# OpenShift Container Platform 4.8

# Nodes

Configuring and managing nodes in OpenShift Container Platform

# OpenShift Container Platform 4.8 Nodes

Configuring and managing nodes in OpenShift Container Platform

## Legal Notice

## Abstract

This document provides instructions for configuring and managing the nodes, Pods, and containers in your cluster. It also provides information on configuring Pod scheduling and placement, using jobs and DaemonSets to automate tasks, and other tasks to ensure an efficient cluster.

# Table of Contents

# CHAPTER 1. OVERVIEW OF NODES

## 1.1. ABOUT NODES

A node is a virtual or bare-metal machine in a Kubernetes cluster. Worker nodes host your application containers, grouped as pods. The control plane nodes run services that are required to control the Kubernetes cluster. In OpenShift Container Platform, the control plane nodes contain more than just the Kubernetes services for managing the OpenShift Container Platform cluster.

Having stable and healthy nodes in a cluster is fundamental to the smooth functioning of your hosted application. In OpenShift Container Platform, you can access, manage, and monitor a node through the **Node** object representing the node. Using the OpenShift CLI ( **oc**) or the web console, you can perform the following operations on a node.

The following components of a node are responsible for maintaining the running of pods and providing the Kubernetes runtime environment.

- Container runtime:: The container runtime is responsible for running containers. Kubernetes offers several runtimes such as containerd, cri-o, rktlet, and Docker.

- Kubelet:: Kubelet runs on nodes and reads the container manifests. It ensures that the defined containers have started and are running. The kubelet process maintains the state of work and the node server. Kubelet manages network rules and port forwarding. The kubelet manages containers that are created by Kubernetes only.

- Kube-proxy:: Kube-proxy runs on every node in the cluster and maintains the network traffic between the Kubernetes resources. A Kube-proxy ensures that the networking environment is isolated and accessible.

- DNS:: Cluster DNS is a DNS server which serves DNS records for Kubernetes services. Containers started by Kubernetes automatically include this DNS server in their DNS searches.
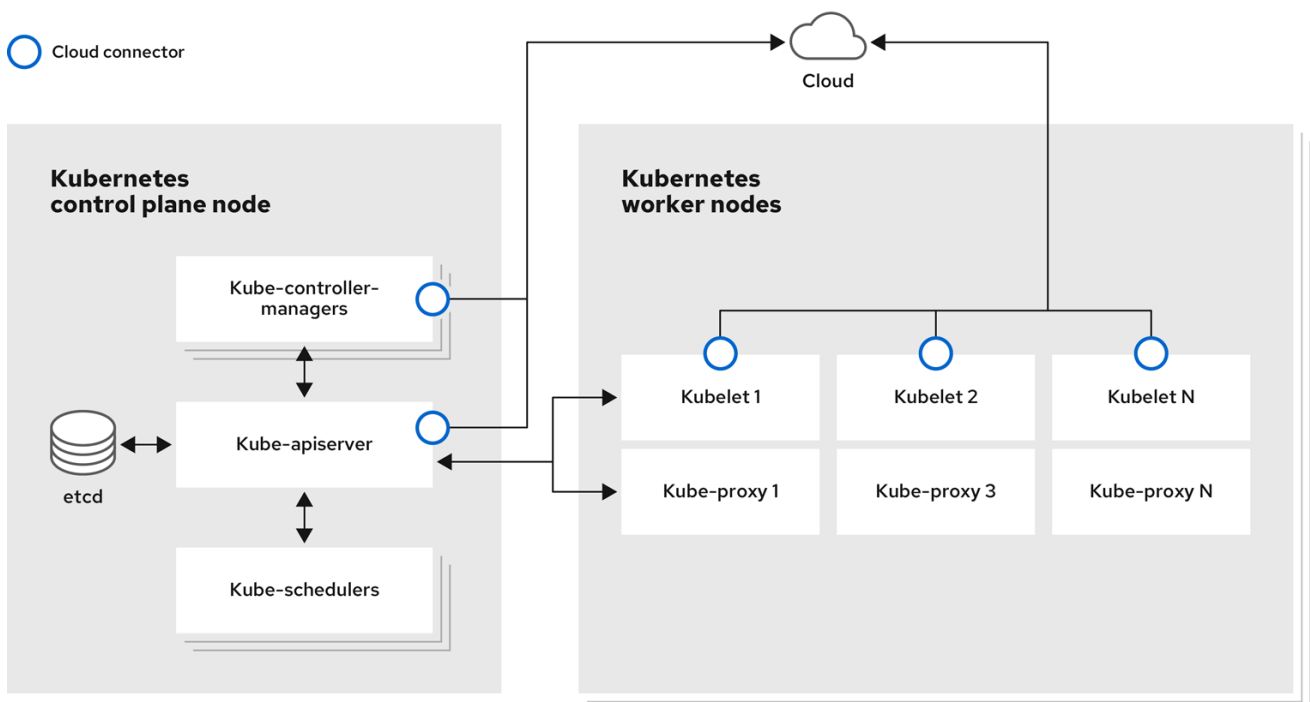
## Read operations

The read operations allow an administrator or a developer to get information about nodes in an OpenShift Container Platform cluster.

- List all the nodes in a cluster .

- Get information about a node, such as memory and CPU usage, health, status, and age.

- List pods running on a node .

## Management operations

As an administrator, you can easily manage a node in an OpenShift Container Platform cluster through several tasks:

- Add or update node labels . A label is a key-value pair applied to a **Node** object. You can control the scheduling of pods using labels.

- Change node configuration using a custom resource definition (CRD), or the **kubeletConfig** object.

- Configure nodes to allow or disallow the scheduling of pods. Healthy worker nodes with a **Ready** status allow pod placement by default while the control plane nodes do not; you can change this default behavior by configuring the worker nodes to be unschedulable  and the control plane nodes to be schedulable.

- Allocate resources for nodes using the **system-reserved** setting. You can allow OpenShift Container Platform to automatically determine the optimal **system-reserved** CPU and memory resources for your nodes, or you can manually determine and set the best resources for your nodes.

- Configure the number of pods that can run on a node  based on the number of processor cores on the node, a hard limit, or both.

- Reboot a node gracefully using pod anti-affinity.

- Delete a node from a cluster  by scaling down the cluster using a machine set. To delete a node from a bare-metal cluster, you must first drain all pods on the node and then manually delete the node.

## Enhancement operations

OpenShift Container Platform allows you to do more than just access and manage nodes; as an administrator, you can perform the following tasks on nodes to make the cluster more efficient, application-friendly, and to provide a better environment for your developers.

- Manage node-level tuning for high-performance applications that require some level of kernel tuning by using the Node Tuning Operator .

- Enable TLS security profiles on the node to protect communication between the kubelet and the Kubernetes API server.

- Run background tasks on nodes automatically with daemon sets . You can create and use daemon sets to create shared storage, run a logging pod on every node, or deploy a monitoring agent on all nodes.

- Free node resources using garbage collection . You can ensure that your nodes are running efficiently by removing terminated containers and the images not referenced by any running pods.

- Add kernel arguments to a set of nodes .

- Configure an OpenShift Container Platform cluster to have worker nodes at the network edge (remote worker nodes). For information on the challenges of having remote worker nodes in an OpenShift Container Platform cluster and some recommended approaches for managing pods on a remote worker node, see Using remote worker nodes at the network edge .

## 1.2. ABOUT PODS

A pod is one or more containers deployed together on a node. As a cluster administrator, you can define a pod, assign it to run on a healthy node that is ready for scheduling, and manage. A pod runs as long as the containers are running. You cannot change a pod once it is defined and is running. Some operations you can perform when working with pods are:

### Read operations
As an administrator, you can get information about pods in a project through the following tasks:

- List pods associated with a project , including information such as the number of replicas and restarts, current status, and age.

- View pod usage statistics such as CPU, memory, and storage consumption.

### Management operations
The following list of tasks provides an overview of how an administrator can manage pods in an OpenShift Container Platform cluster.

- Control scheduling of pods using the advanced scheduling features available in OpenShift Container Platform:

  - Node-to-pod binding rules such as pod affinity, node affinity, and anti-affinity.

  - Node labels and selectors .

  - Taints and tolerations .

  - Pod topology spread constraints .

  - Custom schedulers .

- Configure the descheduler to evict pods based on specific strategies so that the scheduler reschedules the pods to more appropriate nodes.

- Configure how pods behave after a restart using pod controllers and restart policies .

- Limit both egress and ingress traffic on a pod .

- Add and remove volumes to and from any object that has a pod template . A volume is a mounted file system available to all the containers in a pod. Container storage is ephemeral; you can use volumes to persist container data.

### Enhancement operations
You can work with pods more easily and efficiently with the help of various tools and features available in OpenShift Container Platform. The following operations involve using those tools and features to better manage pods.

| Operation | User | More information |
|---|---|---|
| Create and use a horizontal pod autoscaler. | Developer | You can use a horizontal pod autoscaler to specify the minimum and the maximum number of pods you want to run, as well as the CPU utilization or memory utilization your pods should target. Using a horizontal pod autoscaler, you can automatically scale pods. |
| Install and use a vertical pod autoscaler. | Administrator and developer | As an administrator, use a vertical pod autoscaler to better use cluster resources by monitoring the resources and the resource requirements of workloads.<br><br>As a developer, use a vertical pod autoscaler to ensure your pods stay up during periods of high demand by scheduling pods to nodes that have enough resources for each pod. |
| Provide access to external resources using device plugins. | Administrator | A device plugin is a gRPC service running on nodes (external to the kubelet), which manages specific hardware resources. You can deploy a device plugin to provide a consistent and portable solution to consume hardware devices across clusters. |
| Provide sensitive data to pods using the **Secret** object. | Administrator | Some applications need sensitive information, such as passwords and usernames. You can use the **Secret** object to provide such information to an application pod. |

## 1.3. ABOUT CONTAINERS

A container is the basic unit of an OpenShift Container Platform application, which comprises the application code packaged along with its dependencies, libraries, and binaries. Containers provide consistency across environments and multiple deployment targets: physical servers, virtual machines (VMs), and private or public cloud.

Linux container technologies are lightweight mechanisms for isolating running processes and limiting access to only designated resources. As an administrator, You can perform various tasks on a Linux container, such as:

- Copy files to and from a container .

- Allow containers to consume API objects.

- Execute remote commands in a container .

- Use port forwarding to access applications in a container .

OpenShift Container Platform provides specialized containers called Init containers. Init containers run before application containers and can contain utilities or setup scripts not present in an application image. You can use an Init container to perform tasks before the rest of a pod is deployed.

Apart from performing specific tasks on nodes, pods, and containers, you can work with the overall OpenShift Container Platform cluster to keep the cluster efficient and the application pods highly available.

# CHAPTER 2. WORKING WITH PODS

## 2.1. USING PODS

A *pod* is one or more containers deployed together on one host, and the smallest compute unit that can be defined, deployed, and managed.

### 2.1.1. Understanding pods

Pods are the rough equivalent of a machine instance (physical or virtual) to a Container. Each pod is allocated its own internal IP address, therefore owning its entire port space, and containers within pods can share their local storage and networking.

Pods have a lifecycle; they are defined, then they are assigned to run on a node, then they run until their container(s) exit or they are removed for some other reason. Pods, depending on policy and exit code, might be removed after exiting, or can be retained to enable access to the logs of their containers.

OpenShift Container Platform treats pods as largely immutable; changes cannot be made to a pod definition while it is running. OpenShift Container Platform implements changes by terminating an existing pod and recreating it with modified configuration, base image(s), or both. Pods are also treated as expendable, and do not maintain state when recreated. Therefore pods should usually be managed by higher-level controllers, rather than directly by users.

> **NOTE**
>
> For the maximum number of pods per OpenShift Container Platform node host, see the Cluster Limits.

> **WARNING**
>
> Bare pods that are not managed by a replication controller will be not rescheduled upon node disruption.

### 2.1.2. Example pod configurations

OpenShift Container Platform leverages the Kubernetes concept of a *pod*, which is one or more containers deployed together on one host, and the smallest compute unit that can be defined, deployed, and managed.

The following is an example definition of a pod from a Rails application. It demonstrates many features of pods, most of which are discussed in other topics and thus only briefly mentioned here:

**Pod object definition (YAML)**

```
kind: Pod
apiVersion: v1
metadata:
  name: example
  namespace: default
```

```
    selfLink: /api/v1/namespaces/default/pods/example
    uid: 5cc30063-0265780783bc
    resourceVersion: '165032'
    creationTimestamp: '2019-02-13T20:31:37Z'
    labels:
      app: hello-openshift ❶
    annotations:
      openshift.io/scc: anyuid
  spec:
    restartPolicy: Always ❷
    serviceAccountName: default
    imagePullSecrets:
      - name: default-dockercfg-5zrhb
    priority: 0
    schedulerName: default-scheduler
    terminationGracePeriodSeconds: 30
    nodeName: ip-10-0-140-16.us-east-2.compute.internal
    securityContext: ❸
      seLinuxOptions:
        level: 's0:c11,c10'
    containers: ❹
      - resources: {}
        terminationMessagePath: /dev/termination-log
        name: hello-openshift
        securityContext:
          capabilities:
            drop:
              - MKNOD
          procMount: Default
        ports:
          - containerPort: 8080
            protocol: TCP
        imagePullPolicy: Always
        volumeMounts: ❺
          - name: default-token-wbqsl
            readOnly: true
            mountPath: /var/run/secrets/kubernetes.io/serviceaccount ❻
        terminationMessagePolicy: File
        image: registry.redhat.io/openshift4/ose-ogging-eventrouter:v4.3 ❼
    serviceAccount: default ❽
    volumes: ❾
      - name: default-token-wbqsl
        secret:
          secretName: default-token-wbqsl
          defaultMode: 420
    dnsPolicy: ClusterFirst
  status:
    phase: Pending
    conditions:
      - type: Initialized
        status: 'True'
        lastProbeTime: null
        lastTransitionTime: '2019-02-13T20:31:37Z'
      - type: Ready
        status: 'False'
```

```
          lastProbeTime: null
          lastTransitionTime: '2019-02-13T20:31:37Z'
          reason: ContainersNotReady
          message: 'containers with unready status: [hello-openshift]'
        - type: ContainersReady
          status: 'False'
          lastProbeTime: null
          lastTransitionTime: '2019-02-13T20:31:37Z'
          reason: ContainersNotReady
          message: 'containers with unready status: [hello-openshift]'
        - type: PodScheduled
          status: 'True'
          lastProbeTime: null
          lastTransitionTime: '2019-02-13T20:31:37Z'
     hostIP: 10.0.140.16
     startTime: '2019-02-13T20:31:37Z'
     containerStatuses:
       - name: hello-openshift
         state:
           waiting:
             reason: ContainerCreating
         lastState: {}
         ready: false
         restartCount: 0
         image: openshift/hello-openshift
         imageID: ''
     qosClass: BestEffort
```

**1**  Pods can be "tagged" with one or more labels, which can then be used to select and manage groups of pods in a single operation. The labels are stored in key/value format in the **metadata** hash.

**2**  The pod restart policy with possible values **Always**, **OnFailure**, and **Never**. The default value is **Always**.

**3**  OpenShift Container Platform defines a security context for containers which specifies whether they are allowed to run as privileged containers, run as a user of their choice, and more. The default context is very restrictive but administrators can modify this as needed.

**4**  **containers** specifies an array of one or more container definitions.

**5**  The container specifies where external storage volumes are mounted within the container. In this case, there is a volume for storing access to credentials the registry needs for making requests against the OpenShift Container Platform API.

**6**  Specify the volumes to provide for the pod. Volumes mount at the specified path. Do not mount to the container root, /, or any path that is the same in the host and the container. This can corrupt your host system if the container is sufficiently privileged, such as the host /**dev/pts** files. It is safe to mount the host by using /**host**.

**7**  Each container in the pod is instantiated from its own container image.

**8**  Pods making requests against the OpenShift Container Platform API is a common enough pattern that there is a **serviceAccount** field for specifying which service account user the pod should authenticate as when making the requests. This enables fine-grained access control for custom infrastructure components.

9 The pod defines storage volumes that are available to its container(s) to use. In this case, it provides an ephemeral volume for a **secret** volume containing the default service account tokens.

If you attach persistent volumes that have high file counts to pods, those pods can fail or can take a long time to start. For more information, see When using Persistent Volumes with high file counts in OpenShift, why do pods fail to start or take an excessive amount of time to achieve "Ready" state?.

> **NOTE**
>
> This pod definition does not include attributes that are filled by OpenShift Container Platform automatically after the pod is created and its lifecycle begins. The Kubernetes pod documentation has details about the functionality and purpose of pods.

### 2.1.3. Additional resources

- For more information on pods and storage see Understanding persistent storage and Understanding ephemeral storage .

## 2.2. VIEWING PODS

As an administrator, you can view the pods in your cluster and to determine the health of those pods and the cluster as a whole.

### 2.2.1. About pods

OpenShift Container Platform leverages the Kubernetes concept of a *pod*, which is one or more containers deployed together on one host, and the smallest compute unit that can be defined, deployed, and managed. Pods are the rough equivalent of a machine instance (physical or virtual) to a container.

You can view a list of pods associated with a specific project or view usage statistics about pods.

### 2.2.2. Viewing pods in a project

You can view a list of pods associated with the current project, including the number of replica, the current status, number or restarts and the age of the pod.

**Procedure**

To view the pods in a project:

1. Change to the project:

   ```
   $ oc project <project-name>
   ```

2. Run the following command:

   ```
   $ oc get pods
   ```

   For example:

   ```
   $ oc get pods -n openshift-console
   ```

**Example output**

```
NAME                READY  STATUS   RESTARTS  AGE
console-698d866b78-bnshf  1/1    Running  2       165m
console-698d866b78-m87pm  1/1    Running  2       165m
```

Add the **-o wide** flags to view the pod IP address and the node where the pod is located.

```
$ oc get pods -o wide
```

**Example output**

```
NAME                READY  STATUS   RESTARTS  AGE    IP        NODE
NOMINATED NODE
console-698d866b78-bnshf  1/1    Running  2       166m   10.128.0.24  ip-10-0-152-
71.ec2.internal   <none>
console-698d866b78-m87pm  1/1    Running  2       166m   10.129.0.23  ip-10-0-173-
237.ec2.internal   <none>
```

### 2.2.3. Viewing pod usage statistics

You can display usage statistics about pods, which provide the runtime environments for containers. These usage statistics include CPU, memory, and storage consumption.

**Prerequisites**

- You must have **cluster-reader** permission to view the usage statistics.

- Metrics must be installed to view the usage statistics.

**Procedure**

To view the usage statistics:

1. Run the following command:

   ```
   $ oc adm top pods
   ```

   For example:

   ```
   $ oc adm top pods -n openshift-console
   ```

   **Example output**

   ```
   NAME                CPU(cores)  MEMORY(bytes)
   console-7f58c69899-q8c8k   0m       22Mi
   console-7f58c69899-xhbgg   0m       25Mi
   downloads-594fcccf94-bcxk8  3m        18Mi
   downloads-594fcccf94-kv4p6  2m        15Mi
   ```

2. Run the following command to view the usage statistics for pods with labels:

   ```
   $ oc adm top pod --selector=''
   ```

You must choose the selector (label query) to filter on. Supports **=**, **==**, and **!=**.

## 2.2.4. Viewing resource logs

You can view the log for various resources in the OpenShift CLI (oc) and web console. Logs read from the tail, or end, of the log.

**Prerequisites**

- Access to the OpenShift CLI (oc).

**Procedure (UI)**

1. In the OpenShift Container Platform console, navigate to **Workloads → Pods** or navigate to the pod through the resource you want to investigate.

    **NOTE**

    Some resources, such as builds, do not have pods to query directly. In such instances, you can locate the **Logs** link on the **Details** page for the resource.

2. Select a project from the drop-down menu.

3. Click the name of the pod you want to investigate.

4. Click **Logs**.

**Procedure (CLI)**

- View the log for a specific pod:

    ```
    $ oc logs -f <pod_name> -c <container_name>
    ```

    where:

    **-f**
    Optional: Specifies that the output follows what is being written into the logs.
    **<pod_name>**
    Specifies the name of the pod.
    **<container_name>**
    Optional: Specifies the name of a container. When a pod has more than one container, you must specify the container name.

    For example:

    ```
    $ oc logs ruby-58cd97df55-mww7r
    ```

    ```
    $ oc logs -f ruby-57f7f4855b-znl92 -c ruby
    ```

    The contents of log files are printed out.

- View the log for a specific resource:

  ```
  $ oc logs <object_type>/<resource_name>  ❶
  ```

  ❶ Specifies the resource type and name.

  For example:

  ```
  $ oc logs deployment/ruby
  ```

  The contents of log files are printed out.

## 2.3. CONFIGURING AN OPENSHIFT CONTAINER PLATFORM CLUSTER FOR PODS

As an administrator, you can create and maintain an efficient cluster for pods.

By keeping your cluster efficient, you can provide a better environment for your developers using such tools as what a pod does when it exits, ensuring that the required number of pods is always running, when to restart pods designed to run only once, limit the bandwidth available to pods, and how to keep pods running during disruptions.

### 2.3.1. Configuring how pods behave after restart

A pod restart policy determines how OpenShift Container Platform responds when Containers in that pod exit. The policy applies to all Containers in that pod.

The possible values are:

- **Always** - Tries restarting a successfully exited Container on the pod continuously, with an exponential back-off delay (10s, 20s, 40s) capped at 5 minutes. The default is **Always**.

- **OnFailure** - Tries restarting a failed Container on the pod with an exponential back-off delay (10s, 20s, 40s) capped at 5 minutes.

- **Never** - Does not try to restart exited or failed Containers on the pod. Pods immediately fail and exit.

After the pod is bound to a node, the pod will never be bound to another node. This means that a controller is necessary in order for a pod to survive node failure:

| Condition | Controller Type | Restart Policy |
|---|---|---|
| Pods that are expected to terminate (such as batch computations) | Job | **OnFailure** or **Never** |
| Pods that are expected to not terminate (such as web servers) | Replication controller | **Always**. |

| Condition | Controller Type | Restart Policy |
|-----------|-----------------|----------------|
| Pods that must run one-per-machine | Daemon set | Any |

If a Container on a pod fails and the restart policy is set to **OnFailure**, the pod stays on the node and the Container is restarted. If you do not want the Container to restart, use a restart policy of **Never**.

If an entire pod fails, OpenShift Container Platform starts a new pod. Developers must address the possibility that applications might be restarted in a new pod. In particular, applications must handle temporary files, locks, incomplete output, and so forth caused by previous runs.
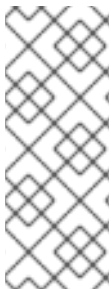
> **NOTE**
>
> Kubernetes architecture expects reliable endpoints from cloud providers. When a cloud provider is down, the kubelet prevents OpenShift Container Platform from restarting.
>
> If the underlying cloud provider endpoints are not reliable, do not install a cluster using cloud provider integration. Install the cluster as if it was in a no-cloud environment. It is not recommended to toggle cloud provider integration on or off in an installed cluster.

For details on how OpenShift Container Platform uses restart policy with failed Containers, see the Example States in the Kubernetes documentation.

## 2.3.2. Limiting the bandwidth available to pods

You can apply quality-of-service traffic shaping to a pod and effectively limit its available bandwidth. Egress traffic (from the pod) is handled by policing, which simply drops packets in excess of the configured rate. Ingress traffic (to the pod) is handled by shaping queued packets to effectively handle data. The limits you place on a pod do not affect the bandwidth of other pods.

**Procedure**

To limit the bandwidth on a pod:

1. Write an object definition JSON file, and specify the data traffic speed using **kubernetes.io/ingress-bandwidth** and **kubernetes.io/egress-bandwidth** annotations. For example, to limit both pod egress and ingress bandwidth to 10M/s:

   **Limited Pod object definition**

   ```
   {
       "kind": "Pod",
       "spec": {
           "containers": [
               {
                   "image": "openshift/hello-openshift",
                   "name": "hello-openshift"
               }
           ]
       },
       "apiVersion": "v1",
       "metadata": {
   ```

```
        "name": "iperf-slow",
        "annotations": {
            "kubernetes.io/ingress-bandwidth": "10M",
            "kubernetes.io/egress-bandwidth": "10M"
        }
    }
}
```

2. Create the pod using the object definition:

```
$ oc create -f <file_or_dir_path>
```

### 2.3.3. Understanding how to use pod disruption budgets to specify the number of pods that must be up

A *pod disruption budget* is part of the Kubernetes API, which can be managed with **oc** commands like other object types. They allow the specification of safety constraints on pods during operations, such as draining a node for maintenance.

**PodDisruptionBudget** is an API object that specifies the minimum number or percentage of replicas that must be up at a time. Setting these in projects can be helpful during node maintenance (such as scaling a cluster down or a cluster upgrade) and is only honored on voluntary evictions (not on node failures).

A **PodDisruptionBudget** object's configuration consists of the following key parts:

- A label selector, which is a label query over a set of pods.

- An availability level, which specifies the minimum number of pods that must be available simultaneously, either:

  - **minAvailable** is the number of pods must always be available, even during a disruption.

  - **maxUnavailable** is the number of pods can be unavailable during a disruption.

> **NOTE**
>
> A **maxUnavailable** of **0%** or **0** or a **minAvailable** of **100%** or equal to the number of replicas is permitted but can block nodes from being drained.

You can check for pod disruption budgets across all projects with the following:

```
$ oc get poddisruptionbudget --all-namespaces
```

**Example output**

```
NAMESPACE         NAME          MIN-AVAILABLE   SELECTOR
another-project   another-pdb   4               bar=foo
test-project      my-pdb        2               foo=bar
```

The **PodDisruptionBudget** is considered healthy when there are at least **minAvailable** pods running in the system. Every pod above that limit can be evicted.

> **NOTE**
>
> Depending on your pod priority and preemption settings, lower-priority pods might be removed despite their pod disruption budget requirements.

### 2.3.3.1. Specifying the number of pods that must be up with pod disruption budgets

You can use a **PodDisruptionBudget** object to specify the minimum number or percentage of replicas that must be up at a time.

### Procedure

To configure a pod disruption budget:

1. Create a YAML file with the an object definition similar to the following:

   ```
   apiVersion: policy/v1    1
   kind: PodDisruptionBudget
   metadata:
     name: my-pdb
   spec:
     minAvailable: 2    2
     selector:    3
       matchLabels:
         foo: bar
   ```

   **1**    **PodDisruptionBudget** is part of the **policy/v1** API group.

   **2**    The minimum number of pods that must be available simultaneously. This can be either an integer or a string specifying a percentage, for example, **20%**.

   **3**    A label query over a set of resources. The result of **matchLabels** and **matchExpressions** are logically conjoined. Leave this paramter blank, for example **selector {}**, to select all pods in the project.

   Or:

   ```
   apiVersion: policy/v1    1
   kind: PodDisruptionBudget
   metadata:
     name: my-pdb
   spec:
     maxUnavailable: 25%    2
     selector:    3
       matchLabels:
         foo: bar
   ```

   **1**    **PodDisruptionBudget** is part of the **policy/v1** API group.

   **2**    The maximum number of pods that can be unavailable simultaneously. This can be either an integer or a string specifying a percentage, for example, **20%**.

   **3**

A label query over a set of resources. The result of **matchLabels** and **matchExpressions** are logically conjoined. Leave this paramter blank, for example **selector {}**, to select all

2. Run the following command to add the object to project:

```
$ oc create -f </path/to/file> -n <project_name>
```

### 2.3.4. Preventing pod removal using critical pods

There are a number of core components that are critical to a fully functional cluster, but, run on a regular cluster node rather than the master. A cluster might stop working properly if a critical add-on is evicted.

Pods marked as critical are not allowed to be evicted.

#### Procedure

To make a pod critical:

1. Create a **Pod** spec or edit existing pods to include the **system-cluster-critical** priority class:

```
spec:
  template:
    metadata:
      name: critical-pod
    priorityClassName: system-cluster-critical 1
```

**1** Default priority class for pods that should never be evicted from a node.

Alternatively, you can specify **system-node-critical** for pods that are important to the cluster but can be removed if necessary.

2. Create the pod:

```
$ oc create -f <file-name>.yaml
```

## 2.4. AUTOMATICALLY SCALING PODS WITH THE HORIZONTAL POD AUTOSCALER

As a developer, you can use a horizontal pod autoscaler (HPA) to specify how OpenShift Container Platform should automatically increase or decrease the scale of a replication controller or deployment configuration, based on metrics collected from the pods that belong to that replication controller or deployment configuration. You can create an HPA for any **Deployment**, **DeploymentConfig**, **ReplicaSet**, **ReplicationController**, or **StatefulSet** object.

> **NOTE**
>
> It is recommended to use a **Deployment** object or **ReplicaSet** object unless you need a specific feature or behavior provided by other objects. For more information on these objects, see Understanding Deployment and DeploymentConfig objects .

### 2.4.1. Understanding horizontal pod autoscalers

You can create a horizontal pod autoscaler to specify the minimum and maximum number of pods you want to run, as well as the CPU utilization or memory utilization your pods should target.

