEBHOOK ADMISSION PLUGINS

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

6.4.1. Mutating admission plugin

The mutating admission plugin 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 plugin 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 plugin 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 plugin 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 plugin.

[10] One or more operations that trigger the API server to call this webhook admission plugin. 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.
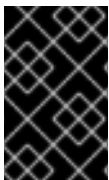
> **IMPORTANT**
>
> In OpenShift Dedicated 4, objects created by users or control loops through a mutating admission plugin might return unexpected results, especially if values set in an initial request are overwritten, which is not recommended.

6.4.2. Validating admission plugin

A validating admission plugin 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 plugin, to ensure that all **nodeSelector** fields are constrained by the node selector restrictions on the namespace.

Sample validating admission plugin 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 plugin 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 plugin.

10 One or more operations that trigger the API server to call this webhook admission plugin. Possible values are **create**, **update**, **delete** or **connect**. Replace **<operation>** and **<resource>** with the

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. ADDITIONAL RESOURCES

- Controlling pod placement using node taints

- Pod priority names