After you create a horizontal pod autoscaler, OpenShift Container Platform begins to query the CPU and/or memory resource metrics on the pods. When these metrics are available, the horizontal pod autoscaler computes the ratio of the current metric utilization with the desired metric utilization, and scales up or down accordingly. The query and scaling occurs at a regular interval, but can take one to two minutes before metrics become available.

For replication controllers, this scaling corresponds directly to the replicas of the replication controller. For deployment configurations, scaling corresponds directly to the replica count of the deployment configuration. Note that autoscaling applies only to the latest deployment in the **Complete** phase.

OpenShift Container Platform automatically accounts for resources and prevents unnecessary autoscaling during resource spikes, such as during start up. Pods in the **unready** state have **0 CPU** usage when scaling up and the autoscaler ignores the pods when scaling down. Pods without known metrics have **0% CPU** usage when scaling up and **100% CPU** when scaling down. This allows for more stability during the HPA decision. To use this feature, you must configure readiness checks to determine if a new pod is ready for use.

To use horizontal pod autoscalers, your cluster administrator must have properly configured cluster metrics.

### 2.4.1.1. Supported metrics

The following metrics are supported by horizontal pod autoscalers:

Table 2.1. Metrics

| Metric | Description | API version |
|---|---|---|
| CPU utilization | Number of CPU cores used. Can be used to calculate a percentage of the pod's requested CPU. | **autoscaling/v1**, **autoscaling/v2beta2** |
| Memory utilization | Amount of memory used. Can be used to calculate a percentage of the pod's requested memory. | **autoscaling/v2beta2** |

IMPORTANT

For memory-based autoscaling, memory usage must increase and decrease proportionally to the replica count. On average:

- An increase in replica count must lead to an overall decrease in memory (working set) usage per-pod.

- A decrease in replica count must lead to an overall increase in per-pod memory usage.

Use the OpenShift Container Platform web console to check the memory behavior of your application and ensure that your application meets these requirements before using memory-based autoscaling.

The following example shows autoscaling for the **image-registry Deployment** object. The initial deployment requires 3 pods. The HPA object increases the minimum to 5. If CPU usage on the pods reaches 75%, the pods increase to 7:

```
$ oc autoscale deployment/image-registry --min=5 --max=7 --cpu-percent=75
```

**Example output**

```
horizontalpodautoscaler.autoscaling/image-registry autoscaled
```

**Sample HPA for the image-registry Deployment object with minReplicas set to 3**

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: image-registry
  namespace: default
spec:
  maxReplicas: 7
  minReplicas: 3
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: image-registry
  targetCPUUtilizationPercentage: 75
status:
  currentReplicas: 5
  desiredReplicas: 0
```

1. View the new state of the deployment:

   ```
   $ oc get deployment image-registry
   ```

   There are now 5 pods in the deployment:

   **Example output**

   ```
   NAME            REVISION   DESIRED   CURRENT   TRIGGERED BY
   image-registry  1          5         5         config
   ```

### 2.4.1.2. Scaling policies

The **autoscaling/v2beta2** API allows you to add *scaling policies* to a horizontal pod autoscaler. A scaling policy controls how the OpenShift Container Platform horizontal pod autoscaler (HPA) scales pods. Scaling policies allow you to restrict the rate that HPAs scale pods up or down by setting a specific number or specific percentage to scale in a specified period of time. You can also define a *stabilization window*, which uses previously computed desired states to control scaling if the metrics are fluctuating. You can create multiple policies for the same scaling direction, and determine which policy is used, based on the amount of change. You can also restrict the scaling by timed iterations. The HPA scales pods during an iteration, then performs scaling, as needed, in further iterations.

**Sample HPA object with a scaling policy**

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-resource-metrics-memory
  namespace: default
spec:
  behavior:
    scaleDown: 1
      policies: 2
      - type: Pods 3
        value: 4 4
        periodSeconds: 60 5
      - type: Percent
        value: 10 6
        periodSeconds: 60
      selectPolicy: Min 7
      stabilizationWindowSeconds: 300 8
    scaleUp: 9
      policies:
      - type: Pods
        value: 5 10
        periodSeconds: 70
      - type: Percent
        value: 12 11
        periodSeconds: 80
      selectPolicy: Max
      stabilizationWindowSeconds: 0
...
```

[1] Specifies the direction for the scaling policy, either **scaleDown** or **scaleUp**. This example creates a policy for scaling down.

[2] Defines the scaling policy.

[3] Determines if the policy scales by a specific number of pods or a percentage of pods during each iteration. The default value is **pods**.

[4] Determines the amount of scaling, either the number of pods or percentage of pods, during each iteration. There is no default value for scaling down by number of pods.

[5] Determines the length of a scaling iteration. The default value is **15** seconds.

[6] The default value for scaling down by percentage is 100%.

[7] Determines which policy to use first, if multiple policies are defined. Specify **Max** to use the policy that allows the highest amount of change, **Min** to use the policy that allows the lowest amount of change, or **Disabled** to prevent the HPA from scaling in that policy direction. The default value is **Max**.

[8] Determines the time period the HPA should look back at desired states. The default value is **0**.

[9] This example creates a policy for scaling up.

[10] The amount of scaling up by the number of pods. The default value for scaling up the number of pods is 4%.

**11** The amount of scaling up by the percentage of pods. The default value for scaling up by percentage is 100%.

## Example policy for scaling down

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-resource-metrics-memory
  namespace: default
spec:
...
  minReplicas: 20
...
  behavior:
    scaleDown:
      stabilizationWindowSeconds: 300
      policies:
      - type: Pods
        value: 4
        periodSeconds: 30
      - type: Percent
        value: 10
        periodSeconds: 60
      selectPolicy: Max
    scaleUp:
      selectPolicy: Disabled
```

In this example, when the number of pods is greater than 40, the percent-based policy is used for scaling down, as that policy results in a larger change, as required by the **selectPolicy**.

If there are 80 pod replicas, in the first iteration the HPA reduces the pods by 8, which is 10% of the 80 pods (based on the **type: Percent** and **value: 10** parameters), over one minute ( **periodSeconds: 60**). For the next iteration, the number of pods is 72. The HPA calculates that 10% of the remaining pods is 7.2, which it rounds up to 8 and scales down 8 pods. On each subsequent iteration, the number of pods to be scaled is re-calculated based on the number of remaining pods. When the number of pods falls below 40, the pods-based policy is applied, because the pod-based number is greater than the percent-based number. The HPA reduces 4 pods at a time (**type: Pods** and **value: 4**), over 30 seconds (**periodSeconds: 30**), until there are 20 replicas remaining ( **minReplicas**).

The **selectPolicy: Disabled** parameter prevents the HPA from scaling up the pods. You can manually scale up by adjusting the number of replicas in the replica set or deployment set, if needed.

If set, you can view the scaling policy by using the **oc edit** command:

```
$ oc edit hpa hpa-resource-metrics-memory
```

## Example output

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  annotations:
    autoscaling.alpha.kubernetes.io/behavior:\
```

```
'{"ScaleUp":{"StabilizationWindowSeconds":0,"SelectPolicy":"Max","Policies":
[{"Type":"Pods","Value":4,"PeriodSeconds":15},{"Type":"Percent","Value":100,"PeriodSeconds":15}]},\
"ScaleDown":{"StabilizationWindowSeconds":300,"SelectPolicy":"Min","Policies":
[{"Type":"Pods","Value":4,"PeriodSeconds":60},{"Type":"Percent","Value":10,"PeriodSeconds":60}]}}'
...
```

## 2.4.2. Creating a horizontal pod autoscaler by using the web console

From the web console, you can create a horizontal pod autoscaler (HPA) that specifies the minimum and maximum number of pods you want to run on a **Deployment** or **DeploymentConfig** object. You can also define the amount of CPU or memory usage that your pods should target.

> **NOTE**
>
> An HPA cannot be added to deployments that are part of an Operator-backed service, Knative service, or Helm chart.

### Procedure

To create an HPA in the web console:

1. In the **Topology** view, click the node to reveal the side pane.

2. From the **Actions** drop-down list, select **Add HorizontalPodAutoscaler** to open the **Add HorizontalPodAutoscaler** form.

   Figure 2.1. Add HorizontalPodAutoscaler

   

3. From the **Add HorizontalPodAutoscaler** form, define the name, minimum and maximum pod limits, the CPU and memory usage, and click **Save**.

   > **NOTE**
   >
   > If any of the values for CPU and memory usage are missing, a warning is displayed.

To edit an HPA in the web console:

1. In the **Topology** view, click the node to reveal the side pane.

2. From the **Actions** drop-down list, select **Edit HorizontalPodAutoscaler** to open the **Edit Horizontal Pod Autoscaler** form.

3. From the **Edit Horizontal Pod Autoscaler** form, edit the minimum and maximum pod limits and the CPU and memory usage, and click **Save**.

> **NOTE**
>
> While creating or editing the horizontal pod autoscaler in the web console, you can switch from **Form view** to **YAML view**.

To remove an HPA in the web console:

1. In the **Topology** view, click the node to reveal the side panel.

2. From the **Actions** drop-down list, select **Remove HorizontalPodAutoscaler**.

3. In the confirmation pop-up window, click **Remove** to remove the HPA.

## 2.4.3. Creating a horizontal pod autoscaler for CPU utilization by using the CLI

Using the OpenShift Container Platform CLI, you can create a horizontal pod autoscaler (HPA) to automatically scale an existing **Deployment**, **DeploymentConfig**, **ReplicaSet**, **ReplicationController**, or **StatefulSet** object. The HPA scales the pods associated with that object to maintain the CPU usage you specify.

> **NOTE**
>
> It is recommended to use a **Deployment** object or **ReplicaSet** object unless you need a specific feature or behavior provided by other objects.

The HPA increases and decreases the number of replicas between the minimum and maximum numbers to maintain the specified CPU utilization across all pods.

When autoscaling for CPU utilization, you can use the **oc autoscale** command and specify the minimum and maximum number of pods you want to run at any given time and the average CPU utilization your pods should target. If you do not specify a minimum, the pods are given default values from the OpenShift Container Platform server.

To autoscale for a specific CPU value, create a **HorizontalPodAutoscaler** object with the target CPU and pod limits.

### Prerequisites

To use horizontal pod autoscalers, your cluster administrator must have properly configured cluster metrics. You can use the **oc describe PodMetrics <pod-name>** command to determine if metrics are configured. If metrics are configured, the output appears similar to the following, with **Cpu** and **Memory** displayed under **Usage**.

```
$ oc describe PodMetrics openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
```

**Example output**

```
Name:         openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
Namespace:    openshift-kube-scheduler
Labels:       <none>
Annotations:  <none>
API Version:  metrics.k8s.io/v1beta1
Containers:
 Name:  wait-for-host-port
 Usage:
  Memory:  0
 Name:      scheduler
 Usage:
  Cpu:     8m
  Memory:  45440Ki
Kind:         PodMetrics
Metadata:
 Creation Timestamp:  2019-05-23T18:47:56Z
 Self Link:           /apis/metrics.k8s.io/v1beta1/namespaces/openshift-kube-scheduler/pods/openshift-
kube-scheduler-ip-10-0-135-131.ec2.internal
 Timestamp:           2019-05-23T18:47:56Z
 Window:              1m0s
Events:                <none>
```

**Procedure**

To create a horizontal pod autoscaler for CPU utilization:

1. Perform one of the following:

   - To scale based on the percent of CPU utilization, create a **HorizontalPodAutoscaler** object for an existing object:

     ```
     $ oc autoscale <object_type>/<name> \     1
       --min <number> \                         2
       --max <number> \                         3
       --cpu-percent=<percent>                  4
     ```

     **1**     Specify the type and name of the object to autoscale. The object must exist and be a **Deployment**, **DeploymentConfig/dc**, **ReplicaSet/rs**, **ReplicationController/rc**, or **StatefulSet**.

     **2**     Optionally, specify the minimum number of replicas when scaling down.

     **3**     Specify the maximum number of replicas when scaling up.

     **4**     Specify the target average CPU utilization over all the pods, represented as a percent of requested CPU. If not specified or negative, a default autoscaling policy is used.

     For example, the following command shows autoscaling for the **image-registry Deployment** object. The initial deployment requires 3 pods. The HPA object increases the minimum to 5. If CPU usage on the pods reaches 75%, the pods will increase to 7:

     ```
     $ oc autoscale deployment/image-registry --min=5 --max=7 --cpu-percent=75
     ```

- To scale for a specific CPU value, create a YAML file similar to the following for an existing object:

  a. Create a YAML file similar to the following:

  ```
  apiVersion: autoscaling/v2beta2 1
  kind: HorizontalPodAutoscaler
  metadata:
    name: cpu-autoscale 2
    namespace: default
  spec:
    scaleTargetRef:
      apiVersion: apps/v1 3
      kind: Deployment 4
      name: example 5
    minReplicas: 1 6
    maxReplicas: 10 7
    metrics: 8
    - type: Resource
      resource:
        name: cpu 9
        target:
          type: AverageValue 10
          averageValue: 500m 11
  ```

  **1**    Use the **autoscaling/v2beta2** API.

  **2**    Specify a name for this horizontal pod autoscaler object.

  **3**    Specify the API version of the object to scale:

  - For a **Deployment**, **ReplicaSet**, **Statefulset** object, use **apps/v1**.

  - For a **ReplicationController**, use **v1**.

  - For a **DeploymentConfig**, use **apps.openshift.io/v1**.

  **4**    Specify the type of object. The object must be a **Deployment**, **DeploymentConfig/dc**, **ReplicaSet/rs**, **ReplicationController/rc**, or **StatefulSet**.

  **5**    Specify the name of the object to scale. The object must exist.

  **6**    Specify the minimum number of replicas when scaling down.

  **7**    Specify the maximum number of replicas when scaling up.

  **8**    Use the **metrics** parameter for memory utilization.

  **9**    Specify **cpu** for CPU utilization.

  **10**    Set to **AverageValue**.

  **11**    Set to **averageValue** with the targeted CPU value.

  b. Create the horizontal pod autoscaler:

```
$ oc create -f <file-name>.yaml
```

2. Verify that the horizontal pod autoscaler was created:

```
$ oc get hpa cpu-autoscale
```

**Example output**

```
NAME           REFERENCE           TARGETS       MINPODS  MAXPODS  REPLICAS
AGE
cpu-autoscale  Deployment/example  173m/500m     1        10       1         20m
```

### 2.4.4. Creating a horizontal pod autoscaler object for memory utilization by using the CLI

Using the OpenShift Container Platform CLI, you can create a horizontal pod autoscaler (HPA) to automatically scale an existing **Deployment**, **DeploymentConfig**, **ReplicaSet**, **ReplicationController**, or **StatefulSet** object. The HPA scales the pods associated with that object to maintain the average memory utilization you specify, either a direct value or a percentage of requested memory.

> **NOTE**
>
> It is recommended to use a **Deployment** object or **ReplicaSet** object unless you need a specific feature or behavior provided by other objects.

The HPA increases and decreases the number of replicas between the minimum and maximum numbers to maintain the specified memory utilization across all pods.

For memory utilization, you can specify the minimum and maximum number of pods and the average memory utilization your pods should target. If you do not specify a minimum, the pods are given default values from the OpenShift Container Platform server.

**Prerequisites**

To use horizontal pod autoscalers, your cluster administrator must have properly configured cluster metrics. You can use the **oc describe PodMetrics <pod-name>** command to determine if metrics are configured. If metrics are configured, the output appears similar to the following, with **Cpu** and **Memory** displayed under **Usage**.

```
$ oc describe PodMetrics openshift-kube-scheduler-ip-10-0-129-223.compute.internal -n openshift-kube-scheduler
```

**Example output**

```
Name:         openshift-kube-scheduler-ip-10-0-129-223.compute.internal
Namespace:    openshift-kube-scheduler
Labels:       <none>
Annotations:  <none>
API Version:  metrics.k8s.io/v1beta1
Containers:
  Name:  wait-for-host-port
  Usage:
    Cpu:    0
```

```
   Memory:  0
  Name:      scheduler
  Usage:
    Cpu:      8m
    Memory:  45440Ki
Kind:        PodMetrics
Metadata:
  Creation Timestamp:  2020-02-14T22:21:14Z
  Self Link:           /apis/metrics.k8s.io/v1beta1/namespaces/openshift-kube-scheduler/pods/openshift-
kube-scheduler-ip-10-0-129-223.compute.internal
  Timestamp:           2020-02-14T22:21:14Z
  Window:              5m0s
Events:                <none>
```

## Procedure

To create a horizontal pod autoscaler for memory utilization:

1. Create a YAML file for one of the following:

   - To scale for a specific memory value, create a **HorizontalPodAutoscaler** object similar to the following for an existing object:

```
apiVersion: autoscaling/v2beta2 1
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-resource-metrics-memory 2
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1 3
    kind: Deployment 4
    name: example 5
  minReplicas: 1 6
  maxReplicas: 10 7
  metrics: 8
  - type: Resource
    resource:
      name: memory 9
      target:
        type: AverageValue 10
        averageValue: 500Mi 11
  behavior: 12
    scaleDown:
      stabilizationWindowSeconds: 300
      policies:
      - type: Pods
        value: 4
        periodSeconds: 60
      - type: Percent
        value: 10
        periodSeconds: 60
      selectPolicy: Max
```

**1**    Use the **autoscaling/v2beta2** API.

**2**    Specify a name for this horizontal pod autoscaler object.

**3**    Specify the API version of the object to scale:

- For a **Deployment**, **ReplicaSet**, or **Statefulset** object, use **apps/v1**.

- For a **ReplicationController**, use **v1**.

- For a **DeploymentConfig**, use **apps.openshift.io/v1**.

**4**    Specify the type of object. The object must be a **Deployment**, **DeploymentConfig**, **ReplicaSet**, **ReplicationController**, or **StatefulSet**.

**5**    Specify the name of the object to scale. The object must exist.

**6**    Specify the minimum number of replicas when scaling down.

**7**    Specify the maximum number of replicas when scaling up.

**8**    Use the **metrics** parameter for memory utilization.

**9**    Specify **memory** for memory utilization.

**10**    Set the type to **AverageValue**.

**11**    Specify **averageValue** and a specific memory value.

**12**    Optional: Specify a scaling policy to control the rate of scaling up or down.

- To scale for a percentage, create a **HorizontalPodAutoscaler** object similar to the following for an existing object:

```
apiVersion: autoscaling/v2beta2 1
kind: HorizontalPodAutoscaler
metadata:
  name: memory-autoscale 2
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1 3
    kind: Deployment 4
    name: example 5
  minReplicas: 1 6
  maxReplicas: 10 7
  metrics: 8
  - type: Deployment
    resource:
      name: memory 9
      target:
        type: Utilization 10
        averageUtilization: 50 11
  behavior: 12
```

```
      scaleUp:
        stabilizationWindowSeconds: 180
        policies:
        - type: Pods
          value: 6
          periodSeconds: 120
        - type: Percent
          value: 10
          periodSeconds: 120
        selectPolicy: Max
```

**1**    Use the **autoscaling/v2beta2** API.

**2**    Specify a name for this horizontal pod autoscaler object.

**3**    Specify the API version of the object to scale:

- For a ReplicationController, use **v1**.

- For a DeploymentConfig, use **apps.openshift.io/v1**.

- For a Deployment, ReplicaSet, Statefulset object, use **apps/v1**.

**4**    Specify the type of object. The object must be a **Deployment**, **DeploymentConfig**, **ReplicaSet**, **ReplicationController**, or **StatefulSet**.

**5**    Specify the name of the object to scale. The object must exist.

**6**    Specify the minimum number of replicas when scaling down.

**7**    Specify the maximum number of replicas when scaling up.

**8**    Use the **metrics** parameter for memory utilization.

**9**    Specify **memory** for memory utilization.

**10**    Set to **Utilization**.

**11**    Specify **averageUtilization** and a target average memory utilization over all the pods, represented as a percent of requested memory. The target pods must have memory requests configured.

**12**    Optional: Specify a scaling policy to control the rate of scaling up or down.

2. Create the horizontal pod autoscaler:

```
$ oc create -f <file-name>.yaml
```

For example:

```
$ oc create -f hpa.yaml
```

**Example output**

```
horizontalpodautoscaler.autoscaling/hpa-resource-metrics-memory created
```

3. Verify that the horizontal pod autoscaler was created:

```
$ oc get hpa hpa-resource-metrics-memory
```

**Example output**

```
NAME                      REFERENCE          TARGETS       MINPODS  MAXPODS
REPLICAS  AGE
hpa-resource-metrics-memory  Deployment/example  2441216/500Mi  1      10      1
20m
```

```
$ oc describe hpa hpa-resource-metrics-memory
```

**Example output**

```
Name:                 hpa-resource-metrics-memory
Namespace:            default
Labels:               <none>
Annotations:          <none>
CreationTimestamp:       Wed, 04 Mar 2020 16:31:37 +0530
Reference:            Deployment/example
Metrics:              ( current / target )
  resource memory on pods:  2441216 / 500Mi
Min replicas:          1
Max replicas:          10
ReplicationController pods:  1 current / 1 desired
Conditions:
  Type          Status  Reason          Message
  ----          ------  ------          -------
  AbleToScale    True    ReadyForNewScale  recommended size matches current size
  ScalingActive  True    ValidMetricFound   the HPA was able to successfully calculate a
replica count from memory resource
  ScalingLimited  False  DesiredWithinRange  the desired count is within the acceptable
range
Events:
  Type    Reason           Age        From                Message
  ----    ------           ----       ----                -------
  Normal  SuccessfulRescale    6m34s           horizontal-pod-autoscaler  New size: 1;
reason: All metrics below target
```

## 2.4.5. Understanding horizontal pod autoscaler status conditions by using the CLI

You can use the status conditions set to determine whether or not the horizontal pod autoscaler (HPA) is able to scale and whether or not it is currently restricted in any way.

The HPA status conditions are available with the **v2beta1** version of the autoscaling API.

The HPA responds with the following status conditions:

- The **AbleToScale** condition indicates whether HPA is able to fetch and update metrics, as well as whether any backoff–related conditions could prevent scaling.
  - A **True** condition indicates scaling is allowed.

- A **False** condition indicates scaling is not allowed for the reason specified.

- The **ScalingActive** condition indicates whether the HPA is enabled (for example, the replica count of the target is not zero) and is able to calculate desired metrics.

  - A **True** condition indicates metrics is working properly.

  - A **False** condition generally indicates a problem with fetching metrics.

- The **ScalingLimited** condition indicates that the desired scale was capped by the maximum or minimum of the horizontal pod autoscaler.

  - A **True** condition indicates that you need to raise or lower the minimum or maximum replica count in order to scale.

  - A **False** condition indicates that the requested scaling is allowed.

```
$ oc describe hpa cm-test
```

**Example output**

```
Name:                cm-test
Namespace:             prom
Labels:            <none>
Annotations:            <none>
CreationTimestamp:        Fri, 16 Jun 2017 18:09:22 +0000
Reference:                ReplicationController/cm-test
Metrics:            ( current / target )
  "http_requests" on pods:     66m / 500m
Min replicas:            1
Max replicas:            4
ReplicationController pods:    1 current / 1 desired
Conditions: ❶
  Type            Status    Reason          Message
  ----            ------    ------          -------
  AbleToScale     True      ReadyForNewScale    the last scale time was sufficiently old
as to warrant a new scale
  ScalingActive    True     ValidMetricFound    the HPA was able to successfully
calculate a replica count from pods metric http_request
  ScalingLimited    False    DesiredWithinRange  the desired replica count is within the
acceptable range
Events:
```

❶ The horizontal pod autoscaler status messages.

The following is an example of a pod that is unable to scale:

**Example output**

```
Conditions:
  Type        Status  Reason          Message
  ----        ------  ------          -------
  AbleToScale  False   FailedGetScale  the HPA controller was unable to get the target's current
scale: no matches for kind "ReplicationController" in group "apps"
```

```
Events:
 Type    Reason        Age            From                Message
 ----    ------        ----           ----                -------
 Warning  FailedGetScale  6s (x3 over 36s)  horizontal-pod-autoscaler  no matches for kind
"ReplicationController" in group "apps"
```

The following is an example of a pod that could not obtain the needed metrics for scaling:

**Example output**

```
Conditions:
 Type            Status   Reason              Message
 ----            ------   ------              -------
 AbleToScale       True    SucceededGetScale     the HPA controller was able to get the target's
current scale
 ScalingActive      False   FailedGetResourceMetric   the HPA was unable to compute the replica
count: failed to get cpu utilization: unable to get metrics for resource cpu: no metrics returned from
resource metrics API
```

The following is an example of a pod where the requested autoscaling was less than the required minimums:

**Example output**

```
Conditions:
 Type            Status   Reason            Message
 ----            ------   ------            -------
 AbleToScale       True    ReadyForNewScale     the last scale time was sufficiently old as to warrant
a new scale
 ScalingActive     True    ValidMetricFound     the HPA was able to successfully calculate a replica
count from pods metric http_request
 ScalingLimited    False    DesiredWithinRange   the desired replica count is within the acceptable
range
```

### 2.4.5.1. Viewing horizontal pod autoscaler status conditions by using the CLI

You can view the status conditions set on a pod by the horizontal pod autoscaler (HPA).

**NOTE**

The horizontal pod autoscaler status conditions are available with the **v2beta1** version of the autoscaling API.

#### Prerequisites

To use horizontal pod autoscalers, your cluster administrator must have properly configured cluster metrics. You can use the **oc describe PodMetrics <pod-name>** command to determine if metrics are configured. If metrics are configured, the output appears similar to the following, with **Cpu** and **Memory** displayed under **Usage**.

```
$ oc describe PodMetrics openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
```

**Example output**

```
Name:        openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
Namespace:   openshift-kube-scheduler
Labels:      <none>
Annotations: <none>
API Version: metrics.k8s.io/v1beta1
Containers:
  Name:  wait-for-host-port
  Usage:
    Memory:  0
  Name:      scheduler
  Usage:
    Cpu:     8m
    Memory:  45440Ki
Kind:        PodMetrics
Metadata:
  Creation Timestamp:  2019-05-23T18:47:56Z
  Self Link:           /apis/metrics.k8s.io/v1beta1/namespaces/openshift-kube-scheduler/pods/openshift-
kube-scheduler-ip-10-0-135-131.ec2.internal
Timestamp:           2019-05-23T18:47:56Z
Window:              1m0s
Events:              <none>
```

## Procedure

To view the status conditions on a pod, use the following command with the name of the pod:

```
$ oc describe hpa <pod-name>
```

For example:

```
$ oc describe hpa cm-test
```

The conditions appear in the **Conditions** field in the output.

## Example output

```
Name:                  cm-test
Namespace:             prom
Labels:                <none>
Annotations:           <none>
CreationTimestamp:     Fri, 16 Jun 2017 18:09:22 +0000
Reference:             ReplicationController/cm-test
Metrics:               ( current / target )
  "http_requests" on pods:    66m / 500m
Min replicas:          1
Max replicas:          4
ReplicationController pods:    1 current / 1 desired
Conditions:          1
  Type           Status   Reason           Message
  ----           ------   ------           -------
  AbleToScale    True     ReadyForNewScale    the last scale time was sufficiently old as to warrant
a new scale
  ScalingActive  True     ValidMetricFound    the HPA was able to successfully calculate a replica
```

> count from pods metric http_request
>   ScalingLimited    False    DesiredWithinRange  the desired replica count is within the acceptable
> range

## 2.4.6. Additional resources

- For more information on replication controllers and deployment controllers, see Understanding deployments and deployment configs.

# 2.5. AUTOMATICALLY ADJUST POD RESOURCE LEVELS WITH THE VERTICAL POD AUTOSCALER

The OpenShift Container Platform Vertical Pod Autoscaler Operator (VPA) automatically reviews the historic and current CPU and memory resources for containers in pods and can update the resource limits and requests based on the usage values it learns. The VPA uses individual custom resources (CR) to update all of the pods associated with a workload object, such as a **Deployment**, **DeploymentConfig**, **StatefulSet**, **Job**, **DaemonSet**, **ReplicaSet**, or **ReplicationController**, in a project.

The VPA helps you to understand the optimal CPU and memory usage for your pods and can automatically maintain pod resources through the pod lifecycle.

## 2.5.1. About the Vertical Pod Autoscaler Operator

The Vertical Pod Autoscaler Operator (VPA) is implemented as an API resource and a custom resource (CR). The CR determines the actions the Vertical Pod Autoscaler Operator should take with the pods associated with a specific workload object, such as a daemon set, replication controller, and so forth, in a project.

The VPA automatically computes historic and current CPU and memory usage for the containers in those pods and uses this data to determine optimized resource limits and requests to ensure that these pods are operating efficiently at all times. For example, the VPA reduces resources for pods that are requesting more resources than they are using and increases resources for pods that are not requesting enough.

The VPA automatically deletes any pods that are out of alignment with its recommendations one at a time, so that your applications can continue to serve requests with no downtime. The workload objects then re-deploy the pods with the original resource limits and requests. The VPA uses a mutating admission webhook to update the pods with optimized resource limits and requests before the pods are admitted to a node. If you do not want the VPA to delete pods, you can view the VPA resource limits and requests and manually update the pods as needed.

> **NOTE**
>
> By default, workload objects must specify a minimum of two replicas in order for the VPA to automatically delete their pods. Workload objects that specify fewer replicas than this minimum are not deleted. If you manually delete these pods, when the workload object redeploys the pods, the VPA does update the new pods with its recommendations. You can change this minimum by modifying the **VerticalPodAutoscalerController** object as shown shown in *Changing the VPA minimum value*.

For example, if you have a pod that uses 50% of the CPU but only requests 10%, the VPA determines that the pod is consuming more CPU than requested and deletes the pod. The workload object, such as replica set, restarts the pods and the VPA updates the new pod with its recommended resources.

For developers, you can use the VPA to help ensure your pods stay up during periods of high demand by scheduling pods onto nodes that have appropriate resources for each pod.

Administrators can use the VPA to better utilize cluster resources, such as preventing pods from reserving more CPU resources than needed. The VPA monitors the resources that workloads are actually using and adjusts the resource requirements so capacity is available to other workloads. The VPA also maintains the ratios between limits and requests that are specified in initial container configuration.

> **NOTE**
>
> If you stop running the VPA or delete a specific VPA CR in your cluster, the resource requests for the pods already modified by the VPA do not change. Any new pods get the resources defined in the workload object, not the previous recommendations made by the VPA.

## 2.5.2. Installing the Vertical Pod Autoscaler Operator

You can use the OpenShift Container Platform web console to install the Vertical Pod Autoscaler Operator (VPA).

Procedure

1. In the OpenShift Container Platform web console, click **Operators** → **OperatorHub**.

2. Choose **VerticalPodAutoscaler** from the list of available Operators, and click **Install**.

3. On the **Install Operator** page, ensure that the **Operator recommended namespace** option is selected. This installs the Operator in the mandatory **openshift-vertical-pod-autoscaler** namespace, which is automatically created if it does not exist.

4. Click **Install**.

5. Verify the installation by listing the VPA Operator components:

   a. Navigate to **Workloads** → **Pods**.

   b. Select the **openshift-vertical-pod-autoscaler** project from the drop-down menu and verify that there are four pods running.

   c. Navigate to **Workloads** → **Deployments** to verify that there are four deployments running.

6. Optional. Verify the installation in the OpenShift Container Platform CLI using the following command:

   ```
   $ oc get all -n openshift-vertical-pod-autoscaler
   ```

   The output shows four pods and four deployments:

   Example output

   ```
   NAME                                                 READY   STATUS    RESTARTS   AGE
   pod/vertical-pod-autoscaler-operator-85b4569c47-2gmhc  1/1     Running   0          3m13s
   pod/vpa-admission-plugin-default-67644fc87f-xq7k9       1/1     Running   0          2m56s
   pod/vpa-recommender-default-7c54764b59-8gckt            1/1     Running   0          2m56s
   ```

```
pod/vpa-updater-default-7f6cc87858-47vw9                    1/1    Running  0        2m56s

NAME                TYPE        CLUSTER-IP     EXTERNAL-IP  PORT(S)   AGE
service/vpa-webhook  ClusterIP   172.30.53.206  <none>       443/TCP   2m56s

NAME                                        READY  UP-TO-DATE  AVAILABLE  AGE
deployment.apps/vertical-pod-autoscaler-operator  1/1    1           1          3m13s
deployment.apps/vpa-admission-plugin-default       1/1    1           1          2m56s
deployment.apps/vpa-recommender-default            1/1    1           1          2m56s
deployment.apps/vpa-updater-default                1/1    1           1          2m56s

NAME                                              DESIRED  CURRENT  READY  AGE
replicaset.apps/vertical-pod-autoscaler-operator-85b4569c47  1      1        1      3m13s
replicaset.apps/vpa-admission-plugin-default-67644fc87f      1      1        1      2m56s
replicaset.apps/vpa-recommender-default-7c54764b59          1      1        1      2m56s
replicaset.apps/vpa-updater-default-7f6cc87858             1      1        1      2m56s
```

### 2.5.3. About Using the Vertical Pod Autoscaler Operator

To use the Vertical Pod Autoscaler Operator (VPA), you create a VPA custom resource (CR) for a workload object in your cluster. The VPA learns and applies the optimal CPU and memory resources for the pods associated with that workload object. You can use a VPA with a deployment, stateful set, job, daemon set, replica set, or replication controller workload object. The VPA CR must be in the same project as the pods you want to monitor.

You use the VPA CR to associate a workload object and specify which mode the VPA operates in:

- The **Auto** and **Recreate** modes automatically apply the VPA CPU and memory recommendations throughout the pod lifetime. The VPA deletes any pods in the project that are out of alignment with its recommendations. When redeployed by the workload object, the VPA updates the new pods with its recommendations.

- The **Initial** mode automatically applies VPA recommendations only at pod creation.

- The **Off** mode only provides recommended resource limits and requests, allowing you to manually apply the recommendations. The **off** mode does not update pods.

You can also use the CR to opt-out certain containers from VPA evaluation and updates.

For example, a pod has the following limits and requests:

```
resources:
  limits:
    cpu: 1
    memory: 500Mi
  requests:
    cpu: 500m
    memory: 100Mi
```

After creating a VPA that is set to **auto**, the VPA learns the resource usage and deletes the pod. When redeployed, the pod uses the new resource limits and requests:

```
resources:
  limits:
    cpu: 50m
```

```
      memory: 1250Mi
    requests:
      cpu: 25m
      memory: 262144k
```

You can view the VPA recommendations using the following command:

```
$ oc get vpa <vpa-name> --output yaml
```

After a few minutes, the output shows the recommendations for CPU and memory requests, similar to the following:

### Example output

```
...
status:
...
  recommendation:
    containerRecommendations:
    - containerName: frontend
      lowerBound:
        cpu: 25m
        memory: 262144k
      target:
        cpu: 25m
        memory: 262144k
      uncappedTarget:
        cpu: 25m
        memory: 262144k
      upperBound:
        cpu: 262m
        memory: "274357142"
    - containerName: backend
      lowerBound:
        cpu: 12m
        memory: 131072k
      target:
        cpu: 12m
        memory: 131072k
      uncappedTarget:
        cpu: 12m
        memory: 131072k
      upperBound:
        cpu: 476m
        memory: "498558823"
...
```

The output shows the recommended resources, **target**, the minimum recommended resources, **lowerBound**, the highest recommended resources, **upperBound**, and the most recent resource recommendations, **uncappedTarget**.

The VPA uses the **lowerBound** and **upperBound** values to determine if a pod needs to be updated. If a pod has resource requests below the **lowerBound** values or above the **upperBound** values, the VPA terminates and recreates the pod with the **target** values.

### 2.5.3.1. Changing the VPA minimum value

By default, workload objects must specify a minimum of two replicas in order for the VPA to automatically delete and update their pods. As a result, workload objects that specify fewer than two replicas are not automatically acted upon by the VPA. The VPA does update new pods from these workload objects if the pods are restarted by some process external to the VPA. You can change this cluster-wide minimum value by modifying the **minReplicas** parameter in the **VerticalPodAutoscalerController** custom resource (CR).

For example, if you set **minReplicas** to **3**, the VPA does not delete and update pods for workload objects that specify fewer than three replicas.

> **NOTE**
>
> If you set **minReplicas** to **1**, the VPA can delete the only pod for a workload object that specifies only one replica. You should use this setting with one-replica objects only if your workload can tolerate downtime whenever the VPA deletes a pod to adjust its resources. To avoid unwanted downtime with one-replica objects, configure the VPA CRs with the **podUpdatePolicy** set to **Initial**, which automatically updates the pod only when it is restarted by some process external to the VPA, or **Off**, which allows you to update the pod manually at an appropriate time for your application.

Example **VerticalPodAutoscalerController** object

```
apiVersion: autoscaling.openshift.io/v1
kind: VerticalPodAutoscalerController
metadata:
  creationTimestamp: "2021-04-21T19:29:49Z"
  generation: 2
  name: default
  namespace: openshift-vertical-pod-autoscaler
  resourceVersion: "142172"
  uid: 180e17e9-03cc-427f-9955-3b4d7aeb2d59
spec:
  minReplicas: 3 ❶
  podMinCPUMillicores: 25
  podMinMemoryMb: 250
  recommendationOnly: false
  safetyMarginFraction: 0.15
```

❶ Specify the minimum number of replicas in a workload object for the VPA to act on. Any objects with replicas fewer than the minimum are not automatically deleted by the VPA.

### 2.5.3.2. Automatically applying VPA recommendations

To use the VPA to automatically update pods, create a VPA CR for a specific workload object with **updateMode** set to **Auto** or **Recreate**.

When the pods are created for the workload object, the VPA constantly monitors the containers to analyze their CPU and memory needs. The VPA deletes any pods that do not meet the VPA recommendations for CPU and memory. When redeployed, the pods use the new resource limits and requests based on the VPA recommendations, honoring any pod disruption budget set for your applications. The recommendations are added to the **status** field of the VPA CR for reference.

> **NOTE**
>
> By default, workload objects must specify a minimum of two replicas in order for the VPA to automatically delete their pods. Workload objects that specify fewer replicas than this minimum are not deleted. If you manually delete these pods, when the workload object redeploys the pods, the VPA does update the new pods with its recommendations. You can change this minimum by modifying the **VerticalPodAutoscalerController** object as shown shown in *Changing the VPA minimum value*.

### Example VPA CR for the Auto mode

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:      Deployment 1
    name:      frontend 2
  updatePolicy:
    updateMode: "Auto" 3
```

**1** The type of workload object you want this VPA CR to manage.

**2** The name of the workload object you want this VPA CR to manage.

**3** Set the mode to **Auto** or **Recreate**:

- **Auto**. The VPA assigns resource requests on pod creation and updates the existing pods by terminating them when the requested resources differ significantly from the new recommendation.

- **Recreate**. The VPA assigns resource requests on pod creation and updates the existing pods by terminating them when the requested resources differ significantly from the new recommendation. This mode should be used rarely, only if you need to ensure that the pods are restarted whenever the resource request changes.

> **NOTE**
>
> There must be operating pods in the project before the VPA can determine recommended resources and apply the recommendations to new pods.

### 2.5.3.3. Automatically applying VPA recommendations on pod creation

To use the VPA to apply the recommended resources only when a pod is first deployed, create a VPA CR for a specific workload object with **updateMode** set to **Initial**.

Then, manually delete any pods associated with the workload object that you want to use the VPA recommendations. In the **Initial** mode, the VPA does not delete pods and does not update the pods as it learns new resource recommendations.

### Example VPA CR for the Initial mode

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
spec:
 targetRef:
   apiVersion: "apps/v1"
   kind:      Deployment 1
   name:      frontend 2
 updatePolicy:
   updateMode: "Initial" 3
```

**1**    The type of workload object you want this VPA CR to manage.

**2**    The name of the workload object you want this VPA CR to manage.

**3**    Set the mode to **Initial**. The VPA assigns resources when pods are created and does not change the resources during the lifetime of the pod.
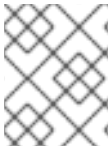
> **NOTE**
>
> There must be operating pods in the project before a VPA can determine recommended resources and apply the recommendations to new pods.

### 2.5.3.4. Manually applying VPA recommendations

To use the VPA to only determine the recommended CPU and memory values, create a VPA CR for a specific workload object with **updateMode** set to **off**.

When the pods are created for that workload object, the VPA analyzes the CPU and memory needs of the containers and records those recommendations in the **status** field of the VPA CR. The VPA does not update the pods as it determines new resource recommendations.

### Example VPA CR for the Off mode

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
spec:
 targetRef:
   apiVersion: "apps/v1"
   kind:      Deployment 1
   name:      frontend 2
 updatePolicy:
   updateMode: "Off" 3
```

**1**    The type of workload object you want this VPA CR to manage.

**2**    The name of the workload object you want this VPA CR to manage.

**3**    Set the mode to **Off**.

You can view the recommendations using the following command.

```
$ oc get vpa <vpa-name> --output yaml
```

With the recommendations, you can edit the workload object to add CPU and memory requests, then delete and redeploy the pods using the recommended resources.

> **NOTE**
>
> There must be operating pods in the project before a VPA can determine recommended resources.

### 2.5.3.5. Exempting containers from applying VPA recommendations

If your workload object has multiple containers and you do not want the VPA to evaluate and act on all of the containers, create a VPA CR for a specific workload object and add a **resourcePolicy** to opt-out specific containers.

When the VPA updates the pods with recommended resources, any containers with a **resourcePolicy** are not updated and the VPA does not present recommendations for those containers in the pod.

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:      Deployment 1
    name:      frontend 2
  updatePolicy:
    updateMode: "Auto" 3
  resourcePolicy: 4
    containerPolicies:
    - containerName: my-opt-sidecar
      mode: "Off"
```

**1**    The type of workload object you want this VPA CR to manage.

**2**    The name of the workload object you want this VPA CR to manage.

**3**    Set the mode to **Auto**, **Recreate**, or **Off**. The **Recreate** mode should be used rarely, only if you need to ensure that the pods are restarted whenever the resource request changes.

**4**    Specify the containers you want to opt-out and set **mode** to **Off**.

For example, a pod has two containers, the same resource requests and limits:

```
# ...
spec:
  containers:
  - name: frontend
    resources:
```

```
      limits:
        cpu: 1
        memory: 500Mi
      requests:
        cpu: 500m
        memory: 100Mi
  - name: backend
    resources:
      limits:
        cpu: "1"
        memory: 500Mi
      requests:
        cpu: 500m
        memory: 100Mi
# ...
```

After launching a VPA CR with the **backend** container set to opt-out, the VPA terminates and recreates the pod with the recommended resources applied only to the **frontend** container:

```
...
spec:
  containers:
    name: frontend
    resources:
      limits:
        cpu: 50m
        memory: 1250Mi
      requests:
        cpu: 25m
        memory: 262144k
...
    name: backend
    resources:
      limits:
        cpu: "1"
        memory: 500Mi
      requests:
        cpu: 500m
        memory: 100Mi
...
```

## 2.5.4. Using the Vertical Pod Autoscaler Operator

You can use the Vertical Pod Autoscaler Operator (VPA) by creating a VPA custom resource (CR). The CR indicates which pods it should analyze and determines the actions the VPA should take with those pods.

### Procedure

To create a VPA CR for a specific workload object:

1. Change to the project where the workload object you want to scale is located.

   a. Create a VPA CR YAML file:

   ```
   apiVersion: autoscaling.k8s.io/v1
   ```

```
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:      Deployment 1
    name:      frontend 2
  updatePolicy:
    updateMode: "Auto" 3
  resourcePolicy: 4
    containerPolicies:
    - containerName: my-opt-sidecar
      mode: "Off"
```

**1** Specify the type of workload object you want this VPA to manage: **Deployment**, **StatefulSet**, **Job**, **DaemonSet**, **ReplicaSet**, or **ReplicationController**.

**2** Specify the name of an existing workload object you want this VPA to manage.

**3** Specify the VPA mode:

- **auto** to automatically apply the recommended resources on pods associated with the controller. The VPA terminates existing pods and creates new pods with the recommended resource limits and requests.

- **recreate** to automatically apply the recommended resources on pods associated with the workload object. The VPA terminates existing pods and creates new pods with the recommended resource limits and requests. The **recreate** mode should be used rarely, only if you need to ensure that the pods are restarted whenever the resource request changes.

- **initial** to automatically apply the recommended resources when pods associated with the workload object are created. The VPA does not update the pods as it learns new resource recommendations.

- **off** to only generate resource recommendations for the pods associated with the workload object. The VPA does not update the pods as it learns new resource recommendations and does not apply the recommendations to new pods.

**4** Optional. Specify the containers you want to opt–out and set the mode to **Off**.

b. Create the VPA CR:

```
$ oc create -f <file-name>.yaml
```

After a few moments, the VPA learns the resource usage of the containers in the pods associated with the workload object.

You can view the VPA recommendations using the following command:

```
$ oc get vpa <vpa-name> --output yaml
```

The output shows the recommendations for CPU and memory requests, similar to the following:

**Example output**

```
...
status:

...

  recommendation:
    containerRecommendations:
    - containerName: frontend
      lowerBound: 1
        cpu: 25m
        memory: 262144k
      target: 2
        cpu: 25m
        memory: 262144k
      uncappedTarget: 3
        cpu: 25m
        memory: 262144k
      upperBound: 4
        cpu: 262m
        memory: "274357142"
    - containerName: backend
      lowerBound:
        cpu: 12m
        memory: 131072k
      target:
        cpu: 12m
        memory: 131072k
      uncappedTarget:
        cpu: 12m
        memory: 131072k
      upperBound:
        cpu: 476m
        memory: "498558823"

...
```

[1] **lowerBound** is the minimum recommended resource levels.

[2] **target** is the recommended resource levels.

[3] **upperBound** is the highest recommended resource levels.

[4] **uncappedTarget** is the most recent resource recommendations.

## 2.5.5. Uninstalling the Vertical Pod Autoscaler Operator

You can remove the Vertical Pod Autoscaler Operator (VPA) from your OpenShift Container Platform cluster. After uninstalling, the resource requests for the pods already modified by an existing VPA CR do not change. Any new pods get the resources defined in the workload object, not the previous

recommendations made by the Vertical Pod Autoscaler Operator.
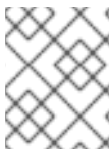
> **NOTE**
>
> You can remove a specific VPA CR by using the **oc delete vpa <vpa-name>** command. The same actions apply for resource requests as uninstalling the vertical pod autoscaler.

After removing the VPA Operator, it is recommended that you remove the other components associated with the Operator to avoid potential issues.

**Prerequisites**

- The Vertical Pod Autoscaler Operator must be installed.

**Procedure**

1. In the OpenShift Container Platform web console, click **Operators → Installed Operators**.

2. Switch to the **openshift-vertical-pod-autoscaler** project.

3. For the **VerticalPodAutoscaler** Operator, click the Options menu       and select **Uninstall Operator**.

4. In the dialog box, click **Uninstall**.

5. Optional: To remove all operands associated with the Operator, in the dialog box, select **Delete all operand instances for this operator** checkbox.

6. Click **Uninstall**.

7. Optional: Use the OpenShift CLI to remove the VPA components:

   a. Delete the VPA namespace:

      ```
      $ oc delete namespace openshift-vertical-pod-autoscaler
      ```

   b. Delete the VPA custom resource definition (CRD) objects:

      ```
      $ oc delete crd verticalpodautoscalercheckpoints.autoscaling.k8s.io
      ```

      ```
      $ oc delete crd verticalpodautoscalercontrollers.autoscaling.openshift.io
      ```

      ```
      $ oc delete crd verticalpodautoscalers.autoscaling.k8s.io
      ```

      Deleting the CRDs removes the associated roles, cluster roles, and role bindings.

      > **NOTE**
      >
      > This action removes from the cluster all user-created VPA CRs. If you re-install the VPA, you must create these objects again.

c.  Delete the VPA Operator:

```
$ oc delete operator/vertical-pod-autoscaler.openshift-vertical-pod-autoscaler
```

## 2.6. PROVIDING SENSITIVE DATA TO PODS

Some applications need sensitive information, such as passwords and user names, that you do not want developers to have.

As an administrator, you can use **Secret** objects to provide this information without exposing that information in clear text.

### 2.6.1. Understanding secrets

The **Secret** object type provides a mechanism to hold sensitive information such as passwords, OpenShift Container Platform client configuration files, private source repository credentials, and so on. Secrets decouple sensitive content from the pods. You can mount secrets into containers using a volume plugin or the system can use secrets to perform actions on behalf of a pod.

Key properties include:

- Secret data can be referenced independently from its definition.

- Secret data volumes are backed by temporary file-storage facilities (tmpfs) and never come to rest on a node.

- Secret data can be shared within a namespace.

### YAML **Secret** object definition

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
  namespace: my-namespace
type: Opaque 1
data: 2
  username: dmFsdWUtMQ0K 3
  password: dmFsdWUtMg0KDQo=
stringData: 4
  hostname: myapp.mydomain.com 5
```

**1**  Indicates the structure of the secret's key names and values.

**2**  The allowable format for the keys in the **data** field must meet the guidelines in the **DNS_SUBDOMAIN** value in the Kubernetes identifiers glossary.

**3**  The value associated with keys in the **data** map must be base64 encoded.

**4**  Entries in the **stringData** map are converted to base64 and the entry will then be moved to the **data** map automatically. This field is write-only; the value will only be returned via the **data** field.

**5**  The value associated with keys in the **stringData** map is made up of plain text strings.

You must create a secret before creating the pods that depend on that secret.

When creating secrets:

- Create a secret object with secret data.

- Update the pod's service account to allow the reference to the secret.

- Create a pod, which consumes the secret as an environment variable or as a file (using a **secret** volume).

### 2.6.1.1. Types of secrets

The value in the **type** field indicates the structure of the secret's key names and values. The type can be used to enforce the presence of user names and keys in the secret object. If you do not want validation, use the **opaque** type, which is the default.

Specify one of the following types to trigger minimal server-side validation to ensure the presence of specific key names in the secret data:

- **kubernetes.io/service-account-token**. Uses a service account token.

- **kubernetes.io/basic-auth**. Use with Basic Authentication.

- **kubernetes.io/ssh-auth**. Use with SSH Key Authentication.

- **kubernetes.io/tls**. Use with TLS certificate authorities.

Specify **type: Opaque** if you do not want validation, which means the secret does not claim to conform to any convention for key names or values. An *opaque* secret, allows for unstructured **key:value** pairs that can contain arbitrary values.

> **NOTE**
>
> You can specify other arbitrary types, such as **example.com/my-secret-type**. These types are not enforced server-side, but indicate that the creator of the secret intended to conform to the key/value requirements of that type.

For examples of different secret types, see the code samples in *Using Secrets*.

### 2.6.1.2. Secret data keys

Secret keys must be in a DNS subdomain.

### 2.6.2. Understanding how to create secrets

As an administrator you must create a secret before developers can create the pods that depend on that secret.

When creating secrets:

1. Create a secret object that contains the data you want to keep secret. The specific data required for each secret type is descibed in the following sections.

   **Example YAML object that creates an opaque secret**

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
type: Opaque 1
data: 2
  username: dmFsdWUtMQ0K
  password: dmFsdWUtMQ0KDQo=
stringData: 3
  hostname: myapp.mydomain.com
  secret.properties: |
    property1=valueA
    property2=valueB
```

**1** Specifies the type of secret.

**2** Specifies encoded string and data.

**3** Specifies decoded string and data.

Use either the **data** or **stringdata** fields, not both.

2. Update the pod's service account to reference the secret:

   **YAML of a service account that uses a secret**

   ```
   apiVersion: v1
   kind: ServiceAccount
    ...
   secrets:
   - name: test-secret
   ```

3. Create a pod, which consumes the secret as an environment variable or as a file (using a **secret** volume):

   **YAML of a pod populating files in a volume with secret data**

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: secret-example-pod
   spec:
     containers:
       - name: secret-test-container
         image: busybox
         command: [ "/bin/sh", "-c", "cat /etc/secret-volume/*" ]
         volumeMounts: 1
           - name: secret-volume
             mountPath: /etc/secret-volume 2
             readOnly: true 3
     volumes:
       - name: secret-volume
   ```

```
    secret:
      secretName: test-secret 4
  restartPolicy: Never
```

**1**   Add a **volumeMounts** field to each container that needs the secret.

**2**   Specifies an unused directory name where you would like the secret to appear. Each key in the secret data map becomes the filename under **mountPath**.

**3**   Set to **true**. If true, this instructs the driver to provide a read-only volume.

**4**   Specifies the name of the secret.

### YAML of a pod populating environment variables with secret data

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  containers:
    - name: secret-test-container
      image: busybox
      command: [ "/bin/sh", "-c", "export" ]
      env:
        - name: TEST_SECRET_USERNAME_ENV_VAR
          valueFrom:
            secretKeyRef: 1
              name: test-secret
              key: username
  restartPolicy: Never
```

**1**   Specifies the environment variable that consumes the secret key.

### YAML of a build config populating environment variables with secret data

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: secret-example-bc
spec:
  strategy:
    sourceStrategy:
      env:
        - name: TEST_SECRET_USERNAME_ENV_VAR
          valueFrom:
            secretKeyRef: 1
              name: test-secret
              key: username
```

**1**   Specifies the environment variable that consumes the secret key.

### 2.6.2.1. Secret creation restrictions

To use a secret, a pod needs to reference the secret. A secret can be used with a pod in three ways:

- To populate environment variables for containers.

- As files in a volume mounted on one or more of its containers.

- By kubelet when pulling images for the pod.

Volume type secrets write data into the container as a file using the volume mechanism. Image pull secrets use service accounts for the automatic injection of the secret into all pods in a namespace.

When a template contains a secret definition, the only way for the template to use the provided secret is to ensure that the secret volume sources are validated and that the specified object reference actually points to a **Secret** object. Therefore, a secret needs to be created before any pods that depend on it. The most effective way to ensure this is to have it get injected automatically through the use of a service account.

Secret API objects reside in a namespace. They can only be referenced by pods in that same namespace.

Individual secrets are limited to 1MB in size. This is to discourage the creation of large secrets that could exhaust apiserver and kubelet memory. However, creation of a number of smaller secrets could also exhaust memory.

### 2.6.2.2. Creating an opaque secret

As an administrator, you can create an opaque secret, which allows you to store unstructured **key:value** pairs that can contain arbitrary values.

**Procedure**

1. Create a **Secret** object in a YAML file on a control plane node.
   For example:

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: mysecret
   type: Opaque ❶
   data:
     username: dXNlci1uYW1l
     password: cGFzc3dvcmQ=
   ```

   ❶     Specifies an opaque secret.

2. Use the following command to create a **Secret** object:

   ```
   $ oc create -f <filename>.yaml
   ```

3. To use the secret in a pod:

   a. Update the pod's service account to reference the secret, as shown in the "Understanding how to create secrets" section.

b. Create the pod, which consumes the secret as an environment variable or as a file (using a **secret** volume), as shown in the "Understanding how to create secrets" section.

**Additional resources**

- For more information on using secrets in pods, see Understanding how to create secrets .

### 2.6.2.3. Creating a service account token secret

As an administrator, you can create a service account token secret, which allows you to distribute a service account token to applications that must authenticate to the API.

> **NOTE**
>
> It is recommended to obtain bound service account tokens using the TokenRequest API instead of using service account token secrets. The tokens obtained from the TokenRequest API are more secure than the tokens stored in secrets, because they have a bounded lifetime and are not readable by other API clients.
>
> You should create a service account token secret only if you cannot use the TokenRequest API and if the security exposure of a non-expiring token in a readable API object is acceptable to you.
>
> See the Additional resources section that follows for information on creating bound service account tokens.

**Procedure**

1. Create a **Secret** object in a YAML file on a control plane node:

   **Example secret object:**

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: secret-sa-sample
     annotations:
       kubernetes.io/service-account.name: "sa-name" 1
   type: kubernetes.io/service-account-token 2
   ```

   **1** Specifies an existing service account name. If you are creating both the **ServiceAccount** and the **Secret** objects, create the **ServiceAccount** object first.

   **2** Specifies a service account token secret.

2. Use the following command to create the **Secret** object:

   ```
   $ oc create -f <filename>.yaml
   ```

3. To use the secret in a pod:

   a. Update the pod's service account to reference the secret, as shown in the "Understanding how to create secrets" section.

b.  Create the pod, which consumes the secret as an environment variable or as a file (using a **secret** volume), as shown in the "Understanding how to create secrets" section.

**Additional resources**

- For more information on using secrets in pods, see Understanding how to create secrets .

- For information on requesting bound service account tokens, see Using bound service account tokens

- For information on creating service accounts, see Understanding and creating service accounts .

### 2.6.2.4. Creating a basic authentication secret

As an administrator, you can create a basic authentication secret, which allows you to store the credentials needed for basic authentication. When using this secret type, the **data** parameter of the **Secret** object must contain the following keys encoded in the base64 format:

- **username**: the user name for authentication

- **password**: the password or token for authentication

> **NOTE**
>
> You can use the **stringData** parameter to use clear text content.

**Procedure**

1.  Create a **Secret** object in a YAML file on a control plane node:

    **Example secret object**

    ```
    apiVersion: v1
    kind: Secret
    metadata:
      name: secret-basic-auth
    type: kubernetes.io/basic-auth 1
    data:
    stringData: 2
      username: admin
      password: t0p-Secret
    ```

    **1**  Specifies a basic authentication secret.

    **2**  Specifies the basic authentication values to use.

2.  Use the following command to create the **Secret** object:

    ```
    $ oc create -f <filename>.yaml
    ```

3.  To use the secret in a pod:

a. Update the pod's service account to reference the secret, as shown in the "Understanding how to create secrets" section.

b. Create the pod, which consumes the secret as an environment variable or as a file (using a **secret** volume), as shown in the "Understanding how to create secrets" section.

**Additional resources**

- For more information on using secrets in pods, see Understanding how to create secrets .

### 2.6.2.5. Creating an SSH authentication secret

As an administrator, you can create an SSH authentication secret, which allows you to store data used for SSH authentication. When using this secret type, the **data** parameter of the **Secret** object must contain the SSH credential to use.

**Procedure**

1. Create a **Secret** object in a YAML file on a control plane node:

   **Example secret object:**

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: secret-ssh-auth
   type: kubernetes.io/ssh-auth ❶
   data:
     ssh-privatekey: | ❷
           MIIEpQIBAAKCAQEAulqb/Y ...
   ```

   ❶ Specifies an SSH authentication secret.

   ❷ Specifies the SSH key/value pair as the SSH credentials to use.

2. Use the following command to create the **Secret** object:

   ```
   $ oc create -f <filename>.yaml
   ```

3. To use the secret in a pod:

   a. Update the pod's service account to reference the secret, as shown in the "Understanding how to create secrets" section.

   b. Create the pod, which consumes the secret as an environment variable or as a file (using a **secret** volume), as shown in the "Understanding how to create secrets" section.

**Additional resources**

- Understanding how to create secrets .

### 2.6.2.6. Creating a Docker configuration secret

As an administrator, you can create a Docker configuration secret, which allows you to store the credentials for accessing a container image registry.

- **kubernetes.io/dockercfg**. Use this secret type to store your local Docker configuration file. The **data** parameter of the **secret** object must contain the contents of a **.dockercfg** file encoded in the base64 format.

- **kubernetes.io/dockerconfigjson**. Use this secret type to store your local Docker configuration JSON file. The **data** parameter of the **secret** object must contain the contents of a **.docker/config.json** file encoded in the base64 format.

**Procedure**

1. Create a **Secret** object in a YAML file on a control plane node.

   **Example Docker configuration secret object**

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: secret-docker-cfg
     namespace: my-project
   type: kubernetes.io/dockerconfig ❶
   data:

   .dockerconfig:bm5ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2cgYXV
   0aCBrZXlzCg== ❷
   ```

   ❶ Specifies that the secret is using a Docker configuration file.

   ❷ The output of a base64-encoded Docker configuration file

   **Example Docker configuration JSON secret object**

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: secret-docker-json
     namespace: my-project
   type: kubernetes.io/dockerconfig ❶
   data:

   .dockerconfigjson:bm5ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2cg
   YXV0aCBrZXlzCg== ❷
   ```

   ❶ Specifies that the secret is using a Docker configuration JSONfile.

   ❷ The output of a base64-encoded Docker configuration JSON file

2. Use the following command to create the **Secret** object

   ```
   $ oc create -f <filename>.yaml
   ```

3. To use the secret in a pod:

    a. Update the pod's service account to reference the secret, as shown in the "Understanding how to create secrets" section.

    b. Create the pod, which consumes the secret as an environment variable or as a file (using a **secret** volume), as shown in the "Understanding how to create secrets" section.

**Additional resources**

- For more information on using secrets in pods, see Understanding how to create secrets .

### 2.6.3. Understanding how to update secrets

When you modify the value of a secret, the value (used by an already running pod) will not dynamically change. To change a secret, you must delete the original pod and create a new pod (perhaps with an identical PodSpec).

Updating a secret follows the same workflow as deploying a new Container image. You can use the **kubectl rolling-update** command.

The **resourceVersion** value in a secret is not specified when it is referenced. Therefore, if a secret is updated at the same time as pods are starting, the version of the secret that is used for the pod is not defined.

> **NOTE**
>
> Currently, it is not possible to check the resource version of a secret object that was used when a pod was created. It is planned that pods will report this information, so that a controller could restart ones using an old **resourceVersion**. In the interim, do not update the data of existing secrets, but create new ones with distinct names.

### 2.6.4. About using signed certificates with secrets

To secure communication to your service, you can configure OpenShift Container Platform to generate a signed serving certificate/key pair that you can add into a secret in a project.

A *service serving certificate secret* is intended to support complex middleware applications that need out-of-the-box certificates. It has the same settings as the server certificates generated by the administrator tooling for nodes and masters.

**Service Pod spec configured for a service serving certificates secret.**

```
apiVersion: v1
kind: Service
metadata:
  name: registry
  annotations:
    service.beta.openshift.io/serving-cert-secret-name: registry-cert 1
# ...
```

**1**    Specify the name for the certificate

Other pods can trust cluster-created certificates (which are only signed for internal DNS names), by using the CA bundle in the ***/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt*** file that is automatically mounted in their pod.

The signature algorithm for this feature is **x509.SHA256WithRSA**. To manually rotate, delete the generated secret. A new certificate is created.

### 2.6.4.1. Generating signed certificates for use with secrets

To use a signed serving certificate/key pair with a pod, create or edit the service to add the **service.beta.openshift.io/serving-cert-secret-name** annotation, then add the secret to the pod.

#### Procedure

To create a *service serving certificate secret*:

1. Edit the **Pod** spec for your service.

2. Add the **service.beta.openshift.io/serving-cert-secret-name** annotation with the name you want to use for your secret.

   ```
   kind: Service
   apiVersion: v1
   metadata:
     name: my-service
     annotations:
         service.beta.openshift.io/serving-cert-secret-name: my-cert
   spec:
     selector:
       app: MyApp
     ports:
     - protocol: TCP
       port: 80
       targetPort: 9376
   ```
   **1**

   The certificate and key are in PEM format, stored in **tls.crt** and **tls.key** respectively.

3. Create the service:

   ```
   $ oc create -f <file-name>.yaml
   ```

4. View the secret to make sure it was created:

   a. View a list of all secrets:

      ```
      $ oc get secrets
      ```

      **Example output**

      ```
      NAME              TYPE                  DATA    AGE
      my-cert           kubernetes.io/tls       2     9m
      ```

   b. View details on your secret:

      ```
      $ oc describe secret my-cert
      ```

**Example output**

```
Name:         my-cert
Namespace:    openshift-console
Labels:       <none>
Annotations:  service.beta.openshift.io/expiry: 2023-03-08T23:22:40Z
              service.beta.openshift.io/originating-service-name: my-service
              service.beta.openshift.io/originating-service-uid: 640f0ec3-afc2-4380-bf31-
a8c784846a11
              service.beta.openshift.io/expiry: 2023-03-08T23:22:40Z


Type:  kubernetes.io/tls

Data
====
tls.key:  1679 bytes
tls.crt:  2595 bytes
```

5. Edit your **Pod** spec with that secret.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-service-pod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
  volumes:
  - name: foo
    secret:
      secretName: my-cert
      items:
      - key: username
        path: my-group/my-username
        mode: 511
```

When it is available, your pod will run. The certificate will be good for the internal service DNS name, **<service.name>.<service.namespace>.svc**.

The certificate/key pair is automatically replaced when it gets close to expiration. View the expiration date in the **service.beta.openshift.io/expiry** annotation on the secret, which is in RFC3339 format.

> **NOTE**
>
> In most cases, the service DNS name **<service.name>.<service.namespace>.svc** is not externally routable. The primary use of **<service.name>.<service.namespace>.svc** is for intracluster or intraservice communication, and with re-encrypt routes.

## 2.6.5. Troubleshooting secrets

If a service certificate generation fails with (service's **service.beta.openshift.io/serving-cert-generation-error** annotation contains):

> secret/ssl-key references serviceUID 62ad25ca-d703-11e6-9d6f-0e9c0057b608, which does not match 77b6dd80-d716-11e6-9d6f-0e9c0057b60

The service that generated the certificate no longer exists, or has a different **serviceUID**. You must force certificates regeneration by removing the old secret, and clearing the following annotations on the service **service.beta.openshift.io/serving-cert-generation-error**, **service.beta.openshift.io/serving-cert-generation-error-num**:

1. Delete the secret:

   > $ oc delete secret <secret_name>

2. Clear the annotations:

   > $ oc annotate service <service_name> service.beta.openshift.io/serving-cert-generation-error-

   > $ oc annotate service <service_name> service.beta.openshift.io/serving-cert-generation-error-num-

> **NOTE**
>
> The command removing annotation has a **-** after the annotation name to be removed.

## 2.7. CREATING AND USING CONFIG MAPS

The following sections define config maps and how to create and use them.

### 2.7.1. Understanding config maps

Many applications require configuration using some combination of configuration files, command line arguments, and environment variables. In OpenShift Container Platform, these configuration artifacts are decoupled from image content to keep containerized applications portable.

The **ConfigMap** object provides mechanisms to inject containers with configuration data while keeping containers agnostic of OpenShift Container Platform. A config map can be used to store fine-grained information like individual properties or coarse-grained information like entire configuration files or JSON blobs.

The **ConfigMap** API object holds key-value pairs of configuration data that can be consumed in pods or used to store configuration data for system components such as controllers. For example:

**ConfigMap Object Definition**

```
kind: ConfigMap
apiVersion: v1
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
```

```
     name: example-config
     namespace: default
   data:  ①
     example.property.1: hello
     example.property.2: world
     example.property.file: |-
       property.1=value-1
       property.2=value-2
       property.3=value-3
   binaryData:
     bar: L3Jvb3QvMTAw  ②
```

**1 1** Contains the configuration data.

**2** Points to a file that contains non-UTF8 data, for example, a binary Java keystore file. Enter the file data in Base 64.

> **NOTE**
>
> You can use the **binaryData** field when you create a config map from a binary file, such as an image.

Configuration data can be consumed in pods in a variety of ways. A config map can be used to:

- Populate environment variable values in containers

- Set command-line arguments in a container

- Populate configuration files in a volume

Users and system components can store configuration data in a config map.

A config map is similar to a secret, but designed to more conveniently support working with strings that do not contain sensitive information.

**Config map restrictions**
**A config map must be created before its contents can be consumed in pods.**

Controllers can be written to tolerate missing configuration data. Consult individual components configured by using config maps on a case-by-case basis.

**ConfigMap objects reside in a project.**

They can only be referenced by pods in the same project.

**The Kubelet only supports the use of a config map for pods it gets from the API server.**

This includes any pods created by using the CLI, or indirectly from a replication controller. It does not include pods created by using the OpenShift Container Platform node's **--manifest-url** flag, its **--config** flag, or its REST API because these are not common ways to create pods.

## 2.7.2. Creating a config map in the OpenShift Container Platform web console

You can create a config map in the OpenShift Container Platform web console.

Procedure

- To create a config map as a cluster administrator:

  1. In the Administrator perspective, select **Workloads → Config Maps**.

  2. At the top right side of the page, select **Create Config Map**.

  3. Enter the contents of your config map.

  4. Select **Create**.

- To create a config map as a developer:

  1. In the Developer perspective, select **Config Maps**.

  2. At the top right side of the page, select **Create Config Map**.

  3. Enter the contents of your config map.

  4. Select **Create**.

## 2.7.3. Creating a config map by using the CLI

You can use the following command to create a config map from directories, specific files, or literal values.

Procedure

- Create a config map:

```
$ oc create configmap <configmap_name> [options]
```

### 2.7.3.1. Creating a config map from a directory

You can create a config map from a directory. This method allows you to use multiple files within a directory to create a config map.

Procedure

The following example procedure outlines how to create a config map from a directory.

1. Start with a directory with some files that already contain the data with which you want to populate a config map:

```
$ ls example-files
```

**Example output**

```
game.properties
ui.properties
```

```
$ cat example-files/game.properties
```

**Example output**

```
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
```

```
$ cat example-files/ui.properties
```

**Example output**

```
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

2. Create a config map holding the content of each file in this directory by entering the following command:

```
$ oc create configmap game-config \
    --from-file=example-files/
```

When the **--from-file** option points to a directory, each file directly in that directory is used to populate a key in the config map, where the name of the key is the file name, and the value of the key is the content of the file.

For example, the previous command creates the following config map:

```
$ oc describe configmaps game-config
```

**Example output**

```
Name:         game-config
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data

game.properties:    158 bytes
ui.properties:      83 bytes
```

You can see that the two keys in the map are created from the file names in the directory specified in the command. Because the content of those keys might be large, the output of **oc describe** only shows the names of the keys and their sizes.

3. Enter the **oc get** command for the object with the **-o** option to see the values of the keys:

```
$ oc get configmaps game-config -o yaml
```

**Example output**

```
apiVersion: v1
data:
  game.properties: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:34:05Z
  name: game-config
  namespace: default
  resourceVersion: "407"
  selflink: /api/v1/namespaces/default/configmaps/game-config
  uid: 30944725-d66e-11e5-8cd0-68f728db1985
```

### 2.7.3.2. Creating a config map from a file

You can create a config map from a file.

#### Procedure

The following example procedure outlines how to create a config map from a file.

> **NOTE**
>
> If you create a config map from a file, you can include files containing non–UTF8 data that are placed in this field without corrupting the non–UTF8 data. OpenShift Container Platform detects binary files and transparently encodes the file as **MIME**. On the server, the **MIME** payload is decoded and stored without corrupting the data.

You can pass the **--from-file** option multiple times to the CLI. The following example yields equivalent results to the creating from directories example.

1. Create a config map by specifying a specific file:

   ```
   $ oc create configmap game-config-2 \
       --from-file=example-files/game.properties \
       --from-file=example-files/ui.properties
   ```

2. Verify the results:

   ```
   $ oc get configmaps game-config-2 -o yaml
   ```

   **Example output**

```
apiVersion: v1
data:
  game.properties: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:52:05Z
  name: game-config-2
  namespace: default
  resourceVersion: "516"
  selflink: /api/v1/namespaces/default/configmaps/game-config-2
  uid: b4952dc3-d670-11e5-8cd0-68f728db1985
```

You can specify the key to set in a config map for content imported from a file. This can be set by passing a **key=value** expression to the **--from-file** option. For example:

1. Create a config map by specifying a key–value pair:

```
$ oc create configmap game-config-3 \
    --from-file=game-special-key=example-files/game.properties
```

2. Verify the results:

```
$ oc get configmaps game-config-3 -o yaml
```

**Example output**

```
apiVersion: v1
data:
  game-special-key: |- 1
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
  name: game-config-3
  namespace: default
```

```
resourceVersion: "530"
selflink: /api/v1/namespaces/default/configmaps/game-config-3
uid: 05f8da22-d671-11e5-8cd0-68f728db1985
```

**1**    This is the key that you set in the preceding step.

### 2.7.3.3. Creating a config map from literal values

You can supply literal values for a config map.

#### Procedure

The **--from-literal** option takes a **key=value** syntax that allows literal values to be supplied directly on the command line.

1. Create a config map by specifying a literal value:

   ```
   $ oc create configmap special-config \
       --from-literal=special.how=very \
       --from-literal=special.type=charm
   ```

2. Verify the results:

   ```
   $ oc get configmaps special-config -o yaml
   ```

   **Example output**

   ```
   apiVersion: v1
   data:
     special.how: very
     special.type: charm
   kind: ConfigMap
   metadata:
     creationTimestamp: 2016-02-18T19:14:38Z
     name: special-config
     namespace: default
     resourceVersion: "651"
     selflink: /api/v1/namespaces/default/configmaps/special-config
     uid: dadce046-d673-11e5-8cd0-68f728db1985
   ```

### 2.7.4. Use cases: Consuming config maps in pods

The following sections describe some uses cases when consuming **ConfigMap** objects in pods.

#### 2.7.4.1. Populating environment variables in containers by using config maps

Config maps can be used to populate individual environment variables in containers or to populate environment variables in containers from all keys that form valid environment variable names.

As an example, consider the following config map:

**ConfigMap with two environment variables**

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config 1
  namespace: default 2
data:
  special.how: very 3
  special.type: charm 4
```

**1**  Name of the config map.

**2**  The project in which the config map resides. Config maps can only be referenced by pods in the same project.

**3** **4** Environment variables to inject.

**ConfigMap with one environment variable**

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config 1
  namespace: default
data:
  log_level: INFO 2
```

**1**  Name of the config map.

**2**  Environment variable to inject.

**Procedure**

- You can consume the keys of this **ConfigMap** in a pod using **configMapKeyRef** sections.

  **Sample Pod specification configured to inject specific environment variables**

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env: 1
        - name: SPECIAL_LEVEL_KEY 2
          valueFrom:
            configMapKeyRef:
              name: special-config 3
              key: special.how 4
```

```
      - name: SPECIAL_TYPE_KEY
        valueFrom:
          configMapKeyRef:
            name: special-config 5
            key: special.type 6
            optional: true 7
    envFrom: 8
      - configMapRef:
          name: env-config 9
restartPolicy: Never
```

**1**     Stanza to pull the specified environment variables from a **ConfigMap**.

**2**     Name of a pod environment variable that you are injecting a key's value into.

**3** **5** Name of the **ConfigMap** to pull specific environment variables from.

**4** **6** Environment variable to pull from the **ConfigMap**.

**7**     Makes the environment variable optional. As optional, the pod will be started even if the specified **ConfigMap** and keys do not exist.

**8**     Stanza to pull all environment variables from a **ConfigMap**.

**9**     Name of the **ConfigMap** to pull all environment variables from.

When this pod is run, the pod logs will include the following output:

```
SPECIAL_LEVEL_KEY=very
log_level=INFO
```

> **NOTE**
>
> **SPECIAL_TYPE_KEY=charm** is not listed in the example output because **optional: true** is set.

### 2.7.4.2. Setting command-line arguments for container commands with config maps

A config map can also be used to set the value of the commands or arguments in a container. This is accomplished by using the Kubernetes substitution syntax **$(VAR_NAME)**. Consider the following config map:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

**Procedure**

- To inject values into a command in a container, you must consume the keys you want to use as environment variables, as in the consuming ConfigMaps in environment variables use case. Then you can refer to them in a container's command using the **$(VAR_NAME)** syntax.

**Sample Pod specification configured to inject specific environment variables**

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
 containers:
   - name: test-container
     image: gcr.io/google_containers/busybox
     command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)" ]
```
**1**
```
     env:
       - name: SPECIAL_LEVEL_KEY
        valueFrom:
          configMapKeyRef:
            name: special-config
            key: special.how
       - name: SPECIAL_TYPE_KEY
        valueFrom:
          configMapKeyRef:
            name: special-config
            key: special.type
 restartPolicy: Never
```

**1** Inject the values into a command in a container using the keys you want to use as environment variables.

When this pod is run, the output from the echo command run in the test–container container is as follows:

```
very charm
```

### 2.7.4.3. Injecting content into a volume by using config maps

You can inject content into a volume by using config maps.

**Example ConfigMap custom resource (CR)**

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

**Procedure**

You have a couple different options for injecting content into a volume by using config maps.

- The most basic way to inject content into a volume by using a config map is to populate the volume with files where the key is the file name and the content of the file is the value of the key:

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "cat", "/etc/config/special.how" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config ❶
  restartPolicy: Never
```

❶    File containing key.

When this pod is run, the output of the cat command will be:

```
very
```

- You can also control the paths within the volume where config map keys are projected:

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "cat", "/etc/config/path/to/special-key" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
        - key: special.how
          path: path/to/special-key ❶
  restartPolicy: Never
```

❶    Path to config map key.

When this pod is run, the output of the cat command will be:

> very

## 2.8. USING DEVICE PLUGINS TO ACCESS EXTERNAL RESOURCES WITH PODS

Device plugins allow you to use a particular device type (GPU, InfiniBand, or other similar computing resources that require vendor-specific initialization and setup) in your OpenShift Container Platform pod without needing to write custom code.

### 2.8.1. Understanding device plugins

The device plugin provides a consistent and portable solution to consume hardware devices across clusters. The device plugin provides support for these devices through an extension mechanism, which makes these devices available to Containers, provides health checks of these devices, and securely shares them.
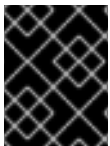
> **IMPORTANT**
>
> OpenShift Container Platform supports the device plugin API, but the device plugin Containers are supported by individual vendors.

A device plugin is a gRPC service running on the nodes (external to the **kubelet**) that is responsible for managing specific hardware resources. Any device plugin must support following remote procedure calls (RPCs):

```
service DevicePlugin {
      // GetDevicePluginOptions returns options to be communicated with Device
      // Manager
      rpc GetDevicePluginOptions(Empty) returns (DevicePluginOptions) {}

      // ListAndWatch returns a stream of List of Devices
      // Whenever a Device state change or a Device disappears, ListAndWatch
      // returns the new list
      rpc ListAndWatch(Empty) returns (stream ListAndWatchResponse) {}

      // Allocate is called during container creation so that the Device
      // Plug-in can run device specific operations and instruct Kubelet
      // of the steps to make the Device available in the container
      rpc Allocate(AllocateRequest) returns (AllocateResponse) {}

      // PreStartcontainer is called, if indicated by Device Plug-in during
      // registration phase, before each container start. Device plug-in
      // can run device specific operations such as reseting the device
      // before making devices available to the container
      rpc PreStartcontainer(PreStartcontainerRequest) returns (PreStartcontainerResponse) {}
}
```

**Example device plugins**

- Nvidia GPU device plugin for COS-based operating system

- [Nvidia official GPU device plugin](#)

- [Solarflare device plugin](#)

- [KubeVirt device plugins: vfio and kvm](#)

> **NOTE**
>
> For easy device plugin reference implementation, there is a stub device plugin in the Device Manager code: *vendor/k8s.io/kubernetes/pkg/kubelet/cm/deviceplugin/device_plugin_stub.go*.

### 2.8.1.1. Methods for deploying a device plugin

- Daemon sets are the recommended approach for device plugin deployments.

- Upon start, the device plugin will try to create a UNIX domain socket at */var/lib/kubelet/device-plugin/* on the node to serve RPCs from Device Manager.

- Since device plugins must manage hardware resources, access to the host file system, as well as socket creation, they must be run in a privileged security context.

- More specific details regarding deployment steps can be found with each device plugin implementation.

## 2.8.2. Understanding the Device Manager

Device Manager provides a mechanism for advertising specialized node hardware resources with the help of plugins known as device plugins.

You can advertise specialized hardware without requiring any upstream code changes.
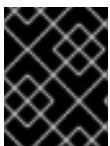
> **IMPORTANT**
>
> OpenShift Container Platform supports the device plugin API, but the device plugin Containers are supported by individual vendors.

Device Manager advertises devices as **Extended Resources**. User pods can consume devices, advertised by Device Manager, using the same **Limit/Request** mechanism, which is used for requesting any other **Extended Resource**.

Upon start, the device plugin registers itself with Device Manager invoking **Register** on the */var/lib/kubelet/device-plugins/kubelet.sock* and starts a gRPC service at */var/lib/kubelet/device-plugins/<plugin>.sock* for serving Device Manager requests.

Device Manager, while processing a new registration request, invokes **ListAndWatch** remote procedure call (RPC) at the device plugin service. In response, Device Manager gets a list of **Device** objects from the plugin over a gRPC stream. Device Manager will keep watching on the stream for new updates from the plugin. On the plugin side, the plugin will also keep the stream open and whenever there is a change in the state of any of the devices, a new device list is sent to the Device Manager over the same streaming connection.

While handling a new pod admission request, Kubelet passes requested **Extended Resources** to the Device Manager for device allocation. Device Manager checks in its database to verify if a corresponding plugin exists or not. If the plugin exists and there are free allocatable devices as well as per local cache,

**Allocate** RPC is invoked at that particular device plugin.

Additionally, device plugins can also perform several other device–specific operations, such as driver installation, device initialization, and device resets. These functionalities vary from implementation to implementation.

## 2.8.3. Enabling Device Manager

Enable Device Manager to implement a device plugin to advertise specialized hardware without any upstream code changes.

Device Manager provides a mechanism for advertising specialized node hardware resources with the help of plugins known as device plugins.

1. Obtain the label associated with the static **MachineConfigPool** CRD for the type of node you want to configure by entering the following command. Perform one of the following steps:

   a. View the machine config:

   ```
   # oc describe machineconfig <name>
   ```

   For example:

   ```
   # oc describe machineconfig 00-worker
   ```

   **Example output**

   ```
   Name:        00-worker
   Namespace:
   Labels:      machineconfiguration.openshift.io/role=worker 1
   ```

   **1**     Label required for the Device Manager.

**Procedure**

1. Create a custom resource (CR) for your configuration change.

   **Sample configuration for a Device Manager CR**

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: KubeletConfig
   metadata:
     name: devicemgr 1
   spec:
     machineConfigPoolSelector:
       matchLabels:
         machineconfiguration.openshift.io: devicemgr 2
     kubeletConfig:
       feature-gates:
         - DevicePlugins=true 3
   ```

   **1**     Assign a name to CR.

**2** Enter the label from the Machine Config Pool.

**3** Set **DevicePlugins** to 'true`.

2. Create the Device Manager:

```
$ oc create -f devicemgr.yaml
```

**Example output**

```
kubeletconfig.machineconfiguration.openshift.io/devicemgr created
```

3. Ensure that Device Manager was actually enabled by confirming that **/var/lib/kubelet/device-plugins/kubelet.sock** is created on the node. This is the UNIX domain socket on which the Device Manager gRPC server listens for new plugin registrations. This sock file is created when the Kubelet is started only if Device Manager is enabled.

## 2.9. INCLUDING POD PRIORITY IN POD SCHEDULING DECISIONS

You can enable pod priority and preemption in your cluster. Pod priority indicates the importance of a pod relative to other pods and queues the pods based on that priority. pod preemption allows the cluster to evict, or preempt, lower-priority pods so that higher-priority pods can be scheduled if there is no available space on a suitable node pod priority also affects the scheduling order of pods and out-of-resource eviction ordering on the node.

To use priority and preemption, you create priority classes that define the relative weight of your pods. Then, reference a priority class in the pod specification to apply that weight for scheduling.

### 2.9.1. Understanding pod priority

When you use the Pod Priority and Preemption feature, the scheduler orders pending pods by their priority, and a pending pod is placed ahead of other pending pods with lower priority in the scheduling queue. As a result, the higher priority pod might be scheduled sooner than pods with lower priority if its scheduling requirements are met. If a pod cannot be scheduled, scheduler continues to schedule other lower priority pods.

#### 2.9.1.1. Pod priority classes

You can assign pods a priority class, which is a non-namespaced object that defines a mapping from a name to the integer value of the priority. The higher the value, the higher the priority.

A priority class object can take any 32-bit integer value smaller than or equal to 1000000000 (one billion). Reserve numbers larger than one billion for critical pods that should not be preempted or evicted. By default, OpenShift Container Platform has two reserved priority classes for critical system pods to have guaranteed scheduling.

```
$ oc get priorityclasses
```

**Example output**

```
NAME              VALUE       GLOBAL-DEFAULT  AGE
cluster-logging    1000000     false           29s
```

```
system-cluster-critical    2000000000    false          72m
system-node-critical       2000001000    false          72m
```

- **system-node-critical** – This priority class has a value of 2000001000 and is used for all pods that should never be evicted from a node. Examples of pods that have this priority class are **sdn-ovs**, **sdn**, and so forth. A number of critical components include the **system-node-critical** priority class by default, for example:

  - master-api

  - master-controller

  - master-etcd

  - sdn

  - sdn-ovs

  - sync

- **system-cluster-critical** – This priority class has a value of 2000000000 (two billion) and is used with pods that are important for the cluster. Pods with this priority class can be evicted from a node in certain circumstances. For example, pods configured with the **system-node-critical** priority class can take priority. However, this priority class does ensure guaranteed scheduling. Examples of pods that can have this priority class are fluentd, add-on components like descheduler, and so forth. A number of critical components include the **system-cluster-critical** priority class by default, for example:

  - fluentd

  - metrics-server

  - descheduler

- **cluster-logging** – This priority is used by Fluentd to make sure Fluentd pods are scheduled to nodes over other apps.

### 2.9.1.2. Pod priority names

After you have one or more priority classes, you can create pods that specify a priority class name in a **Pod** spec. The priority admission controller uses the priority class name field to populate the integer value of the priority. If the named priority class is not found, the pod is rejected.

### 2.9.2. Understanding pod preemption

When a developer creates a pod, the pod goes into a queue. If the developer configured the pod for pod priority or preemption, the scheduler picks a pod from the queue and tries to schedule the pod on a node. If the scheduler cannot find space on an appropriate node that satisfies all the specified requirements of the pod, preemption logic is triggered for the pending pod.

When the scheduler preempts one or more pods on a node, the **nominatedNodeName** field of higher-priority **Pod** spec is set to the name of the node, along with the **nodename** field. The scheduler uses the **nominatedNodeName** field to keep track of the resources reserved for pods and also provides information to the user about preemptions in the clusters.

After the scheduler preempts a lower-priority pod, the scheduler honors the graceful termination period

of the pod. If another node becomes available while scheduler is waiting for the lower-priority pod to terminate, the scheduler can schedule the higher-priority pod on that node. As a result, the **nominatedNodeName** field and **nodeName** field of the **Pod** spec might be different.

Also, if the scheduler preempts pods on a node and is waiting for termination, and a pod with a higher-priority pod than the pending pod needs to be scheduled, the scheduler can schedule the higher-priority pod instead. In such a case, the scheduler clears the **nominatedNodeName** of the pending pod, making the pod eligible for another node.

Preemption does not necessarily remove all lower-priority pods from a node. The scheduler can schedule a pending pod by removing a portion of the lower-priority pods.

The scheduler considers a node for pod preemption only if the pending pod can be scheduled on the node.

### 2.9.2.1. Non-preempting priority classes (Technology Preview)

Pods with the preemption policy set to **Never** are placed in the scheduling queue ahead of lower-priority pods, but they cannot preempt other pods. A non-preempting pod waiting to be scheduled stays in the scheduling queue until sufficient resources are free and it can be scheduled. Non-preempting pods, like other pods, are subject to scheduler back-off. This means that if the scheduler tries unsuccessfully to schedule these pods, they are retried with lower frequency, allowing other pods with lower priority to be scheduled before them.

Non-preempting pods can still be preempted by other, high-priority pods.

### 2.9.2.2. Pod preemption and other scheduler settings

If you enable pod priority and preemption, consider your other scheduler settings:

**Pod priority and pod disruption budget**

A pod disruption budget specifies the minimum number or percentage of replicas that must be up at a time. If you specify pod disruption budgets, OpenShift Container Platform respects them when preempting pods at a best effort level. The scheduler attempts to preempt pods without violating the pod disruption budget. If no such pods are found, lower-priority pods might be preempted despite their pod disruption budget requirements.

**Pod priority and pod affinity**

Pod affinity requires a new pod to be scheduled on the same node as other pods with the same label.

If a pending pod has inter-pod affinity with one or more of the lower-priority pods on a node, the scheduler cannot preempt the lower-priority pods without violating the affinity requirements. In this case, the scheduler looks for another node to schedule the pending pod. However, there is no guarantee that the scheduler can find an appropriate node and pending pod might not be scheduled.

To prevent this situation, carefully configure pod affinity with equal-priority pods.

### 2.9.2.3. Graceful termination of preempted pods

When preempting a pod, the scheduler waits for the pod graceful termination period to expire, allowing the pod to finish working and exit. If the pod does not exit after the period, the scheduler kills the pod. This graceful termination period creates a time gap between the point that the scheduler preempts the pod and the time when the pending pod can be scheduled on the node.

To minimize this gap, configure a small graceful termination period for lower-priority pods.

### 2.9.3. Configuring priority and preemption

You apply pod priority and preemption by creating a priority class object and associating pods to the priority using the **priorityClassName** in your **Pod** specs.

**Sample priority class object**

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority 1
value: 1000000 2
preemptionPolicy: PreemptLowerPriority 3
globalDefault: false 4
description: "This priority class should be used for XYZ service pods only." 5
```

| | |
|---|---|
| **1** | The name of the priority class object. |
| **2** | The priority value of the object. |
| **3** | Optional field that indicates whether this priority class is preempting or non-preempting. The preemption policy defaults to **PreemptLowerPriority**, which allows pods of that priority class to preempt lower-priority pods. If the preemption policy is set to **Never**, pods in that priority class are non-preempting. |
| **4** | Optional field that indicates whether this priority class should be used for pods without a priority class name specified. This field is **false** by default. Only one priority class with **globalDefault** set to **true** can exist in the cluster. If there is no priority class with **globalDefault:true**, the priority of pods with no priority class name is zero. Adding a priority class with **globalDefault:true** affects only pods created after the priority class is added and does not change the priorities of existing pods. |
| **5** | Optional arbitrary text string that describes which pods developers should use with this priority class. |

**Procedure**

To configure your cluster to use priority and preemption:

1. Create one or more priority classes:

   a. Specify a name and value for the priority.

   b. Optionally specify the **globalDefault** field in the priority class and a description.

2. Create a **Pod** spec or edit existing pods to include the name of a priority class, similar to the following:

   **Sample Pod spec with priority class name**

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: nginx
     labels:
       env: test
   ```

```
spec:
 containers:
 - name: nginx
   image: nginx
   imagePullPolicy: IfNotPresent
 priorityClassName: high-priority 1
```

**1**    Specify the priority class to use with this pod.

3. Create the pod:

```
$ oc create -f <file-name>.yaml
```

You can add the priority name directly to the pod configuration or to a pod template.

## 2.10. PLACING PODS ON SPECIFIC NODES USING NODE SELECTORS

A *node selector* specifies a map of key–value pairs. The rules are defined using custom labels on nodes and selectors specified in pods.

For the pod to be eligible to run on a node, the pod must have the indicated key–value pairs as the label on the node.

If you are using node affinity and node selectors in the same pod configuration, see the important considerations below.

### 2.10.1. Using node selectors to control pod placement

You can use node selectors on pods and labels on nodes to control where the pod is scheduled. With node selectors, OpenShift Container Platform schedules the pods on nodes that contain matching labels.

You add labels to a node, a machine set, or a machine config. Adding the label to the machine set ensures that if the node or machine goes down, new nodes have the label. Labels added to a node or machine config do not persist if the node or machine goes down.

To add node selectors to an existing pod, add a node selector to the controlling object for that pod, such as a **ReplicaSet** object, **DaemonSet** object, **StatefulSet** object, **Deployment** object, or **DeploymentConfig** object. Any existing pods under that controlling object are recreated on a node with a matching label. If you are creating a new pod, you can add the node selector directly to the **Pod** spec.

> **NOTE**
>
> You cannot add a node selector directly to an existing scheduled pod.

#### Prerequisites

To add a node selector to existing pods, determine the controlling object for that pod. For example, the **router-default-66d5cf9464-m2g75** pod is controlled by the **router-default-66d5cf9464** replica set:

```
$ oc describe pod router-default-66d5cf9464-7pwkc

Name:           router-default-66d5cf9464-7pwkc
```

```
Namespace:         openshift-ingress

....

Controlled By:       ReplicaSet/router-default-66d5cf9464
```

The web console lists the controlling object under **ownerReferences** in the pod YAML:

```
ownerReferences:
  - apiVersion: apps/v1
    kind: ReplicaSet
    name: router-default-66d5cf9464
    uid: d81dd094-da26-11e9-a48a-128e7edf0312
    controller: true
    blockOwnerDeletion: true
```

**Procedure**

1. Add labels to a node by using a machine set or editing the node directly:

   - Use a **MachineSet** object to add labels to nodes managed by the machine set when a node is created:

     a. Run the following command to add labels to a **MachineSet** object:

        ```
        $ oc patch MachineSet <name> --type='json' -
        p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"<key>"="
        <value>","<key>"="<value>"}}]'  -n openshift-machine-api
        ```

        For example:

        ```
        $ oc patch MachineSet abc612-msrtw-worker-us-east-1c  --type='json' -
        p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"type":"user-
        node","region":"east"}}]'  -n openshift-machine-api
        ```

        **TIP**

        You can alternatively apply the following YAML to add labels to a machine set:

        ```
        apiVersion: machine.openshift.io/v1beta1
        kind: MachineSet
        metadata:
          name: <machineset>
          namespace: openshift-machine-api
        spec:
          template:
            spec:
              metadata:
                labels:
                  region: "east"
                  type: "user-node"
        ```

b. Verify that the labels are added to the **MachineSet** object by using the **oc edit** command:
For example:

```
$ oc edit MachineSet abc612-msrtw-worker-us-east-1c -n openshift-machine-api
```

**Example MachineSet object**

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet

....

spec:
...
  template:
    metadata:
...
    spec:
      metadata:
        labels:
          region: east
          type: user-node
....
```

- Add labels directly to a node:

  a. Edit the **Node** object for the node:

  ```
  $ oc label nodes <name> <key>=<value>
  ```

  For example, to label a node:

  ```
  $ oc label nodes ip-10-0-142-25.ec2.internal type=user-node region=east
  ```

  **TIP**

  You can alternatively apply the following YAML to add labels to a node:

  ```
  kind: Node
  apiVersion: v1
  metadata:
    name: <node_name>
    labels:
      type: "user-node"
      region: "east"
  ```

  b. Verify that the labels are added to the node:

  ```
  $ oc get nodes -l type=user-node,region=east
  ```

  **Example output**

```
NAME                    STATUS   ROLES    AGE   VERSION
ip-10-0-142-25.ec2.internal   Ready    worker   17m   v1.18.3+002a51f
```

2. Add the matching node selector to a pod:

- To add a node selector to existing and future pods, add a node selector to the controlling object for the pods:

  **Example ReplicaSet object with labels**

  ```
  kind: ReplicaSet

  ....

  spec:

  ....

    template:
      metadata:
        creationTimestamp: null
        labels:
          ingresscontroller.operator.openshift.io/deployment-ingresscontroller: default
          pod-template-hash: 66d5cf9464
      spec:
        nodeSelector:
          kubernetes.io/os: linux
          node-role.kubernetes.io/worker: "
          type: user-node ❶
  ```

  ❶ Add the node selector.

- To add a node selector to a specific, new pod, add the selector to the **Pod** object directly:

  **Example Pod object with a node selector**

  ```
  apiVersion: v1
  kind: Pod

  ....

  spec:
    nodeSelector:
      region: east
      type: user-node
  ```

  > **NOTE**
  >
  > You cannot add a node selector directly to an existing scheduled pod.

# CHAPTER 3. CONTROLLING POD PLACEMENT ONTO NODES (SCHEDULING)

## 3.1. CONTROLLING POD PLACEMENT USING THE SCHEDULER

Pod scheduling is an internal process that determines placement of new pods onto nodes within the cluster.

The scheduler code has a clean separation that watches new pods as they get created and identifies the most suitable node to host them. It then creates bindings (pod to node bindings) for the pods using the master API.

**Default pod scheduling**

OpenShift Container Platform comes with a default scheduler that serves the needs of most users. The default scheduler uses both inherent and customization tools to determine the best fit for a pod.

**Advanced pod scheduling**

In situations where you might want more control over where new pods are placed, the OpenShift Container Platform advanced scheduling features allow you to configure a pod so that the pod is required or has a preference to run on a particular node or alongside a specific pod by.

- Using pod affinity and anti-affinity rules.

- Controlling pod placement with pod affinity.

- Controlling pod placement with node affinity.

- Placing pods on overcomitted nodes.

- Controlling pod placement with node selectors.

- Controlling pod placement with taints and tolerations.

### 3.1.1. Scheduler Use Cases

One of the important use cases for scheduling within OpenShift Container Platform is to support flexible affinity and anti-affinity policies.

#### 3.1.1.1. Infrastructure Topological Levels

Administrators can define multiple topological levels for their infrastructure (nodes) by specifying labels on nodes. For example: **region=r1**, **zone=z1**, **rack=s1**.

These label names have no particular meaning and administrators are free to name their infrastructure levels anything, such as city/building/room. Also, administrators can define any number of levels for their infrastructure topology, with three levels usually being adequate (such as: **regions → zones → racks**). Administrators can specify affinity and anti-affinity rules at each of these levels in any combination.

#### 3.1.1.2. Affinity

Administrators should be able to configure the scheduler to specify affinity at any topological level, or even at multiple levels. Affinity at a particular level indicates that all pods that belong to the same service are scheduled onto nodes that belong to the same level. This handles any latency requirements

of applications by allowing administrators to ensure that peer pods do not end up being too geographically separated. If no node is available within the same affinity group to host the pod, then the pod is not scheduled.

If you need greater control over where the pods are scheduled, see Controlling pod placement on nodes using node affinity rules and Placing pods relative to other pods using affinity and anti-affinity rules .

These advanced scheduling features allow administrators to specify which node a pod can be scheduled on and to force or reject scheduling relative to other pods.

### 3.1.1.3. Anti-Affinity

Administrators should be able to configure the scheduler to specify anti-affinity at any topological level, or even at multiple levels. Anti-affinity (or 'spread') at a particular level indicates that all pods that belong to the same service are spread across nodes that belong to that level. This ensures that the application is well spread for high availability purposes. The scheduler tries to balance the service pods across all applicable nodes as evenly as possible.

If you need greater control over where the pods are scheduled, see Controlling pod placement on nodes using node affinity rules and Placing pods relative to other pods using affinity and anti-affinity rules .

These advanced scheduling features allow administrators to specify which node a pod can be scheduled on and to force or reject scheduling relative to other pods.

## 3.2. CONFIGURING THE DEFAULT SCHEDULER TO CONTROL POD PLACEMENT

The default OpenShift Container Platform pod scheduler is responsible for determining placement of new pods onto nodes within the cluster. It reads data from the pod and tries to find a node that is a good fit based on configured policies. It is completely independent and exists as a standalone/pluggable solution. It does not modify the pod and just creates a binding for the pod that ties the pod to the particular node.

> **IMPORTANT**
>
> Configuring a scheduler policy is deprecated and is planned for removal in a future release. For more information on the Technology Preview alternative, see Scheduling pods using a scheduler profile.

A selection of predicates and priorities defines the policy for the scheduler. See Modifying scheduler policy for a list of predicates and priorities.

**Sample default scheduler object**

```
apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  annotations:
    release.openshift.io/create-only: "true"
  creationTimestamp: 2019-05-20T15:39:01Z
  generation: 1
  name: cluster
  resourceVersion: "1491"
  selfLink: /apis/config.openshift.io/v1/schedulers/cluster
```

```
    uid: 6435dd99-7b15-11e9-bd48-0aec821b8e34
  spec:
   policy: 1
     name: scheduler-policy
   defaultNodeSelector: type=user-node,region=east 2
```

**1**    You can specify the name of a custom scheduler policy file.

**2**    Optional: Specify a default node selector to restrict pod placement to specific nodes. The default node selector is applied to the pods created in all namespaces. Pods can be scheduled on nodes with labels that match the default node selector and any existing pod node selectors. Namespaces having project-wide node selectors are not impacted even if this field is set.

### 3.2.1. Understanding default scheduling

The existing generic scheduler is the default platform-provided scheduler *engine* that selects a node to host the pod in a three-step operation:

**Filters the Nodes**

The available nodes are filtered based on the constraints or requirements specified. This is done by running each node through the list of filter functions called *predicates*.

**Prioritize the Filtered List of Nodes**

This is achieved by passing each node through a series of priority_ functions that assign it a score between 0 - 10, with 0 indicating a bad fit and 10 indicating a good fit to host the pod. The scheduler configuration can also take in a simple *weight* (positive numeric value) for each priority function. The node score provided by each priority function is multiplied by the weight (default weight for most priorities is 1) and then combined by adding the scores for each node provided by all the priorities. This weight attribute can be used by administrators to give higher importance to some priorities.

**Select the Best Fit Node**

The nodes are sorted based on their scores and the node with the highest score is selected to host the pod. If multiple nodes have the same high score, then one of them is selected at random.

#### 3.2.1.1. Understanding Scheduler Policy

The selection of the predicate and priorities defines the policy for the scheduler.

The scheduler configuration file is a JSON file, which must be named **policy.cfg**, that specifies the predicates and priorities the scheduler will consider.

In the absence of the scheduler policy file, the default scheduler behavior is used.
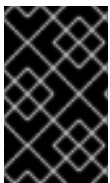
> **IMPORTANT**
>
> The predicates and priorities defined in the scheduler configuration file completely override the default scheduler policy. If any of the default predicates and priorities are required, you must explicitly specify the functions in the policy configuration.

**Sample scheduler config map**

```
  apiVersion: v1
  data:
```

```
  policy.cfg: |
    {
      "kind" : "Policy",
      "apiVersion" : "v1",
      "predicates" : [
            {"name" : "MaxGCEPDVolumeCount"},
            {"name" : "GeneralPredicates"},
            {"name" : "MaxAzureDiskVolumeCount"},
            {"name" : "MaxCSIVolumeCountPred"},
            {"name" : "CheckVolumeBinding"},
            {"name" : "MaxEBSVolumeCount"},
            {"name" : "MatchInterPodAffinity"},
            {"name" : "CheckNodeUnschedulable"},
            {"name" : "NoDiskConflict"},
            {"name" : "NoVolumeZoneConflict"},
            {"name" : "PodToleratesNodeTaints"}
            ],
      "priorities" : [
            {"name" : "LeastRequestedPriority", "weight" : 1},
            {"name" : "BalancedResourceAllocation", "weight" : 1},
            {"name" : "ServiceSpreadingPriority", "weight" : 1},
            {"name" : "NodePreferAvoidPodsPriority", "weight" : 1},
            {"name" : "NodeAffinityPriority", "weight" : 1},
            {"name" : "TaintTolerationPriority", "weight" : 1},
            {"name" : "ImageLocalityPriority", "weight" : 1},
            {"name" : "SelectorSpreadPriority", "weight" : 1},
            {"name" : "InterPodAffinityPriority", "weight" : 1},
            {"name" : "EqualPriority", "weight" : 1}
            ]
    }
kind: ConfigMap
metadata:
  creationTimestamp: "2019-09-17T08:42:33Z"
  name: scheduler-policy
  namespace: openshift-config
  resourceVersion: "59500"
  selfLink: /api/v1/namespaces/openshift-config/configmaps/scheduler-policy
  uid: 17ee8865-d927-11e9-b213-02d1e1709840`
```

The ❶ **GeneralPredicates** predicate represents the **PodFitsResources**, **HostName**, **PodFitsHostPorts**, and **MatchNodeSelector** predicates. Because you are not allowed to configure the same predicate multiple times, the **GeneralPredicates** predicate cannot be used alongside any of the four represented predicates.

## 3.2.2. Creating a scheduler policy file

You can change the default scheduling behavior by creating a JSON file with the desired predicates and priorities. You then generate a config map from the JSON file and point the **cluster** Scheduler object to use the config map.

### Procedure

To configure the scheduler policy:

1. Create a JSON file named **policy.cfg** with the desired predicates and priorities.

**Sample scheduler JSON file**

```
{
"kind" : "Policy",
"apiVersion" : "v1",
"predicates" : [ ❶
    {"name" : "MaxGCEPDVolumeCount"},
    {"name" : "GeneralPredicates"},
    {"name" : "MaxAzureDiskVolumeCount"},
    {"name" : "MaxCSIVolumeCountPred"},
    {"name" : "CheckVolumeBinding"},
    {"name" : "MaxEBSVolumeCount"},
    {"name" : "MatchInterPodAffinity"},
    {"name" : "CheckNodeUnschedulable"},
    {"name" : "NoDiskConflict"},
    {"name" : "NoVolumeZoneConflict"},
    {"name" : "PodToleratesNodeTaints"}
    ],
"priorities" : [ ❷
    {"name" : "LeastRequestedPriority", "weight" : 1},
    {"name" : "BalancedResourceAllocation", "weight" : 1},
    {"name" : "ServiceSpreadingPriority", "weight" : 1},
    {"name" : "NodePreferAvoidPodsPriority", "weight" : 1},
    {"name" : "NodeAffinityPriority", "weight" : 1},
    {"name" : "TaintTolerationPriority", "weight" : 1},
    {"name" : "ImageLocalityPriority", "weight" : 1},
    {"name" : "SelectorSpreadPriority", "weight" : 1},
    {"name" : "InterPodAffinityPriority", "weight" : 1},
    {"name" : "EqualPriority", "weight" : 1}
    ]
}
```

❶     Add the predicates as needed.

❷     Add the priorities as needed.

2. Create a config map based on the scheduler JSON file:

```
$ oc create configmap -n openshift-config --from-file=policy.cfg <configmap-name> ❶
```

❶     Enter a name for the config map.

For example:

```
$ oc create configmap -n openshift-config --from-file=policy.cfg scheduler-policy
```

**Example output**

```
configmap/scheduler-policy created
```

**TIP**

You can alternatively apply the following YAML to create the config map:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: scheduler-policy
  namespace: openshift-config
data: ❶
  policy.cfg: |
    {
    "kind": "Policy",
    "apiVersion": "v1",
    "predicates": [
        {
            "name": "RequireRegion",
            "argument": {
              "labelPreference":
                  {"label": "region"},
                  {"presence": true}
            }
        }
    ],
    "priorities": [
        {
            "name":"ZonePreferred",
            "weight" : 1,
            "argument": {
              "labelPreference":
                  {"label": "zone"},
                  {"presence": true}
            }
        }
    ]
    }
```

❶  The **policy.cfg** file in JSON format with predicates and priorities.

3. Edit the Scheduler Operator custom resource to add the config map:

```
$ oc patch Scheduler cluster --type='merge' -p '{"spec":{"policy":{"name":"<configmap-name>"}}}' --type=merge ❶
```

❶  Specify the name of the config map.

For example:

```
$ oc patch Scheduler cluster --type='merge' -p '{"spec":{"policy":{"name":"scheduler-policy"}}}' --type=merge
```

**TIP**

You can alternatively apply the following YAML to add the config map:

```
apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  name: cluster
spec:
  mastersSchedulable: false
  policy:
    name: scheduler-policy 1
```

**1**    Add the name of the scheduler policy config map.

After making the change to the **Scheduler** config resource, wait for the **openshift-kube-apiserver** pods to redeploy. This can take several minutes. Until the pods redeploy, new scheduler does not take effect.

4. Verify the scheduler policy is configured by viewing the log of a scheduler pod in the **openshift-kube-scheduler** namespace. The following command checks for the predicates and priorities that are being registered by the scheduler:

```
$ oc logs <scheduler-pod> | grep predicates
```

For example:

```
$ oc logs openshift-kube-scheduler-ip-10-0-141-29.ec2.internal | grep predicates
```

**Example output**

```
Creating scheduler with fit predicates 'map[MaxGCEPDVolumeCount:{}
MaxAzureDiskVolumeCount:{} CheckNodeUnschedulable:{} NoDiskConflict:{}
NoVolumeZoneConflict:{} GeneralPredicates:{} MaxCSIVolumeCountPred:{}
CheckVolumeBinding:{} MaxEBSVolumeCount:{} MatchInterPodAffinity:{}
PodToleratesNodeTaints:{}]' and priority functions 'map[InterPodAffinityPriority:{}
LeastRequestedPriority:{} ServiceSpreadingPriority:{} ImageLocalityPriority:{}
SelectorSpreadPriority:{} EqualPriority:{} BalancedResourceAllocation:{}
NodePreferAvoidPodsPriority:{} NodeAffinityPriority:{} TaintTolerationPriority:{}]'
```

## 3.2.3. Modifying scheduler policies

You change scheduling behavior by creating or editing your scheduler policy config map in the **openshift-config** project. Add and remove predicates and priorities to the config map to create a *scheduler policy*.

### Procedure

To modify the current custom scheduling, use one of the following methods:

- Edit the scheduler policy config map:

```
$ oc edit configmap <configmap-name>  -n openshift-config
```

For example:

```
$ oc edit configmap scheduler-policy -n openshift-config
```

**Example output**

```
apiVersion: v1
data:
  policy.cfg: |
    {
        "kind" : "Policy",
        "apiVersion" : "v1",
        "predicates" : [ ❶
            {"name" : "MaxGCEPDVolumeCount"},
            {"name" : "GeneralPredicates"},
            {"name" : "MaxAzureDiskVolumeCount"},
            {"name" : "MaxCSIVolumeCountPred"},
            {"name" : "CheckVolumeBinding"},
            {"name" : "MaxEBSVolumeCount"},
            {"name" : "MatchInterPodAffinity"},
            {"name" : "CheckNodeUnschedulable"},
            {"name" : "NoDiskConflict"},
            {"name" : "NoVolumeZoneConflict"},
            {"name" : "PodToleratesNodeTaints"}
            ],
        "priorities" : [ ❷
            {"name" : "LeastRequestedPriority", "weight" : 1},
            {"name" : "BalancedResourceAllocation", "weight" : 1},
            {"name" : "ServiceSpreadingPriority", "weight" : 1},
            {"name" : "NodePreferAvoidPodsPriority", "weight" : 1},
            {"name" : "NodeAffinityPriority", "weight" : 1},
            {"name" : "TaintTolerationPriority", "weight" : 1},
            {"name" : "ImageLocalityPriority", "weight" : 1},
            {"name" : "SelectorSpreadPriority", "weight" : 1},
            {"name" : "InterPodAffinityPriority", "weight" : 1},
            {"name" : "EqualPriority", "weight" : 1}
            ]
    }
kind: ConfigMap
metadata:
  creationTimestamp: "2019-09-17T17:44:19Z"
  name: scheduler-policy
  namespace: openshift-config
  resourceVersion: "15370"
  selfLink: /api/v1/namespaces/openshift-config/configmaps/scheduler-policy
```

❶ Add or remove predicates as needed.

❷ Add, remove, or change the weight of predicates as needed.

It can take a few minutes for the scheduler to restart the pods with the updated policy.

- Change the policies and predicates being used:

  1. Remove the scheduler policy config map:

i. Remove the scheduler policy config map:

```
$ oc delete configmap -n openshift-config <name>
```

For example:

```
$ oc delete configmap -n openshift-config  scheduler-policy
```

2. Edit the **policy.cfg** file to add and remove policies and predicates as needed.
For example:

```
$ vi policy.cfg
```

**Example output**

```
apiVersion: v1
data:
  policy.cfg: |
    {
    "kind" : "Policy",
    "apiVersion" : "v1",
    "predicates" : [
         {"name" : "MaxGCEPDVolumeCount"},
         {"name" : "GeneralPredicates"},
         {"name" : "MaxAzureDiskVolumeCount"},
         {"name" : "MaxCSIVolumeCountPred"},
         {"name" : "CheckVolumeBinding"},
         {"name" : "MaxEBSVolumeCount"},
         {"name" : "MatchInterPodAffinity"},
         {"name" : "CheckNodeUnschedulable"},
         {"name" : "NoDiskConflict"},
         {"name" : "NoVolumeZoneConflict"},
         {"name" : "PodToleratesNodeTaints"}
         ],
    "priorities" : [
         {"name" : "LeastRequestedPriority", "weight" : 1},
         {"name" : "BalancedResourceAllocation", "weight" : 1},
         {"name" : "ServiceSpreadingPriority", "weight" : 1},
         {"name" : "NodePreferAvoidPodsPriority", "weight" : 1},
         {"name" : "NodeAffinityPriority", "weight" : 1},
         {"name" : "TaintTolerationPriority", "weight" : 1},
         {"name" : "ImageLocalityPriority", "weight" : 1},
         {"name" : "SelectorSpreadPriority", "weight" : 1},
         {"name" : "InterPodAffinityPriority", "weight" : 1},
         {"name" : "EqualPriority", "weight" : 1}
         ]
    }
```

3. Re-create the scheduler policy config map based on the scheduler JSON file:

```
$ oc create configmap -n openshift-config --from-file=policy.cfg <configmap-name> ❶
```

❶     Enter a name for the config map.

For example:

```
$ oc create configmap -n openshift-config --from-file=policy.cfg scheduler-policy
```

**Example output**

```
configmap/scheduler-policy created
```

Example 3.1. Sample config map based on the scheduler JSON file

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: scheduler-policy
  namespace: openshift-config
data:
  policy.cfg: |
    {
    "kind": "Policy",
    "apiVersion": "v1",
    "predicates": [
        {
            "name": "RequireRegion",
            "argument": {
              "labelPreference":
                  {"label": "region"},
                  {"presence": true}
            }
        }
      ],
    "priorities": [
        {
            "name":"ZonePreferred",
            "weight" : 1,
            "argument": {
              "labelPreference":
                  {"label": "zone"},
                  {"presence": true}
            }
        }
      ]
    }
```

### 3.2.3.1. Understanding the scheduler predicates

Predicates are rules that filter out unqualified nodes.

There are several predicates provided by default in OpenShift Container Platform. Some of these predicates can be customized by providing certain parameters. Multiple predicates can be combined to provide additional filtering of nodes.

#### 3.2.3.1.1. Static Predicates

These predicates do not take any configuration parameters or inputs from the user. These are specified in the scheduler configuration using their exact name.

### 3.2.3.1.1.1. Default Predicates

The default scheduler policy includes the following predicates:

The **NoVolumeZoneConflict** predicate checks that the volumes a pod requests are available in the zone.

```
{"name" : "NoVolumeZoneConflict"}
```

The **MaxEBSVolumeCount** predicate checks the maximum number of volumes that can be attached to an AWS instance.

```
{"name" : "MaxEBSVolumeCount"}
```

The **MaxAzureDiskVolumeCount** predicate checks the maximum number of Azure Disk Volumes.

```
{"name" : "MaxAzureDiskVolumeCount"}
```

The **PodToleratesNodeTaints** predicate checks if a pod can tolerate the node taints.

```
{"name" : "PodToleratesNodeTaints"}
```

The **CheckNodeUnschedulable** predicate checks if a pod can be scheduled on a node with **Unschedulable** spec.

```
{"name" : "CheckNodeUnschedulable"}
```

The **CheckVolumeBinding** predicate evaluates if a pod can fit based on the volumes, it requests, for both bound and unbound PVCs.

- For PVCs that are bound, the predicate checks that the corresponding PV's node affinity is satisfied by the given node.

- For PVCs that are unbound, the predicate searched for available PVs that can satisfy the PVC requirements and that the PV node affinity is satisfied by the given node.

The predicate returns true if all bound PVCs have compatible PVs with the node, and if all unbound PVCs can be matched with an available and node-compatible PV.

```
{"name" : "CheckVolumeBinding"}
```

The **NoDiskConflict** predicate checks if the volume requested by a pod is available.

```
{"name" : "NoDiskConflict"}
```

The **MaxGCEPDVolumeCount** predicate checks the maximum number of Google Compute Engine (GCE) Persistent Disks (PD).

```
{"name" : "MaxGCEPDVolumeCount"}
```

The **MaxCSIVolumeCountPred** predicate determines how many Container Storage Interface (CSI) volumes should be attached to a node and whether that number exceeds a configured limit.

```
{"name" : "MaxCSIVolumeCountPred"}
```

The **MatchInterPodAffinity** predicate checks if the pod affinity/anti-affinity rules permit the pod.

```
{"name" : "MatchInterPodAffinity"}
```

### 3.2.3.1.1.2. Other Static Predicates

OpenShift Container Platform also supports the following predicates:

> **NOTE**
>
> The **CheckNode-\*** predicates cannot be used if the Taint Nodes By Condition feature is enabled. The Taint Nodes By Condition feature is enabled by default.

The **CheckNodeCondition** predicate checks if a pod can be scheduled on a node reporting **out of disk**, **network unavailable**, or **not ready** conditions.

```
{"name" : "CheckNodeCondition"}
```

The **CheckNodeLabelPresence** predicate checks if all of the specified labels exist on a node, regardless of their value.

```
{"name" : "CheckNodeLabelPresence"}
```

The **checkServiceAffinity** predicate checks that ServiceAffinity labels are homogeneous for pods that are scheduled on a node.

```
{"name" : "checkServiceAffinity"}
```

The **PodToleratesNodeNoExecuteTaints** predicate checks if a pod tolerations can tolerate a node **NoExecute** taints.

```
{"name" : "PodToleratesNodeNoExecuteTaints"}
```

### 3.2.3.1.2. General Predicates

The following general predicates check whether non-critical predicates and essential predicates pass. Non-critical predicates are the predicates that only non-critical pods must pass and essential predicates are the predicates that all pods must pass.

*The default scheduler policy includes the general predicates.*

**Non-critical general predicates**
The **PodFitsResources** predicate determines a fit based on resource availability (CPU, memory, GPU, and so forth). The nodes can declare their resource capacities and then pods can specify what resources they require. Fit is based on requested, rather than used resources.

```
{"name" : "PodFitsResources"}
```

—

**Essential general predicates**

The **PodFitsHostPorts** predicate determines if a node has free ports for the requested pod ports (absence of port conflicts).

```
{"name" : "PodFitsHostPorts"}
```

The **HostName** predicate determines fit based on the presence of the Host parameter and a string match with the name of the host.

```
{"name" : "HostName"}
```

The **MatchNodeSelector** predicate determines fit based on node selector (nodeSelector) queries defined in the pod.

```
{"name" : "MatchNodeSelector"}
```

### 3.2.3.2. Understanding the scheduler priorities

Priorities are rules that rank nodes according to preferences.

A custom set of priorities can be specified to configure the scheduler. There are several priorities provided by default in OpenShift Container Platform. Other priorities can be customized by providing certain parameters. Multiple priorities can be combined and different weights can be given to each to impact the prioritization.

#### 3.2.3.2.1. Static Priorities

Static priorities do not take any configuration parameters from the user, except weight. A weight is required to be specified and cannot be 0 or negative.

These are specified in the scheduler policy config map in the **openshift-config** project.

##### 3.2.3.2.1.1. Default Priorities

The default scheduler policy includes the following priorities. Each of the priority function has a weight of **1** except **NodePreferAvoidPodsPriority**, which has a weight of **10000**.

The **NodeAffinityPriority** priority prioritizes nodes according to node affinity scheduling preferences

```
{"name" : "NodeAffinityPriority", "weight" : 1}
```

The **TaintTolerationPriority** priority prioritizes nodes that have a fewer number of *intolerable* taints on them for a pod. An intolerable taint is one which has key **PreferNoSchedule**.

```
{"name" : "TaintTolerationPriority", "weight" : 1}
```

The **ImageLocalityPriority** priority prioritizes nodes that already have requested pod container's images.

```
{"name" : "ImageLocalityPriority", "weight" : 1}
```

The **SelectorSpreadPriority** priority looks for services, replication controllers (RC), replication sets

(RS), and stateful sets that match the pod, then finds existing pods that match those selectors. The scheduler favors nodes that have fewer existing matching pods. Then, it schedules the pod on a node with the smallest number of pods that match those selectors as the pod being scheduled.

```
{"name" : "SelectorSpreadPriority", "weight" : 1}
```

The **InterPodAffinityPriority** priority computes a sum by iterating through the elements of **weightedPodAffinityTerm** and adding *weight* to the sum if the corresponding PodAffinityTerm is satisfied for that node. The node(s) with the highest sum are the most preferred.

```
{"name" : "InterPodAffinityPriority", "weight" : 1}
```

The **LeastRequestedPriority** priority favors nodes with fewer requested resources. It calculates the percentage of memory and CPU requested by pods scheduled on the node, and prioritizes nodes that have the highest available/remaining capacity.

```
{"name" : "LeastRequestedPriority", "weight" : 1}
```

The **BalancedResourceAllocation** priority favors nodes with balanced resource usage rate. It calculates the difference between the consumed CPU and memory as a fraction of capacity, and prioritizes the nodes based on how close the two metrics are to each other. This should always be used together with **LeastRequestedPriority**.

```
{"name" : "BalancedResourceAllocation", "weight" : 1}
```

The **NodePreferAvoidPodsPriority** priority ignores pods that are owned by a controller other than a replication controller.

```
{"name" : "NodePreferAvoidPodsPriority", "weight" : 10000}
```

### 3.2.3.2.1.2. Other Static Priorities

OpenShift Container Platform also supports the following priorities:

The **EqualPriority** priority gives an equal weight of  **1** to all nodes, if no priority configurations are provided. We recommend using this priority only for testing environments.

```
{"name" : "EqualPriority", "weight" : 1}
```

The **MostRequestedPriority** priority prioritizes nodes with most requested resources. It calculates the percentage of memory and CPU requested by pods scheduled on the node, and prioritizes based on the maximum of the average of the fraction of requested to capacity.

```
{"name" : "MostRequestedPriority", "weight" : 1}
```

The **ServiceSpreadingPriority** priority spreads pods by minimizing the number of pods belonging to the same service onto the same machine.

```
{"name" : "ServiceSpreadingPriority", "weight" : 1}
```

### 3.2.3.2.2. Configurable Priorities

You can configure these priorities in the scheduler policy config map, in the **openshift-config** namespace, to add labels to affect how the priorities work.

The type of the priority function is identified by the argument that they take. Since these are configurable, multiple priorities of the same type (but different configuration parameters) can be combined as long as their user-defined names are different.

For information on using these priorities, see Modifying Scheduler Policy.

The **ServiceAntiAffinity** priority takes a label and ensures a good spread of the pods belonging to the same service across the group of nodes based on the label values. It gives the same score to all nodes that have the same value for the specified label. It gives a higher score to nodes within a group with the least concentration of pods.

```
{
"kind": "Policy",
"apiVersion": "v1",

"priorities":[
    {
        "name":"<name>",      ❶
        "weight" : 1   ❷
        "argument":{
            "serviceAntiAffinity":{
                "label": "<label>"   ❸
            }
        }
    }
  ]
}
```

❶ Specify a name for the priority.

❷ Specify a weight. Enter a non-zero positive value.

❸ Specify a label to match.

For example:

```
{
"kind": "Policy",
"apiVersion": "v1",
"priorities": [
    {
        "name":"RackSpread",
        "weight" : 1,
        "argument": {
            "serviceAntiAffinity": {
                "label": "rack"
            }
        }
    }
  ]
}
```

> **NOTE**
>
> In some situations using the **ServiceAntiAffinity** parameter based on custom labels does not spread pod as expected. See this Red Hat Solution .

The **labelPreference** parameter gives priority based on the specified label. If the label is present on a node, that node is given priority. If no label is specified, priority is given to nodes that do not have a label. If multiple priorities with the **labelPreference** parameter are set, all of the priorities must have the same weight.

```
{
"kind": "Policy",
"apiVersion": "v1",
"priorities":[
    {
        "name":"<name>",       1
        "weight" : 1           2
        "argument":{
            "labelPreference":{
                "label": "<label>",    3
                "presence": true       4
            }
        }
    }
  ]
}
```

1. Specify a name for the priority.

2. Specify a weight. Enter a non-zero positive value.

3. Specify a label to match.

4. Specify whether the label is required, either **true** or **false**.

### 3.2.4. Sample Policy Configurations

The configuration below specifies the default scheduler configuration, if it were to be specified using the scheduler policy file.

```
{
"kind": "Policy",
"apiVersion": "v1",
"predicates": [
    {
        "name": "RegionZoneAffinity",    1
        "argument": {
            "serviceAffinity": {         2
                "labels": ["region, zone"]    3
            }
        }
    }
  ],
```

```
"priorities": [
  {
    "name":"RackSpread",
    "weight" : 1,
    "argument": {
      "serviceAntiAffinity": {
        "label": "rack"
      }
    }
  }
]
}
```

**1** The name for the predicate.

**2** The type of predicate.

**3** The labels for the predicate.

**4** The name for the priority.

**5** The type of priority.

**6** The labels for the priority.

In all of the sample configurations below, the list of predicates and priority functions is truncated to include only the ones that pertain to the use case specified. In practice, a complete/meaningful scheduler policy should include most, if not all, of the default predicates and priorities listed above.

The following example defines three topological levels, region (affinity) → zone (affinity) → rack (anti-affinity):

```
{
"kind": "Policy",
"apiVersion": "v1",
"predicates": [
  {
    "name": "RegionZoneAffinity",
    "argument": {
      "serviceAffinity": {
        "labels": ["region, zone"]
      }
    }
  }
],
"priorities": [
  {
    "name":"RackSpread",
    "weight" : 1,
    "argument": {
      "serviceAntiAffinity": {
        "label": "rack"
      }
    }
```

```
      }
    ]
  }
```

The following example defines three topological levels, **city** (affinity) → **building** (anti-affinity) → **room** (anti-affinity):

```
{
"kind": "Policy",
"apiVersion": "v1",
"predicates": [
   {
      "name": "CityAffinity",
      "argument": {
        "serviceAffinity": {
          "label": "city"
        }
      }
   }
 ],
"priorities": [
   {
      "name":"BuildingSpread",
      "weight" : 1,
      "argument": {
        "serviceAntiAffinity": {
           "label": "building"
          }
      }
    },
   {
      "name":"RoomSpread",
      "weight" : 1,
      "argument": {
        "serviceAntiAffinity": {
           "label": "room"
          }
      }
    }
  ]
}
```

The following example defines a policy to only use nodes with the 'region' label defined and prefer nodes with the 'zone' label defined:

```
{
"kind": "Policy",
"apiVersion": "v1",
"predicates": [
   {
      "name": "RequireRegion",
      "argument": {
        "labelPreference": {
           "labels": ["region"],
           "presence": true
```

```
            }
          }
        }
    ],
    "priorities": [
      {
          "name":"ZonePreferred",
          "weight" : 1,
          "argument": {
            "labelPreference": {
                "label": "zone",
                "presence": true
                }
          }
      }
    ]
  }
```

The following example combines both static and configurable predicates and also priorities:

```
  {
  "kind": "Policy",
  "apiVersion": "v1",
  "predicates": [
      {
          "name": "RegionAffinity",
          "argument": {
            "serviceAffinity": {
                "labels": ["region"]
            }
          }
      },
      {
          "name": "RequireRegion",
          "argument": {
            "labelsPresence": {
                "labels": ["region"],
                "presence": true
            }
          }
      },
      {
          "name": "BuildingNodesAvoid",
          "argument": {
            "labelsPresence": {
                "label": "building",
                "presence": false
            }
          }
      },
      {"name" : "PodFitsPorts"},
      {"name" : "MatchNodeSelector"}
      ],
  "priorities": [
      {
          "name": "ZoneSpread",
```

```
      "weight" : 2,
      "argument": {
        "serviceAntiAffinity":{
          "label": "zone"
          }
        }
    },
    {
      "name":"ZonePreferred",
      "weight" : 1,
      "argument": {
        "labelPreference":{
          "label": "zone",
          "presence": true
          }
        }
    },
    {"name" : "ServiceSpreadingPriority", "weight" : 1}
    ]
  }
```

## 3.3. SCHEDULING PODS USING A SCHEDULER PROFILE

You can configure OpenShift Container Platform to use a scheduling profile to schedule pods onto nodes within the cluster.
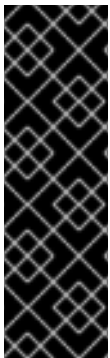
> **IMPORTANT**
>
> Enabling a scheduler profile is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.
>
> For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

### 3.3.1. About scheduler profiles

You can specify a scheduler profile to control how pods are scheduled onto nodes.

> **NOTE**
>
> Scheduler profiles are an alternative to configuring a scheduler policy. Do not set both a scheduler policy and a scheduler profile. If both are set, the scheduler policy takes precedence.

The following scheduler profiles are available:

**LowNodeUtilization**

This profile attempts to spread pods evenly across nodes to get low resource usage per node. This profile provides the default scheduler behavior.

**HighNodeUtilization**

This profile attempts to place as many pods as possible on to as few nodes as possible. This minimizes node count and has high resource usage per node.

**NoScoring**

This is a low-latency profile that strives for the quickest scheduling cycle by disabling all score plugins. This might sacrifice better scheduling decisions for faster ones.

### 3.3.2. Configuring a scheduler profile

You can configure the scheduler to use a scheduler profile.

> **NOTE**
>
> Do not set both a scheduler policy and a scheduler profile. If both are set, the scheduler policy takes precedence.

**Prerequisites**

- Access to the cluster as a user with the **cluster-admin** role.

**Procedure**

1. Edit the **Scheduler** object:

   ```
   $ oc edit scheduler cluster
   ```

2. Specify the profile to use in the **spec.profile** field:

   ```
   apiVersion: config.openshift.io/v1
   kind: Scheduler
   metadata:
     ...
     name: cluster
     resourceVersion: "601"
     selfLink: /apis/config.openshift.io/v1/schedulers/cluster
     uid: b351d6d0-d06f-4a99-a26b-87af62e79f59
   spec:
     mastersSchedulable: false
     policy:
       name: ""
     profile: HighNodeUtilization 1
   ```

   **1** Set to **LowNodeUtilization**, **HighNodeUtilization**, or **NoScoring**.

3. Save the file to apply the changes.

## 3.4. PLACING PODS RELATIVE TO OTHER PODS USING AFFINITY AND ANTI-AFFINITY RULES

Affinity is a property of pods that controls the nodes on which they prefer to be scheduled. Anti-affinity is a property of pods that prevents a pod from being scheduled on a node.

In OpenShift Container Platform *pod affinity* and *pod anti-affinity* allow you to constrain which nodes your pod is eligible to be scheduled on based on the key/value labels on other pods.

## 3.4.1. Understanding pod affinity

*Pod affinity* and *pod anti-affinity* allow you to constrain which nodes your pod is eligible to be scheduled on based on the key/value labels on other pods.

- Pod affinity can tell the scheduler to locate a new pod on the same node as other pods if the label selector on the new pod matches the label on the current pod.

- Pod anti-affinity can prevent the scheduler from locating a new pod on the same node as pods with the same labels if the label selector on the new pod matches the label on the current pod.

For example, using affinity rules, you could spread or pack pods within a service or relative to pods in other services. Anti-affinity rules allow you to prevent pods of a particular service from scheduling on the same nodes as pods of another service that are known to interfere with the performance of the pods of the first service. Or, you could spread the pods of a service across nodes or availability zones to reduce correlated failures.

There are two types of pod affinity rules: *required* and *preferred*.

Required rules **must** be met before a pod can be scheduled on a node. Preferred rules specify that, if the rule is met, the scheduler tries to enforce the rules, but does not guarantee enforcement.

> **NOTE**
>
> Depending on your pod priority and preemption settings, the scheduler might not be able to find an appropriate node for a pod without violating affinity requirements. If so, a pod might not be scheduled.
>
> To prevent this situation, carefully configure pod affinity with equal-priority pods.

You configure pod affinity/anti-affinity through the **Pod** spec files. You can specify a required rule, a preferred rule, or both. If you specify both, the node must first meet the required rule, then attempts to meet the preferred rule.

The following example shows a **Pod** spec configured for pod affinity and anti-affinity.

In this example, the pod affinity rule indicates that the pod can schedule onto a node only if that node has at least one already-running pod with a label that has the key **security** and value **S1**. The pod anti-affinity rule says that the pod prefers to not schedule onto a node if that node is already running a pod with label having key **security** and value **S2**.

Sample **Pod** config file with pod affinity

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity: 1
      requiredDuringSchedulingIgnoredDuringExecution: 2
      - labelSelector:
```

```
      matchExpressions:
      - key: security 3
        operator: In 4
        values:
        - S1 5
      topologyKey: failure-domain.beta.kubernetes.io/zone
    containers:
    - name: with-pod-affinity
      image: docker.io/ocpqe/hello-pod
```

[1] Stanza to configure pod affinity.

[2] Defines a required rule.

[3] [5] The key and value (label) that must be matched to apply the rule.

[4] The operator represents the relationship between the label on the existing pod and the set of values in the **matchExpression** parameters in the specification for the new pod. Can be **In**, **NotIn**, **Exists**, or **DoesNotExist**.

### Sample **Pod** config file with pod anti-affinity

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-antiaffinity
spec:
  affinity:
    podAntiAffinity: 1
      preferredDuringSchedulingIgnoredDuringExecution: 2
      - weight: 100 3
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: security 4
              operator: In 5
              values:
              - S2
          topologyKey: kubernetes.io/hostname
    containers:
    - name: with-pod-affinity
      image: docker.io/ocpqe/hello-pod
```

[1] Stanza to configure pod anti-affinity.

[2] Defines a preferred rule.

[3] Specifies a weight for a preferred rule. The node with the highest weight is preferred.

[4] Description of the pod label that determines when the anti-affinity rule applies. Specify a key and value for the label.

[5] The operator represents the relationship between the label on the existing pod and the set of values in the **matchExpression** parameters in the specification for the new pod. Can be **In**, **NotIn**,

**Exists**, or **DoesNotExist**.

> **NOTE**
>
> If labels on a node change at runtime such that the affinity rules on a pod are no longer met, the pod continues to run on the node.

### 3.4.2. Configuring a pod affinity rule

The following steps demonstrate a simple two-pod configuration that creates pod with a label and a pod that uses affinity to allow scheduling with that pod.

**Procedure**

1. Create a pod with a specific label in the **Pod** spec:

   ```
   $ cat team4.yaml
   apiVersion: v1
   kind: Pod
   metadata:
     name: security-s1
     labels:
       security: S1
   spec:
     containers:
     - name: security-s1
       image: docker.io/ocpqe/hello-pod
   ```

2. When creating other pods, edit the **Pod** spec as follows:

   a. Use the **podAffinity** stanza to configure the **requiredDuringSchedulingIgnoredDuringExecution** parameter or **preferredDuringSchedulingIgnoredDuringExecution** parameter:

   b. Specify the key and value that must be met. If you want the new pod to be scheduled with the other pod, use the same **key** and **value** parameters as the label on the first pod.

   ```
   podAffinity:
     requiredDuringSchedulingIgnoredDuringExecution:
     - labelSelector:
         matchExpressions:
         - key: security
           operator: In
           values:
           - S1
       topologyKey: failure-domain.beta.kubernetes.io/zone
   ```

   c. Specify an **operator**. The operator can be **In**, **NotIn**, **Exists**, or **DoesNotExist**. For example, use the operator **In** to require the label to be in the node.

   d. Specify a **topologyKey**, which is a prepopulated Kubernetes label that the system uses to denote such a topology domain.

3. Create the pod.

```
$ oc create -f <pod-spec>.yaml
```

### 3.4.3. Configuring a pod anti-affinity rule

The following steps demonstrate a simple two-pod configuration that creates pod with a label and a pod that uses an anti-affinity preferred rule to attempt to prevent scheduling with that pod.

**Procedure**

1. Create a pod with a specific label in the **Pod** spec:

   ```
   $ cat team4.yaml
   apiVersion: v1
   kind: Pod
   metadata:
     name: security-s2
     labels:
       security: S2
   spec:
     containers:
     - name: security-s2
       image: docker.io/ocpqe/hello-pod
   ```

2. When creating other pods, edit the **Pod** spec to set the following parameters:

3. Use the **podAntiAffinity** stanza to configure the **requiredDuringSchedulingIgnoredDuringExecution** parameter or **preferredDuringSchedulingIgnoredDuringExecution** parameter:

   a. Specify a weight for the node, 1-100. The node that with highest weight is preferred.

   b. Specify the key and values that must be met. If you want the new pod to not be scheduled with the other pod, use the same **key** and **value** parameters as the label on the first pod.

      ```
      podAntiAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 100
          podAffinityTerm:
            labelSelector:
              matchExpressions:
              - key: security
                operator: In
                values:
                - S2
            topologyKey: kubernetes.io/hostname
      ```

   c. For a preferred rule, specify a weight, 1-100.

   d. Specify an **operator**. The operator can be **In**, **NotIn**, **Exists**, or **DoesNotExist**. For example, use the operator **In** to require the label to be in the node.

4. Specify a **topologyKey**, which is a prepopulated Kubernetes label that the system uses to denote such a topology domain.

5. Create the pod.

```
$ oc create -f <pod-spec>.yaml
```

### 3.4.4. Sample pod affinity and anti-affinity rules

The following examples demonstrate pod affinity and pod anti-affinity.

#### 3.4.4.1. Pod Affinity

The following example demonstrates pod affinity for pods with matching labels and label selectors.

- The pod **team4** has the label **team:4**.

```
$ cat team4.yaml
apiVersion: v1
kind: Pod
metadata:
  name: team4
  labels:
     team: "4"
spec:
  containers:
  - name: ocp
    image: docker.io/ocpqe/hello-pod
```

- The pod **team4a** has the label selector **team:4** under **podAffinity**.

```
$ cat pod-team4a.yaml
apiVersion: v1
kind: Pod
metadata:
  name: team4a
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: team
            operator: In
            values:
            - "4"
        topologyKey: kubernetes.io/hostname
  containers:
  - name: pod-affinity
    image: docker.io/ocpqe/hello-pod
```

- The **team4a** pod is scheduled on the same node as the **team4** pod.

#### 3.4.4.2. Pod Anti-affinity

The following example demonstrates pod anti-affinity for pods with matching labels and label selectors.

- The pod **pod-s1** has the label **security:s1**.

```
cat pod-s1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-s1
  labels:
    security: s1
spec:
  containers:
  - name: ocp
    image: docker.io/ocpqe/hello-pod
```

- The pod **pod-s2** has the label selector **security:s1** under **podAntiAffinity**.

```
cat pod-s2.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-s2
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - s1
        topologyKey: kubernetes.io/hostname
  containers:
  - name: pod-antiaffinity
    image: docker.io/ocpqe/hello-pod
```

- The pod **pod-s2** cannot be scheduled on the same node as **pod-s1**.

### 3.4.4.3. Pod Affinity with no Matching Labels

The following example demonstrates pod affinity for pods without matching labels and label selectors.

- The pod **pod-s1** has the label **security:s1**.

```
$ cat pod-s1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-s1
  labels:
    security: s1
spec:
  containers:
  - name: ocp
    image: docker.io/ocpqe/hello-pod
```

- The pod **pod-s2** has the label selector **security:s2**.

```
$ cat pod-s2.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-s2
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - s2
        topologyKey: kubernetes.io/hostname
  containers:
  - name: pod-affinity
    image: docker.io/ocpqe/hello-pod
```

- The pod **pod-s2** is not scheduled unless there is a node with a pod that has the **security:s2** label. If there is no other pod with that label, the new pod remains in a pending state:

**Example output**

```
NAME     READY   STATUS    RESTARTS  AGE    IP       NODE
pod-s2   0/1     Pending   0         32s    <none>
```

## 3.5. CONTROLLING POD PLACEMENT ON NODES USING NODE AFFINITY RULES

Affinity is a property of pods that controls the nodes on which they prefer to be scheduled.

In OpenShift Container Platform node affinity is a set of rules used by the scheduler to determine where a pod can be placed. The rules are defined using custom labels on the nodes and label selectors specified in pods.

### 3.5.1. Understanding node affinity

Node affinity allows a pod to specify an affinity towards a group of nodes it can be placed on. The node does not have control over the placement.

For example, you could configure a pod to only run on a node with a specific CPU or in a specific availability zone.

There are two types of node affinity rules: *required* and *preferred*.

Required rules **must** be met before a pod can be scheduled on a node. Preferred rules specify that, if the rule is met, the scheduler tries to enforce the rules, but does not guarantee enforcement.

> **NOTE**
>
> If labels on a node change at runtime that results in an node affinity rule on a pod no longer being met, the pod continues to run on the node.

You configure node affinity through the **Pod** spec file. You can specify a required rule, a preferred rule, or both. If you specify both, the node must first meet the required rule, then attempts to meet the preferred rule.

The following example is a **Pod** spec with a rule that requires the pod be placed on a node with a label whose key is **e2e-az-NorthSouth** and whose value is either **e2e-az-North** or **e2e-az-South**:

**Example pod configuration file with a node affinity required rule**

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity: 1
      requiredDuringSchedulingIgnoredDuringExecution: 2
        nodeSelectorTerms:
        - matchExpressions:
          - key: e2e-az-NorthSouth 3
            operator: In 4
            values:
            - e2e-az-North 5
            - e2e-az-South 6
  containers:
  - name: with-node-affinity
    image: docker.io/ocpqe/hello-pod
```

**1** The stanza to configure node affinity.

**2** Defines a required rule.

**3 5 6** The key/value pair (label) that must be matched to apply the rule.

**4** The operator represents the relationship between the label on the node and the set of values in the **matchExpression** parameters in the **Pod** spec. This value can be **In**, **NotIn**, **Exists**, or **DoesNotExist**, **Lt**, or **Gt**.

The following example is a node specification with a preferred rule that a node with a label whose key is **e2e-az-EastWest** and whose value is either **e2e-az-East** or **e2e-az-West** is preferred for the pod:

**Example pod configuration file with a node affinity preferred rule**

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
```

```
    nodeAffinity: 1
      preferredDuringSchedulingIgnoredDuringExecution: 2
      - weight: 1 3
        preference:
          matchExpressions:
          - key: e2e-az-EastWest 4
            operator: In 5
            values:
            - e2e-az-East 6
            - e2e-az-West 7
  containers:
  - name: with-node-affinity
    image: docker.io/ocpqe/hello-pod
```

**1**     The stanza to configure node affinity.

**2**     Defines a preferred rule.

**3**     Specifies a weight for a preferred rule. The node with highest weight is preferred.

**4 6 7** The key/value pair (label) that must be matched to apply the rule.

**5**     The operator represents the relationship between the label on the node and the set of values in the **matchExpression** parameters in the **Pod** spec. This value can be **In**, **NotIn**, **Exists**, or **DoesNotExist**, **Lt**, or **Gt**.

There is no explicit *node anti-affinity* concept, but using the **NotIn** or **DoesNotExist** operator replicates that behavior.

> **NOTE**
>
> If you are using node affinity and node selectors in the same pod configuration, note the following:
>
> - If you configure both **nodeSelector** and **nodeAffinity**, both conditions must be satisfied for the pod to be scheduled onto a candidate node.
>
> - If you specify multiple **nodeSelectorTerms** associated with **nodeAffinity** types, then the pod can be scheduled onto a node if one of the **nodeSelectorTerms** is satisfied.
>
> - If you specify multiple **matchExpressions** associated with **nodeSelectorTerms**, then the pod can be scheduled onto a node only if all **matchExpressions** are satisfied.

## 3.5.2. Configuring a required node affinity rule

Required rules **must** be met before a pod can be scheduled on a node.

### Procedure

The following steps demonstrate a simple configuration that creates a node and a pod that the scheduler is required to place on the node.

1. Add a label to a node using the **oc label node** command:

   ```
   $ oc label node node1 e2e-az-name=e2e-az1
   ```

   **TIP**

   You can alternatively apply the following YAML to add the label:

   ```
   kind: Node
   apiVersion: v1
   metadata:
    name: <node_name>
    labels:
      e2e-az-name: e2e-az1
   ```

2. In the **Pod** spec, use the **nodeAffinity** stanza to configure the **requiredDuringSchedulingIgnoredDuringExecution** parameter:

   a. Specify the key and values that must be met. If you want the new pod to be scheduled on the node you edited, use the same **key** and **value** parameters as the label in the node.

   b. Specify an **operator**. The operator can be **In**, **NotIn**, **Exists**, **DoesNotExist**, **Lt**, or **Gt**. For example, use the operator **In** to require the label to be in the node:

   **Example output**

   ```
   spec:
     affinity:
       nodeAffinity:
         requiredDuringSchedulingIgnoredDuringExecution:
           nodeSelectorTerms:
           - matchExpressions:
             - key: e2e-az-name
               operator: In
               values:
               - e2e-az1
               - e2e-az2
   ```

3. Create the pod:

   ```
   $ oc create -f e2e-az2.yaml
   ```

## 3.5.3. Configuring a preferred node affinity rule

Preferred rules specify that, if the rule is met, the scheduler tries to enforce the rules, but does not guarantee enforcement.

### Procedure

The following steps demonstrate a simple configuration that creates a node and a pod that the scheduler tries to place on the node.

1. Add a label to a node using the **oc label node** command:

```
$ oc label node node1 e2e-az-name=e2e-az3
```

2. In the **Pod** spec, use the **nodeAffinity** stanza to configure the **preferredDuringSchedulingIgnoredDuringExecution** parameter:

    a. Specify a weight for the node, as a number 1–100. The node with highest weight is preferred.

    b. Specify the key and values that must be met. If you want the new pod to be scheduled on the node you edited, use the same **key** and **value** parameters as the label in the node:

    ```
    spec:
      affinity:
        nodeAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 1
            preference:
              matchExpressions:
              - key: e2e-az-name
                operator: In
                values:
                - e2e-az3
    ```

    c. Specify an **operator**. The operator can be **In**, **NotIn**, **Exists**, **DoesNotExist**, **Lt**, or **Gt**. For example, use the Operator **In** to require the label to be in the node.

3. Create the pod.

    ```
    $ oc create -f e2e-az3.yaml
    ```

### 3.5.4. Sample node affinity rules

The following examples demonstrate node affinity.

#### 3.5.4.1. Node affinity with matching labels

The following example demonstrates node affinity for a node and pod with matching labels:

- The Node1 node has the label **zone:us**:

    ```
    $ oc label node node1 zone=us
    ```

    **TIP**

    You can alternatively apply the following YAML to add the label:

    ```
    kind: Node
    apiVersion: v1
    metadata:
      name: <node_name>
      labels:
        zone: us
    ```

- The pod-s1 pod has the **zone** and **us** key/value pair under a required node affinity rule:

```
$ cat pod-s1.yaml
```

**Example output**

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-s1
spec:
  containers:
    - image: "docker.io/ocpqe/hello-pod"
      name: hello-pod
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
            - key: "zone"
              operator: In
              values:
              - us
```

- The pod-s1 pod can be scheduled on Node1:

  ```
  $ oc get pod -o wide
  ```

  **Example output**

  ```
  NAME    READY   STATUS    RESTARTS  AGE   IP    NODE
  pod-s1  1/1     Running   0         4m    IP1   node1
  ```

### 3.5.4.2. Node affinity with no matching labels

The following example demonstrates node affinity for a node and pod without matching labels:

- The Node1 node has the label **zone:emea**:

  ```
  $ oc label node node1 zone=emea
  ```

  **TIP**

  You can alternatively apply the following YAML to add the label:

  ```
  kind: Node
  apiVersion: v1
  metadata:
    name: <node_name>
    labels:
      zone: emea
  ```

- The pod-s1 pod has the **zone** and **us** key/value pair under a required node affinity rule:

```
$ cat pod-s1.yaml
```

**Example output**

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-s1
spec:
  containers:
    - image: "docker.io/ocpqe/hello-pod"
      name: hello-pod
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
            - key: "zone"
              operator: In
              values:
              - us
```

- The pod-s1 pod cannot be scheduled on Node1:

```
$ oc describe pod pod-s1
```

**Example output**

```
...

Events:
 FirstSeen LastSeen Count From            SubObjectPath  Type           Reason
 --------- -------- ----- ----            ------------- --------        ------
 1m     33s    8    default-scheduler Warning     FailedScheduling   No nodes are
available that match all of the following predicates:: MatchNodeSelector (1).
```

### 3.5.5. Additional resources

- For information about changing node labels, see Understanding how to update labels on nodes .

## 3.6. PLACING PODS ONTO OVERCOMMITED NODES

In an *overcommited* state, the sum of the container compute resource requests and limits exceeds the resources available on the system. Overcommitment might be desirable in development environments where a trade–off of guaranteed performance for capacity is acceptable.

Requests and limits enable administrators to allow and manage the overcommitment of resources on a node. The scheduler uses requests for scheduling your container and providing a minimum service guarantee. Limits constrain the amount of compute resource that may be consumed on your node.

### 3.6.1. Understanding overcommitment

Requests and limits enable administrators to allow and manage the overcommitment of resources on a node. The scheduler uses requests for scheduling your container and providing a minimum service guarantee. Limits constrain the amount of compute resource that may be consumed on your node.

OpenShift Container Platform administrators can control the level of overcommit and manage container density on nodes by configuring masters to override the ratio between request and limit set on developer containers. In conjunction with a per-project **LimitRange** object specifying limits and defaults, this adjusts the container limit and request to achieve the desired level of overcommit.

> **NOTE**
>
> That these overrides have no effect if no limits have been set on containers. Create a **LimitRange** object with default limits, per individual project, or in the project template, to ensure that the overrides apply.

After these overrides, the container limits and requests must still be validated by any **LimitRange** object in the project. It is possible, for example, for developers to specify a limit close to the minimum limit, and have the request then be overridden below the minimum limit, causing the pod to be forbidden. This unfortunate user experience should be addressed with future work, but for now, configure this capability and **LimitRange** objects with caution.

## 3.6.2. Understanding nodes overcommitment

In an overcommitted environment, it is important to properly configure your node to provide best system behavior.

When the node starts, it ensures that the kernel tunable flags for memory management are set properly. The kernel should never fail memory allocations unless it runs out of physical memory.

To ensure this behavior, OpenShift Container Platform configures the kernel to always overcommit memory by setting the **vm.overcommit_memory** parameter to **1**, overriding the default operating system setting.

OpenShift Container Platform also configures the kernel not to panic when it runs out of memory by setting the **vm.panic_on_oom** parameter to **0**. A setting of 0 instructs the kernel to call oom_killer in an Out of Memory (OOM) condition, which kills processes based on priority

You can view the current setting by running the following commands on your nodes:

```
$ sysctl -a |grep commit
```

**Example output**

```
vm.overcommit_memory = 1
```

```
$ sysctl -a |grep panic
```

**Example output**

```
vm.panic_on_oom = 0
```

> **NOTE**
>
> The above flags should already be set on nodes, and no further action is required.

You can also perform the following configurations for each node:

- Disable or enforce CPU limits using CPU CFS quotas

- Reserve resources for system processes

- Reserve memory across quality of service tiers

## 3.7. CONTROLLING POD PLACEMENT USING NODE TAINTS

Taints and tolerations allow the node to control which pods should (or should not) be scheduled on them.

### 3.7.1. Understanding taints and tolerations

A *taint* allows a node to refuse a pod to be scheduled unless that pod has a matching *toleration.*

You apply taints to a node through the **Node** specification (**NodeSpec**) and apply tolerations to a pod through the **Pod** specification (**PodSpec**). When you apply a taint a node, the scheduler cannot place a pod on that node unless the pod can tolerate the taint.

**Example taint in a node specification**

```
spec:
  taints:
  - effect: NoExecute
    key: key1
    value: value1
....
```

**Example toleration in a Pod spec**

```
spec:
  tolerations:
  - key: "key1"
    operator: "Equal"
    value: "value1"
    effect: "NoExecute"
    tolerationSeconds: 3600
....
```

Taints and tolerations consist of a key, value, and effect.

**Table 3.1. Taint and toleration components**

| Parameter | Description |
|-----------|-------------|
| **key** | The **key** is any string, up to 253 characters. The key must begin with a letter or number, and may contain letters, numbers, hyphens, dots, and underscores. |

| Parameter | Description |
| --- | --- |
| **value** | The **value** is any string, up to 63 characters. The value must begin with a letter or number, and may contain letters, numbers, hyphens, dots, and underscores. |
| **effect** | The effect is one of the following: |

<table>
<tr><td><b>NoSchedule</b> [1]</td><td><ul><li>New pods that do not match the taint are not scheduled onto that node.</li><li>Existing pods on the node remain.</li></ul></td></tr>
<tr><td><b>PreferNoSchedule</b></td><td><ul><li>New pods that do not match the taint might be scheduled onto that node, but the scheduler tries not to.</li><li>Existing pods on the node remain.</li></ul></td></tr>
<tr><td><b>NoExecute</b></td><td><ul><li>New pods that do not match the taint cannot be scheduled onto that node.</li><li>Existing pods on the node that do not have a matching toleration are removed.</li></ul></td></tr>
</table>

| operator | | |
| --- | --- | --- |
| **Equal** | | The **key**/**value**/**effect** parameters must match. This is the default. |
| **Exists** | | The **key**/**effect** parameters must match. You must leave a blank **value** parameter, which matches any. |

1. If you add a **NoSchedule** taint to a control plane node (also known as the master node) the node must have the **node-role.kubernetes.io/master=:NoSchedule** taint, which is added by default.
   For example:

```
apiVersion: v1
kind: Node
metadata:
  annotations:
    machine.openshift.io/machine: openshift-machine-api/ci-ln-62s7gtb-f76d1-v8jxv-master-0
    machineconfiguration.openshift.io/currentConfig: rendered-master-
cdc1ab7da414629332cc4c3926e6e59c
...
spec:
  taints:
```

```
    - effect: NoSchedule
      key: node-role.kubernetes.io/master
    ...
```

A toleration matches a taint:

- If the **operator** parameter is set to **Equal**:

  - the **key** parameters are the same;

  - the **value** parameters are the same;

  - the **effect** parameters are the same.

- If the **operator** parameter is set to **Exists**:

  - the **key** parameters are the same;

  - the **effect** parameters are the same.

The following taints are built into OpenShift Container Platform:

- **node.kubernetes.io/not-ready**: The node is not ready. This corresponds to the node condition **Ready=False**.

- **node.kubernetes.io/unreachable**: The node is unreachable from the node controller. This corresponds to the node condition **Ready=Unknown**.

- **node.kubernetes.io/memory-pressure**: The node has memory pressure issues. This corresponds to the node condition **MemoryPressure=True**.

- **node.kubernetes.io/disk-pressure**: The node has disk pressure issues. This corresponds to the node condition **DiskPressure=True**.

- **node.kubernetes.io/network-unavailable**: The node network is unavailable.

- **node.kubernetes.io/unschedulable**: The node is unschedulable.

- **node.cloudprovider.kubernetes.io/uninitialized**: When the node controller is started with an external cloud provider, this taint is set on a node to mark it as unusable. After a controller from the cloud-controller-manager initializes this node, the kubelet removes this taint.

- **node.kubernetes.io/pid-pressure**: The node has pid pressure. This corresponds to the node condition **PIDPressure=True**.
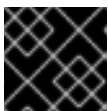
> **IMPORTANT**
>
> OpenShift Container Platform does not set a default pid.available **evictionHard**.

### 3.7.1.1. Understanding how to use toleration seconds to delay pod evictions

You can specify how long a pod can remain bound to a node before being evicted by specifying the **tolerationSeconds** parameter in the **Pod** specification or **MachineSet** object. If a taint with the **NoExecute** effect is added to a node, a pod that does tolerate the taint, which has the **tolerationSeconds** parameter, the pod is not evicted until that time period expires.

**Example output**

```
spec:
  tolerations:
  - key: "key1"
    operator: "Equal"
    value: "value1"
    effect: "NoExecute"
    tolerationSeconds: 3600
```

Here, if this pod is running but does not have a matching toleration, the pod stays bound to the node for 3,600 seconds and then be evicted. If the taint is removed before that time, the pod is not evicted.

### 3.7.1.2. Understanding how to use multiple taints

You can put multiple taints on the same node and multiple tolerations on the same pod. OpenShift Container Platform processes multiple taints and tolerations as follows:

1. Process the taints for which the pod has a matching toleration.

2. The remaining unmatched taints have the indicated effects on the pod:

   - If there is at least one unmatched taint with effect **NoSchedule**, OpenShift Container Platform cannot schedule a pod onto that node.

   - If there is no unmatched taint with effect **NoSchedule** but there is at least one unmatched taint with effect **PreferNoSchedule**, OpenShift Container Platform tries to not schedule the pod onto the node.

   - If there is at least one unmatched taint with effect **NoExecute**, OpenShift Container Platform evicts the pod from the node if it is already running on the node, or the pod is not scheduled onto the node if it is not yet running on the node.

     - Pods that do not tolerate the taint are evicted immediately.

     - Pods that tolerate the taint without specifying **tolerationSeconds** in their **Pod** specification remain bound forever.

     - Pods that tolerate the taint with a specified **tolerationSeconds** remain bound for the specified amount of time.

For example:

- Add the following taints to the node:

  ```
  $ oc adm taint nodes node1 key1=value1:NoSchedule
  ```

  ```
  $ oc adm taint nodes node1 key1=value1:NoExecute
  ```

  ```
  $ oc adm taint nodes node1 key2=value2:NoSchedule
  ```

- The pod has the following tolerations:

  ```
  spec:
    tolerations:
    - key: "key1"
      operator: "Equal"
  ```

```
        value: "value1"
        effect: "NoSchedule"
      - key: "key1"
        operator: "Equal"
        value: "value1"
        effect: "NoExecute"
```

In this case, the pod cannot be scheduled onto the node, because there is no toleration matching the third taint. The pod continues running if it is already running on the node when the taint is added, because the third taint is the only one of the three that is not tolerated by the pod.

### 3.7.1.3. Understanding pod scheduling and node conditions (taint node by condition)

The Taint Nodes By Condition feature, which is enabled by default, automatically taints nodes that report conditions such as memory pressure and disk pressure. If a node reports a condition, a taint is added until the condition clears. The taints have the **NoSchedule** effect, which means no pod can be scheduled on the node unless the pod has a matching toleration.

The scheduler checks for these taints on nodes before scheduling pods. If the taint is present, the pod is scheduled on a different node. Because the scheduler checks for taints and not the actual node conditions, you configure the scheduler to ignore some of these node conditions by adding appropriate pod tolerations.

To ensure backward compatibility, the daemon set controller automatically adds the following tolerations to all daemons:

- node.kubernetes.io/memory-pressure

- node.kubernetes.io/disk-pressure

- node.kubernetes.io/unschedulable (1.10 or later)

- node.kubernetes.io/network-unavailable (host network only)

You can also add arbitrary tolerations to daemon sets.

> **NOTE**
>
> The control plane also adds the **node.kubernetes.io/memory-pressure** toleration on pods that have a QoS class. This is because Kubernetes manages pods in the **Guaranteed** or **Burstable** QoS classes. The new **BestEffort** pods do not get scheduled onto the affected node.

### 3.7.1.4. Understanding evicting pods by condition (taint-based evictions)

The Taint-Based Evictions feature, which is enabled by default, evicts pods from a node that experiences specific conditions, such as **not-ready** and **unreachable**. When a node experiences one of these conditions, OpenShift Container Platform automatically adds taints to the node, and starts evicting and rescheduling the pods on different nodes.

Taint Based Evictions have a **NoExecute** effect, where any pod that does not tolerate the taint is evicted immediately and any pod that does tolerate the taint will never be evicted, unless the pod uses the **tolerationSeconds** parameter.

The **tolerationSeconds** parameter allows you to specify how long a pod stays bound to a node that has a node condition. If the condition still exists after the **tolerationSeconds** period, the taint remains on

the node and the pods with a matching toleration are evicted. If the condition clears before the **tolerationSeconds** period, pods with matching tolerations are not removed.

If you use the **tolerationSeconds** parameter with no value, pods are never evicted because of the not ready and unreachable node conditions.

> **NOTE**
>
> OpenShift Container Platform evicts pods in a rate-limited way to prevent massive pod evictions in scenarios such as the master becoming partitioned from the nodes.
>
> By default, if more than 55% of nodes in a given zone are unhealthy, the node lifecycle controller changes that zone's state to **PartialDisruption** and the rate of pod evictions is reduced. For small clusters (by default, 50 nodes or less) in this state, nodes in this zone are not tainted and evictions are stopped.
>
> For more information, see Rate limits on eviction in the Kubernetes documentation.

OpenShift Container Platform automatically adds a toleration for **node.kubernetes.io/not-ready** and **node.kubernetes.io/unreachable** with **tolerationSeconds=300**, unless the **Pod** configuration specifies either toleration.

```
spec:
  tolerations:
  - key: node.kubernetes.io/not-ready
    operator: Exists
    effect: NoExecute
    tolerationSeconds: 300  1
  - key: node.kubernetes.io/unreachable
    operator: Exists
    effect: NoExecute
    tolerationSeconds: 300
```

**1** These tolerations ensure that the default pod behavior is to remain bound for five minutes after one of these node conditions problems is detected.

You can configure these tolerations as needed. For example, if you have an application with a lot of local state, you might want to keep the pods bound to node for a longer time in the event of network partition, allowing for the partition to recover and avoiding pod eviction.

Pods spawned by a daemon set are created with **NoExecute** tolerations for the following taints with no **tolerationSeconds**:

- **node.kubernetes.io/unreachable**

- **node.kubernetes.io/not-ready**

As a result, daemon set pods are never evicted because of these node conditions.

### 3.7.1.5. Tolerating all taints

You can configure a pod to tolerate all taints by adding an **operator: "Exists"** toleration with no **key** and **value** parameters. Pods with this toleration are not removed from a node that has taints.

**Pod spec for tolerating all taints**

```
spec:
  tolerations:
  - operator: "Exists"
```

## 3.7.2. Adding taints and tolerations

You add tolerations to pods and taints to nodes to allow the node to control which pods should or should not be scheduled on them. For existing pods and nodes, you should add the toleration to the pod first, then add the taint to the node to avoid pods being removed from the node before you can add the toleration.

**Procedure**

1. Add a toleration to a pod by editing the **Pod** spec to include a **tolerations** stanza:

   **Sample pod configuration file with an Equal operator**

   ```
   spec:
     tolerations:
     - key: "key1"    1
       value: "value1"
       operator: "Equal"
       effect: "NoExecute"
       tolerationSeconds: 3600    2
   ```

   **1** The toleration parameters, as described in the **Taint and toleration components** table.

   **2** The **tolerationSeconds** parameter specifies how long a pod can remain bound to a node before being evicted.

   For example:

   **Sample pod configuration file with an Exists operator**

   ```
   spec:
     tolerations:
     - key: "key1"
       operator: "Exists"    1
       effect: "NoExecute"
       tolerationSeconds: 3600
   ```

   **1** The **Exists** operator does not take a **value**.

   This example places a taint on **node1** that has key **key1**, value **value1**, and taint effect **NoExecute**.

2. Add a taint to a node by using the following command with the parameters described in the **Taint and toleration components** table:

   ```
   $ oc adm taint nodes <node_name> <key>=<value>:<effect>
   ```

For example:

```
$ oc adm taint nodes node1 key1=value1:NoExecute
```

This command places a taint on **node1** that has key **key1**, value **value1**, and effect **NoExecute**.

> **NOTE**
>
> If you add a **NoSchedule** taint to a control plane node (also known as the master node) the node must have the **node-role.kubernetes.io/master=:NoSchedule** taint, which is added by default.
>
> For example:
>
> ```
> apiVersion: v1
> kind: Node
> metadata:
>   annotations:
>     machine.openshift.io/machine: openshift-machine-api/ci-ln-62s7gtb-f76d1-v8jxv-master-0
>     machineconfiguration.openshift.io/currentConfig: rendered-master-cdc1ab7da414629332cc4c3926e6e59c
> ...
> spec:
>   taints:
>   - effect: NoSchedule
>     key: node-role.kubernetes.io/master
> ...
> ```

The tolerations on the pod match the taint on the node. A pod with either toleration can be scheduled onto **node1**.

### 3.7.2.1. Adding taints and tolerations using a machine set

You can add taints to nodes using a machine set. All nodes associated with the **MachineSet** object are updated with the taint. Tolerations respond to taints added by a machine set in the same manner as taints added directly to the nodes.

**Procedure**

1. Add a toleration to a pod by editing the **Pod** spec to include a **tolerations** stanza:

   **Sample pod configuration file with Equal operator**

   ```
   spec:
     tolerations:
     - key: "key1" 1
       value: "value1"
       operator: "Equal"
       effect: "NoExecute"
       tolerationSeconds: 3600 2
   ```

**1**    The toleration parameters, as described in the **Taint and toleration components** table.

**2**    The **tolerationSeconds** parameter specifies how long a pod is bound to a node before being evicted.

For example:

**Sample pod configuration file with Exists operator**

```
spec:
 tolerations:
 - key: "key1"
   operator: "Exists"
   effect: "NoExecute"
   tolerationSeconds: 3600
```

2. Add the taint to the **MachineSet** object:

   a. Edit the **MachineSet** YAML for the nodes you want to taint or you can create a new **MachineSet** object:

   ```
   $ oc edit machineset <machineset>
   ```

   b. Add the taint to the **spec.template.spec** section:

   **Example taint in a machine set specification**

   ```
   spec:
   ....
     template:
   ....
       spec:
         taints:
         - effect: NoExecute
           key: key1
           value: value1
   ....
   ```

   This example places a taint that has the key **key1**, value **value1**, and taint effect **NoExecute** on the nodes.

   c. Scale down the machine set to 0:

   ```
   $ oc scale --replicas=0 machineset <machineset> -n openshift-machine-api
   ```

TIP

You can alternatively apply the following YAML to scale the machine set:

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
  name: <machineset>
  namespace: openshift-machine-api
spec:
  replicas: 0
```

Wait for the machines to be removed.

d. Scale up the machine set as needed:

```
$ oc scale --replicas=2 machineset <machineset> -n openshift-machine-api
```

Or:

```
$ oc edit machineset <machineset> -n openshift-machine-api
```

Wait for the machines to start. The taint is added to the nodes associated with the **MachineSet** object.

### 3.7.2.2. Binding a user to a node using taints and tolerations

If you want to dedicate a set of nodes for exclusive use by a particular set of users, add a toleration to their pods. Then, add a corresponding taint to those nodes. The pods with the tolerations are allowed to use the tainted nodes or any other nodes in the cluster.

If you want ensure the pods are scheduled to only those tainted nodes, also add a label to the same set of nodes and add a node affinity to the pods so that the pods can only be scheduled onto nodes with that label.

### Procedure

To configure a node so that users can use only that node:

1. Add a corresponding taint to those nodes:
   For example:

```
$ oc adm taint nodes node1 dedicated=groupName:NoSchedule
```

> **TIP**
>
> You can alternatively apply the following YAML to add the taint:
>
> ```
> kind: Node
> apiVersion: v1
> metadata:
>   name: <node_name>
>   labels:
>     ...
> spec:
>   taints:
>     - key: dedicated
>       value: groupName
>       effect: NoSchedule
> ```

2. Add a toleration to the pods by writing a custom admission controller.

### 3.7.2.3. Creating a project with a node selector and toleration

You can create a project that uses a node selector and toleration, which are set as annotations, to control the placement of pods onto specific nodes. Any subsequent resources created in the project are then scheduled on nodes that have a taint matching the toleration.

**Prerequisites**

- A label for node selection has been added to one or more nodes by using a machine set or editing the node directly.

- A taint has been added to one or more nodes by using a machine set or editing the node directly.

**Procedure**

1. Create a **Project** resource definition, specifying a node selector and toleration in the **metadata.annotations** section:

   **Example project.yaml file**

   ```
   kind: Project
   apiVersion: project.openshift.io/v1
   metadata:
     name: <project_name>     1
     annotations:
       openshift.io/node-selector: '<label>'     2
       scheduler.alpha.kubernetes.io/defaultTolerations: >-
         [{"operator": "Exists", "effect": "NoSchedule", "key":
         "<key_name>"}     3
         ]
   ```

   **1**    The project name.

   **2**    The default node selector label.

**3** The toleration parameters, as described in the **Taint and toleration components** table. This example uses the **NoSchedule** effect, which allows existing pods on the node to

2. Use the **oc apply** command to create the project:

```
$ oc apply -f project.yaml
```

Any subsequent resources created in the **<project_name>** namespace should now be scheduled on the specified nodes.

**Additional resources**

- Adding taints and tolerations manually to nodes or with machine sets

- Creating project-wide node selectors

- Pod placement of Operator workloads

### 3.7.2.4. Controlling nodes with special hardware using taints and tolerations

In a cluster where a small subset of nodes have specialized hardware, you can use taints and tolerations to keep pods that do not need the specialized hardware off of those nodes, leaving the nodes for pods that do need the specialized hardware. You can also require pods that need specialized hardware to use specific nodes.

You can achieve this by adding a toleration to pods that need the special hardware and tainting the nodes that have the specialized hardware.

#### Procedure

To ensure nodes with specialized hardware are reserved for specific pods:

1. Add a toleration to pods that need the special hardware.
   For example:

   ```
   spec:
     tolerations:
       - key: "disktype"
         value: "ssd"
         operator: "Equal"
         effect: "NoSchedule"
         tolerationSeconds: 3600
   ```

2. Taint the nodes that have the specialized hardware using one of the following commands:

   ```
   $ oc adm taint nodes <node-name> disktype=ssd:NoSchedule
   ```

   Or:

   ```
   $ oc adm taint nodes <node-name> disktype=ssd:PreferNoSchedule
   ```

TIP

You can alternatively apply the following YAML to add the taint:

```
kind: Node
apiVersion: v1
metadata:
  name: <node_name>
  labels:
    ...
spec:
  taints:
    - key: disktype
      value: ssd
      effect: PreferNoSchedule
```

### 3.7.3. Removing taints and tolerations

You can remove taints from nodes and tolerations from pods as needed. You should add the toleration to the pod first, then add the taint to the node to avoid pods being removed from the node before you can add the toleration.

#### Procedure

To remove taints and tolerations:

1. To remove a taint from a node:

   ```
   $ oc adm taint nodes <node-name> <key>-
   ```

   For example:

   ```
   $ oc adm taint nodes ip-10-0-132-248.ec2.internal key1-
   ```

   #### Example output

   ```
   node/ip-10-0-132-248.ec2.internal untainted
   ```

2. To remove a toleration from a pod, edit the **Pod** spec to remove the toleration:

   ```
   spec:
     tolerations:
     - key: "key2"
       operator: "Exists"
       effect: "NoExecute"
       tolerationSeconds: 3600
   ```

## 3.8. PLACING PODS ON SPECIFIC NODES USING NODE SELECTORS

A *node selector* specifies a map of key/value pairs that are defined using custom labels on nodes and selectors specified in pods.

For the pod to be eligible to run on a node, the pod must have the same key/value node selector as the label on the node.

## 3.8.1. About node selectors

You can use node selectors on pods and labels on nodes to control where the pod is scheduled. With node selectors, OpenShift Container Platform schedules the pods on nodes that contain matching labels.

You can use a node selector to place specific pods on specific nodes, cluster-wide node selectors to place new pods on specific nodes anywhere in the cluster, and project node selectors to place new pods in a project on specific nodes.

For example, as a cluster administrator, you can create an infrastructure where application developers can deploy pods only onto the nodes closest to their geographical location by including a node selector in every pod they create. In this example, the cluster consists of five data centers spread across two regions. In the U.S., label the nodes as **us-east**, **us-central**, or **us-west**. In the Asia-Pacific region (APAC), label the nodes as **apac-east** or **apac-west**. The developers can add a node selector to the pods they create to ensure the pods get scheduled on those nodes.

A pod is not scheduled if the **Pod** object contains a node selector, but no node has a matching label.

> **IMPORTANT**
>
> If you are using node selectors and node affinity in the same pod configuration, the following rules control pod placement onto nodes:
>
> - If you configure both **nodeSelector** and **nodeAffinity**, both conditions must be satisfied for the pod to be scheduled onto a candidate node.
>
> - If you specify multiple **nodeSelectorTerms** associated with **nodeAffinity** types, then the pod can be scheduled onto a node if one of the **nodeSelectorTerms** is satisfied.
>
> - If you specify multiple **matchExpressions** associated with **nodeSelectorTerms**, then the pod can be scheduled onto a node only if all **matchExpressions** are satisfied.

### Node selectors on specific pods and nodes

You can control which node a specific pod is scheduled on by using node selectors and labels.
To use node selectors and labels, first label the node to avoid pods being descheduled, then add the node selector to the pod.
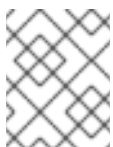
> **NOTE**
>
> You cannot add a node selector directly to an existing scheduled pod. You must label the object that controls the pod, such as deployment config.

For example, the following **Node** object has the **region: east** label:

### Sample **Node** object with a label

```
kind: Node
apiVersion: v1
```

```
metadata:
  name: ip-10-0-131-14.ec2.internal
  selfLink: /api/v1/nodes/ip-10-0-131-14.ec2.internal
  uid: 7bc2580a-8b8e-11e9-8e01-021ab4174c74
  resourceVersion: '478704'
  creationTimestamp: '2019-06-10T14:46:08Z'
  labels:
    kubernetes.io/os: linux
    failure-domain.beta.kubernetes.io/zone: us-east-1a
    node.openshift.io/os_version: '4.5'
    node-role.kubernetes.io/worker: ''
    failure-domain.beta.kubernetes.io/region: us-east-1
    node.openshift.io/os_id: rhcos
    beta.kubernetes.io/instance-type: m4.large
    kubernetes.io/hostname: ip-10-0-131-14
    beta.kubernetes.io/arch: amd64
    region: east ❶
```

❶ Label to match the pod node selector.

A pod has the **type: user-node,region: east** node selector:

**Sample Pod object with node selectors**

```
apiVersion: v1
kind: Pod

....

spec:
  nodeSelector: ❶
    region: east
    type: user-node
```

❶ Node selectors to match the node label.

When you create the pod using the example pod spec, it can be scheduled on the example node.

**Default cluster-wide node selectors**

With default cluster-wide node selectors, when you create a pod in that cluster, OpenShift Container Platform adds the default node selectors to the pod and schedules the pod on nodes with matching labels.
For example, the following **Scheduler** object has the default cluster-wide **region=east** and **type=user-node** node selectors:

**Example Scheduler Operator Custom Resource**

```
apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  name: cluster
...
```

```
spec:
  defaultNodeSelector: type=user-node,region=east
...
```

A node in that cluster has the **type=user-node,region=east** labels:

### Example **Node** object

```
apiVersion: v1
kind: Node
metadata:
  name: ci-ln-qg1il3k-f76d1-hlmhl-worker-b-df2s4
...
  labels:
    region: east
    type: user-node
...
```

### Example **Pod** object with a node selector

```
apiVersion: v1
kind: Pod
...

spec:
  nodeSelector:
    region: east
...
```

When you create the pod using the example pod spec in the example cluster, the pod is created with the cluster-wide node selector and is scheduled on the labeled node:

### Example pod list with the pod on the labeled node

```
NAME    READY  STATUS   RESTARTS  AGE  IP         NODE
NOMINATED NODE   READINESS GATES
pod-s1  1/1    Running  0         20s  10.131.2.6  ci-ln-qg1il3k-f76d1-hlmhl-worker-b-df2s4
<none>          <none>
```

> **NOTE**
>
> If the project where you create the pod has a project node selector, that selector takes preference over a cluster-wide node selector. Your pod is not created or scheduled if the pod does not have the project node selector.

### Project node selectors

With project node selectors, when you create a pod in this project, OpenShift Container Platform adds the node selectors to the pod and schedules the pods on a node with matching labels. If there is a cluster-wide default node selector, a project node selector takes preference.

For example, the following project has the **region=east** node selector:

### Example **Namespace** object

```
apiVersion: v1
kind: Namespace
metadata:
  name: east-region
  annotations:
    openshift.io/node-selector: "region=east"
...
```

The following node has the **type=user-node,region=east** labels:

### Example **Node** object

```
apiVersion: v1
kind: Node
metadata:
  name: ci-ln-qg1il3k-f76d1-hlmhl-worker-b-df2s4
...
  labels:
    region: east
    type: user-node
...
```

When you create the pod using the example pod spec in this example project, the pod is created with the project node selectors and is scheduled on the labeled node:

### Example **Pod** object

```
apiVersion: v1
kind: Pod
metadata:
  namespace: east-region
...
spec:
  nodeSelector:
    region: east
    type: user-node
...
```

### Example pod list with the pod on the labeled node

```
NAME    READY  STATUS   RESTARTS  AGE  IP         NODE
NOMINATED NODE   READINESS GATES
pod-s1  1/1    Running  0         20s  10.131.2.6  ci-ln-qg1il3k-f76d1-hlmhl-worker-b-df2s4
<none>          <none>
```

A pod in the project is not created or scheduled if the pod contains different node selectors. For example, if you deploy the following pod into the example project, it is not be created:

### Example **Pod** object with an invalid node selector

```
apiVersion: v1
kind: Pod
...
```

```
spec:
  nodeSelector:
    region: west

....
```

## 3.8.2. Using node selectors to control pod placement

You can use node selectors on pods and labels on nodes to control where the pod is scheduled. With node selectors, OpenShift Container Platform schedules the pods on nodes that contain matching labels.

You add labels to a node, a machine set, or a machine config. Adding the label to the machine set ensures that if the node or machine goes down, new nodes have the label. Labels added to a node or machine config do not persist if the node or machine goes down.

To add node selectors to an existing pod, add a node selector to the controlling object for that pod, such as a **ReplicaSet** object, **DaemonSet** object, **StatefulSet** object, **Deployment** object, or **DeploymentConfig** object. Any existing pods under that controlling object are recreated on a node with a matching label. If you are creating a new pod, you can add the node selector directly to the **Pod** spec.

> **NOTE**
>
> You cannot add a node selector directly to an existing scheduled pod.

### Prerequisites

To add a node selector to existing pods, determine the controlling object for that pod. For example, the **router-default-66d5cf9464-m2g75** pod is controlled by the **router-default-66d5cf9464** replica set:

```
$ oc describe pod router-default-66d5cf9464-7pwkc

Name:           router-default-66d5cf9464-7pwkc
Namespace:       openshift-ingress

....

Controlled By:     ReplicaSet/router-default-66d5cf9464
```

The web console lists the controlling object under **ownerReferences** in the pod YAML:

```
ownerReferences:
  - apiVersion: apps/v1
    kind: ReplicaSet
    name: router-default-66d5cf9464
    uid: d81dd094-da26-11e9-a48a-128e7edf0312
    controller: true
    blockOwnerDeletion: true
```

### Procedure

1. Add labels to a node by using a machine set or editing the node directly:

- Use a **MachineSet** object to add labels to nodes managed by the machine set when a node is created:

  a. Run the following command to add labels to a **MachineSet** object:

  ```
  $ oc patch MachineSet <name> --type='json' -
  p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"<key>"="
  <value>","<key>"="<value>"}}]'  -n openshift-machine-api
  ```

  For example:

  ```
  $ oc patch MachineSet abc612-msrtw-worker-us-east-1c  --type='json' -
  p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"type":"user-
  node","region":"east"}}]'  -n openshift-machine-api
  ```

  **TIP**

  You can alternatively apply the following YAML to add labels to a machine set:

  ```
  apiVersion: machine.openshift.io/v1beta1
  kind: MachineSet
  metadata:
    name: <machineset>
    namespace: openshift-machine-api
  spec:
    template:
      spec:
        metadata:
          labels:
            region: "east"
            type: "user-node"
  ```

  b. Verify that the labels are added to the **MachineSet** object by using the  **oc edit** command:
  For example:

  ```
  $ oc edit MachineSet abc612-msrtw-worker-us-east-1c -n openshift-machine-api
  ```

  **Example MachineSet object**

  ```
  apiVersion: machine.openshift.io/v1beta1
  kind: MachineSet

  ....

  spec:
  ...
    template:
      metadata:
  ...
        spec:
          metadata:
            labels:
  ```

```
        region: east
        type: user-node
    ....
```

- Add labels directly to a node:

    a. Edit the **Node** object for the node:

    ```
    $ oc label nodes <name> <key>=<value>
    ```

    For example, to label a node:

    ```
    $ oc label nodes ip-10-0-142-25.ec2.internal type=user-node region=east
    ```

    **TIP**

    You can alternatively apply the following YAML to add labels to a node:

    ```
    kind: Node
    apiVersion: v1
    metadata:
      name: <node_name>
      labels:
        type: "user-node"
        region: "east"
    ```

    b. Verify that the labels are added to the node:

    ```
    $ oc get nodes -l type=user-node,region=east
    ```

    **Example output**

    ```
    NAME                      STATUS   ROLES    AGE   VERSION
    ip-10-0-142-25.ec2.internal   Ready    worker   17m   v1.18.3+002a51f
    ```

2. Add the matching node selector to a pod:

    - To add a node selector to existing and future pods, add a node selector to the controlling object for the pods:

    **Example ReplicaSet object with labels**

    ```
    kind: ReplicaSet

    ....

    spec:

    ....

      template:
        metadata:
          creationTimestamp: null
    ```

```
      labels:
        ingresscontroller.operator.openshift.io/deployment-ingresscontroller: default
        pod-template-hash: 66d5cf9464
    spec:
      nodeSelector:
        kubernetes.io/os: linux
        node-role.kubernetes.io/worker: "
        type: user-node 1
```

**1**    Add the node selector.

- To add a node selector to a specific, new pod, add the selector to the **Pod** object directly:

  **Example Pod object with a node selector**

  ```
  apiVersion: v1
  kind: Pod

  ....

  spec:
    nodeSelector:
      region: east
      type: user-node
  ```

  > **NOTE**
  >
  > You cannot add a node selector directly to an existing scheduled pod.

### 3.8.3. Creating default cluster-wide node selectors

You can use default cluster-wide node selectors on pods together with labels on nodes to constrain all pods created in a cluster to specific nodes.

With cluster-wide node selectors, when you create a pod in that cluster, OpenShift Container Platform adds the default node selectors to the pod and schedules the pod on nodes with matching labels.

You configure cluster-wide node selectors by editing the Scheduler Operator custom resource (CR). You add labels to a node, a machine set, or a machine config. Adding the label to the machine set ensures that if the node or machine goes down, new nodes have the label. Labels added to a node or machine config do not persist if the node or machine goes down.

> **NOTE**
>
> You can add additional key/value pairs to a pod. But you cannot add a different value for a default key.

### Procedure

To add a default cluster-wide node selector:

1.  Edit the Scheduler Operator CR to add the default cluster-wide node selectors:

```
$ oc edit scheduler cluster
```

**Example Scheduler Operator CR with a node selector**

```
apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  name: cluster
...

spec:
  defaultNodeSelector: type=user-node,region=east 1
  mastersSchedulable: false
  policy:
    name: ""
```

**1**    Add a node selector with the appropriate **<key>:<value>** pairs.

After making this change, wait for the pods in the **openshift-kube-apiserver** project to redeploy. This can take several minutes. The default cluster-wide node selector does not take effect until the pods redeploy.

2. Add labels to a node by using a machine set or editing the node directly:

   - Use a machine set to add labels to nodes managed by the machine set when a node is created:

     a. Run the following command to add labels to a **MachineSet** object:

        ```
        $ oc patch MachineSet <name> --type='json' -
        p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"<key>"="
        <value>","<key>"="<value>"}}]'  -n openshift-machine-api 1
        ```

        **1**    Add a **<key>/<value>** pair for each label.

        For example:

        ```
        $ oc patch MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c --type='json' -
        p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"type":"user-
        node","region":"east"}}]'  -n openshift-machine-api
        ```

**TIP**

You can alternatively apply the following YAML to add labels to a machine set:

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
  name: <machineset>
  namespace: openshift-machine-api
spec:
  template:
    spec:
      metadata:
        labels:
          region: "east"
          type: "user-node"
```

b. Verify that the labels are added to the **MachineSet** object by using the **oc edit** command:
For example:

```
$ oc edit MachineSet abc612-msrtw-worker-us-east-1c -n openshift-machine-api
```

**Example MachineSet object**

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
  ...
spec:
  ...
  template:
    metadata:
  ...
    spec:
      metadata:
        labels:
          region: east
          type: user-node
  ...
```

c. Redeploy the nodes associated with that machine set by scaling down to **0** and scaling up the nodes:
For example:

```
$ oc scale --replicas=0 MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

```
$ oc scale --replicas=1 MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

d. When the nodes are ready and available, verify that the label is added to the nodes by using the **oc get** command:

```
$ oc get nodes -l <key>=<value>
```

For example:

```
$ oc get nodes -l type=user-node
```

**Example output**

```
NAME                                STATUS   ROLES    AGE   VERSION
ci-ln-l8nry52-f76d1-hl7m7-worker-c-vmqzp  Ready    worker   61s   v1.18.3+002a51f
```

- Add labels directly to a node:

  a. Edit the **Node** object for the node:

  ```
  $ oc label nodes <name> <key>=<value>
  ```

  For example, to label a node:

  ```
  $ oc label nodes ci-ln-l8nry52-f76d1-hl7m7-worker-b-tgq49 type=user-node
  region=east
  ```

  **TIP**

  You can alternatively apply the following YAML to add labels to a node:

  ```
  kind: Node
  apiVersion: v1
  metadata:
    name: <node_name>
    labels:
      type: "user-node"
      region: "east"
  ```

  b. Verify that the labels are added to the node using the **oc get** command:

  ```
  $ oc get nodes -l <key>=<value>,<key>=<value>
  ```

  For example:

  ```
  $ oc get nodes -l type=user-node,region=east
  ```

  **Example output**

  ```
  NAME                                STATUS   ROLES    AGE   VERSION
  ci-ln-l8nry52-f76d1-hl7m7-worker-b-tgq49  Ready    worker   17m   v1.18.3+002a51f
  ```

## 3.8.4. Creating project-wide node selectors

You can use node selectors in a project together with labels on nodes to constrain all pods created in that project to the labeled nodes.

When you create a pod in this project, OpenShift Container Platform adds the node selectors to the pods in the project and schedules the pods on a node with matching labels in the project. If there is a cluster-wide default node selector, a project node selector takes preference.

You add node selectors to a project by editing the **Namespace** object to add the **openshift.io/node-selector** parameter. You add labels to a node, a machine set, or a machine config. Adding the label to the machine set ensures that if the node or machine goes down, new nodes have the label. Labels added to a node or machine config do not persist if the node or machine goes down.

A pod is not scheduled if the **Pod** object contains a node selector, but no project has a matching node selector. When you create a pod from that spec, you receive an error similar to the following message:

**Example error message**

> Error from server (Forbidden): error when creating "pod.yaml": pods "pod-4" is forbidden: pod node label selector conflicts with its project node label selector

**NOTE**

You can add additional key/value pairs to a pod. But you cannot add a different value for a project key.

**Procedure**

To add a default project node selector:

1. Create a namespace or edit an existing namespace to add the **openshift.io/node-selector** parameter:

   ```
   $ oc edit namespace <name>
   ```

   **Example output**

   ```
   apiVersion: v1
   kind: Namespace
   metadata:
     annotations:
       openshift.io/node-selector: "type=user-node,region=east"    1
       openshift.io/description: ""
       openshift.io/display-name: ""
       openshift.io/requester: kube:admin
       openshift.io/sa.scc.mcs: s0:c30,c5
       openshift.io/sa.scc.supplemental-groups: 1000880000/10000
       openshift.io/sa.scc.uid-range: 1000880000/10000
     creationTimestamp: "2021-05-10T12:35:04Z"
     labels:
       kubernetes.io/metadata.name: demo
     name: demo
     resourceVersion: "145537"
     uid: 3f8786e3-1fcb-42e3-a0e3-e2ac54d15001
   ```

```
spec:
  finalizers:
  - kubernetes
```

 Add the **openshift.io/node-selector** with the appropriate **<key>:<value>** pairs.

2. Add labels to a node by using a machine set or editing the node directly:

   - Use a **MachineSet** object to add labels to nodes managed by the machine set when a node is created:

     a. Run the following command to add labels to a **MachineSet** object:

     ```
     $ oc patch MachineSet <name> --type='json' -
     p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"<key>"="
     <value>","<key>"="<value>"}}]'  -n openshift-machine-api
     ```

     For example:

     ```
     $ oc patch MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c --type='json' -
     p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"type":"user-
     node","region":"east"}}]'  -n openshift-machine-api
     ```

     **TIP**

     You can alternatively apply the following YAML to add labels to a machine set:

     ```
     apiVersion: machine.openshift.io/v1beta1
     kind: MachineSet
     metadata:
       name: <machineset>
       namespace: openshift-machine-api
     spec:
       template:
         spec:
           metadata:
             labels:
               region: "east"
               type: "user-node"
     ```

     b. Verify that the labels are added to the **MachineSet** object by using the **oc edit** command:
        For example:

     ```
     $ oc edit MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
     ```

     **Example output**

     ```
     apiVersion: machine.openshift.io/v1beta1
     kind: MachineSet
     metadata:
       ...
     ```

```
spec:
...
  template:
    metadata:
...
    spec:
      metadata:
        labels:
          region: east
          type: user-node
```

c. Redeploy the nodes associated with that machine set:
For example:

```
$ oc scale --replicas=0 MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

```
$ oc scale --replicas=1 MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

d. When the nodes are ready and available, verify that the label is added to the nodes by using the **oc get** command:

```
$ oc get nodes -l <key>=<value>
```

For example:

```
$ oc get nodes -l type=user-node,region=east
```

**Example output**

```
NAME                                 STATUS   ROLES    AGE   VERSION
ci-ln-l8nry52-f76d1-hl7m7-worker-c-vmqzp   Ready    worker   61s   v1.18.3+002a51f
```

- Add labels directly to a node:

  a. Edit the **Node** object to add labels:

  ```
  $ oc label <resource> <name> <key>=<value>
  ```

  For example, to label a node:

  ```
  $ oc label nodes ci-ln-l8nry52-f76d1-hl7m7-worker-c-tgq49 type=user-node region=east
  ```

TIP

You can alternatively apply the following YAML to add labels to a node:

```
kind: Node
apiVersion: v1
metadata:
  name: <node_name>
  labels:
    type: "user-node"
    region: "east"
```

b. Verify that the labels are added to the **Node** object using the **oc get** command:

```
$ oc get nodes -l <key>=<value>
```

For example:

```
$ oc get nodes -l type=user-node,region=east
```

**Example output**

```
NAME                                STATUS   ROLES    AGE   VERSION
ci-ln-l8nry52-f76d1-hl7m7-worker-b-tgq49  Ready    worker   17m   v1.18.3+002a51f
```

**Additional resources**

- [Creating a project with a node selector and toleration](#)

## 3.9. CONTROLLING POD PLACEMENT BY USING POD TOPOLOGY SPREAD CONSTRAINTS

You can use pod topology spread constraints to control the placement of your pods across nodes, zones, regions, or other user-defined topology domains.

### 3.9.1. About pod topology spread constraints

By using a *pod topology spread constraint*, you provide fine-grained control over the distribution of pods across failure domains to help achieve high availability and more efficient resource utilization.

OpenShift Container Platform administrators can label nodes to provide topology information, such as regions, zones, nodes, or other user-defined domains. After these labels are set on nodes, users can then define pod topology spread constraints to control the placement of pods across these topology domains.

You specify which pods to group together, which topology domains they are spread among, and the acceptable skew. Only pods within the same namespace are matched and grouped together when spreading due to a constraint.

### 3.9.2. Configuring pod topology spread constraints

The following steps demonstrate how to configure pod topology spread constraints to distribute pods that match the specified labels based on their zone.

You can specify multiple pod topology spread constraints, but you must ensure that they do not conflict with each other. All pod topology spread constraints must be satisfied for a pod to be placed.

### Prerequisites

- A cluster administrator has added the required labels to nodes.

### Procedure

1. Create a **Pod** spec and specify a pod topology spread constraint:

   Example **pod-spec.yaml** file

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: my-pod
     labels:
       foo: bar
   spec:
     topologySpreadConstraints:
     - maxSkew: 1          1
       topologyKey: topology.kubernetes.io/zone    2
       whenUnsatisfiable: DoNotSchedule    3
       labelSelector:       4
         matchLabels:
           foo: bar       5
     containers:
     - image: "docker.io/ocpqe/hello-pod"
       name: hello-pod
   ```

   **1** The maximum difference in number of pods between any two topology domains. The default is **1**, and you cannot specify a value of **0**.

   **2** The key of a node label. Nodes with this key and identical value are considered to be in the same topology.

   **3** How to handle a pod if it does not satisfy the spread constraint. The default is **DoNotSchedule**, which tells the scheduler not to schedule the pod. Set to **ScheduleAnyway** to still schedule the pod, but the scheduler prioritizes honoring the skew to not make the cluster more imbalanced.

   **4** Pods that match this label selector are counted and recognized as a group when spreading to satisfy the constraint. Be sure to specify a label selector, otherwise no pods can be matched.

   **5** Be sure that this **Pod** spec also sets its labels to match this label selector if you want it to be counted properly in the future.

2. Create the pod:

```
$ oc create -f pod-spec.yaml
```

### 3.9.3. Example pod topology spread constraints

The following examples demonstrate pod topology spread constraint configurations.

#### 3.9.3.1. Single pod topology spread constraint example

This example **Pod** spec defines one pod topology spread constraint. It matches on pods labeled **foo:bar**, distributes among zones, specifies a skew of **1**, and does not schedule the pod if it does not meet these requirements.

```
kind: Pod
apiVersion: v1
metadata:
  name: my-pod
  labels:
    foo: bar
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: topology.kubernetes.io/zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  containers:
  - image: "docker.io/ocpqe/hello-pod"
    name: hello-pod
```

#### 3.9.3.2. Multiple pod topology spread constraints example

This example **Pod** spec defines two pod topology spread constraints. Both match on pods labeled **foo:bar**, specify a skew of **1**, and do not schedule the pod if it does not meet these requirements.

The first constraint distributes pods based on a user-defined label **node**, and the second constraint distributes pods based on a user-defined label **rack**. Both constraints must be met for the pod to be scheduled.

```
kind: Pod
apiVersion: v1
metadata:
  name: my-pod-2
  labels:
    foo: bar
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: node
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  - maxSkew: 1
```

```
    topologyKey: rack
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
  containers:
  - image: "docker.io/ocpqe/hello-pod"
    name: hello-pod
```

### 3.9.4. Additional resources

- [Understanding how to update labels on nodes](#)

## 3.10. RUNNING A CUSTOM SCHEDULER

You can run multiple custom schedulers alongside the default scheduler and configure which scheduler to use for each pod.
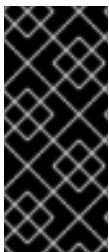
> **IMPORTANT**
>
> It is supported to use a custom scheduler with OpenShift Container Platform, but Red Hat does not directly support the functionality of the custom scheduler.
>
> For information on how to configure the default scheduler, see [Configuring the default scheduler to control pod placement](#).

To schedule a given pod using a specific scheduler, [specify the name of the scheduler in that **Pod** specification](#).

### 3.10.1. Deploying a custom scheduler

To include a custom scheduler in your cluster, include the image for a custom scheduler in a deployment.

**Prerequisites**

- You have access to the cluster as a user with the **cluster-admin** role.

- You have a scheduler binary.

> **NOTE**
>
> Information on how to create a scheduler binary is outside the scope of this document. For an example, see [Configure Multiple Schedulers](#) in the Kubernetes documentation. Note that the actual functionality of your custom scheduler is not supported by Red Hat.

- You have created an image containing the scheduler binary and pushed it to a registry.

**Procedure**

1. Create a file that contains a config map that holds the scheduler configuration file:

   Example **scheduler-config-map.yaml**

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: scheduler-config
  namespace: kube-system ❶
data:
  scheduler-config.yaml: | ❷
    apiVersion: kubescheduler.config.k8s.io/v1beta2
    kind: KubeSchedulerConfiguration
    profiles:
      - schedulerName: custom-scheduler ❸
    leaderElection:
      leaderElect: false
```

❶ This procedure uses the **kube-system** namespace, but you can use the namespace of your choosing.

❷ When you define your **Deployment** resource later in this procedure, you pass this file in to the scheduler command by using the **--config** argument.

❸ Define a scheduler profile for your custom scheduler. This scheduler name is used when defining the **schedulerName** in the **Pod** configuration.

2. Create the config map:

```
$ oc create -f scheduler-config-map.yaml
```

3. Create a file that contains the deployment resources for the custom scheduler:

   Example **custom-scheduler.yaml** file

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: custom-scheduler
  namespace: kube-system ❶
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: custom-scheduler-as-kube-scheduler
subjects:
- kind: ServiceAccount
  name: custom-scheduler
  namespace: kube-system ❷
roleRef:
  kind: ClusterRole
  name: system:kube-scheduler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
```

```
    name: custom-scheduler-as-volume-scheduler
subjects:
- kind: ServiceAccount
  name: custom-scheduler
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: system:volume-scheduler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    component: scheduler
    tier: control-plane
  name: custom-scheduler
  namespace: kube-system
spec:
  selector:
    matchLabels:
      component: scheduler
      tier: control-plane
  replicas: 1
  template:
    metadata:
      labels:
        component: scheduler
        tier: control-plane
        version: second
    spec:
      serviceAccountName: custom-scheduler
      containers:
      - command:
        - /usr/local/bin/kube-scheduler
        - --config=/etc/config/scheduler-config.yaml
        image: "<namespace>/<image_name>:<tag>"
        livenessProbe:
          httpGet:
            path: /healthz
            port: 10259
            scheme: HTTPS
          initialDelaySeconds: 15
        name: kube-second-scheduler
        readinessProbe:
          httpGet:
            path: /healthz
            port: 10259
            scheme: HTTPS
        resources:
          requests:
            cpu: '0.1'
        securityContext:
          privileged: false
        volumeMounts:
        - name: config-volume
```

The numbered callouts **3**, **4**, **5**, **6** appear next to `namespace: kube-system`, `namespace: kube-system`, `- --config=/etc/config/scheduler-config.yaml`, and `image: "<namespace>/<image_name>:<tag>"` respectively.

```
      mountPath: /etc/config
    hostNetwork: false
    hostPID: false
    volumes:
      - name: config-volume
        configMap:
          name: scheduler-config
```

**1** **2** **3** **4** This procedure uses the **kube-system** namespace, but you can use the namespace of your choosing.

**5** The command for your custom scheduler might require different arguments.

**6** Specify the container image that you created for the custom scheduler.

4. Create the deployment resources in the cluster:

```
$ oc create -f custom-scheduler.yaml
```

**Verification**

- Verify that the scheduler pod is running:

```
$ oc get pods -n kube-system
```

The custom scheduler pod is listed as **Running**:

```
NAME                              READY  STATUS   RESTARTS  AGE
custom-scheduler-6cd7c4b8bc-854zb            1/1    Running  0         2m
```

## 3.10.2. Deploying pods using a custom scheduler

After the custom scheduler is deployed in your cluster, you can configure pods to use that scheduler instead of the default scheduler.

> **NOTE**
>
> Each scheduler has a separate view of resources in a cluster. For that reason, each scheduler should operate over its own set of nodes.
>
> If two or more schedulers operate on the same node, they might intervene with each other and schedule more pods on the same node than there are available resources for. Pods might get rejected due to insufficient resources in this case.

**Prerequisites**

- You have access to the cluster as a user with the **cluster-admin** role.

- The custom scheduler has been deployed in the cluster.

**Procedure**

1. If your cluster uses role-based access control (RBAC), add the custom scheduler name to the **system:kube-scheduler** cluster role.

   a. Edit the **system:kube-scheduler** cluster role:

   ```
   $ oc edit clusterrole system:kube-scheduler
   ```

   b. Add the name of the custom scheduler to the **resourceNames** lists for the **leases** and **endpoints** resources:

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   kind: ClusterRole
   metadata:
     annotations:
       rbac.authorization.kubernetes.io/autoupdate: "true"
     creationTimestamp: "2021-07-07T10:19:14Z"
     labels:
       kubernetes.io/bootstrapping: rbac-defaults
     name: system:kube-scheduler
     resourceVersion: "125"
     uid: 53896c70-b332-420a-b2a4-f72c822313f2
   rules:
   ...
   - apiGroups:
     - coordination.k8s.io
     resources:
     - leases
     verbs:
     - create
   - apiGroups:
     - coordination.k8s.io
     resourceNames:
     - kube-scheduler
     - custom-scheduler        1
     resources:
     - leases
     verbs:
     - get
     - update
   - apiGroups:
     - ""
     resources:
     - endpoints
     verbs:
     - create
   - apiGroups:
     - ""
     resourceNames:
     - kube-scheduler
     - custom-scheduler        2
     resources:
     - endpoints
     verbs:
     - get
     - update
   ...
   ```

**1** **2** This example uses **custom-scheduler** as the custom scheduler name.

2. Create a **Pod** configuration and specify the name of the custom scheduler in the **schedulerName** parameter:

   **Example custom-scheduler-example.yaml file**

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: custom-scheduler-example
     labels:
       name: custom-scheduler-example
   spec:
     schedulerName: custom-scheduler     1
     containers:
     - name: pod-with-second-annotation-container
       image: docker.io/ocpqe/hello-pod
   ```

   **1** The name of the custom scheduler to use, which is **custom-scheduler** in this example. When no scheduler name is supplied, the pod is automatically scheduled using the default scheduler.

3. Create the pod:

   ```
   $ oc create -f custom-scheduler-example.yaml
   ```

**Verification**

1. Enter the following command to check that the pod was created:

   ```
   $ oc get pod custom-scheduler-example
   ```

   The **custom-scheduler-example** pod is listed in the output:

   ```
   NAME                      READY   STATUS    RESTARTS   AGE
   custom-scheduler-example  1/1     Running   0          4m
   ```

2. Enter the following command to check that the custom scheduler has scheduled the pod:

   ```
   $ oc describe pod custom-scheduler-example
   ```

   The scheduler, **custom-scheduler**, is listed as shown in the following truncated output:

   ```
   Events:
    Type    Reason     Age        From                Message
    ----    ------     ----       ----                -------
    Normal  Scheduled  <unknown>  custom-scheduler    Successfully
   assigned default/custom-scheduler-example to <node_name>
   ```

### 3.10.3. Additional resources

- [Learning container best practices](#)

## 3.11. EVICTING PODS USING THE DESCHEDULER

While the scheduler is used to determine the most suitable node to host a new pod, the descheduler can be used to evict a running pod so that the pod can be rescheduled onto a more suitable node.

### 3.11.1. About the descheduler

You can use the descheduler to evict pods based on specific strategies so that the pods can be rescheduled onto more appropriate nodes.

You can benefit from descheduling running pods in situations such as the following:

- Nodes are underutilized or overutilized.

- Pod and node affinity requirements, such as taints or labels, have changed and the original scheduling decisions are no longer appropriate for certain nodes.

- Node failure requires pods to be moved.

- New nodes are added to clusters.

- Pods have been restarted too many times.

> **IMPORTANT**
>
> The descheduler does not schedule replacement of evicted pods. The scheduler automatically performs this task for the evicted pods.

When the descheduler decides to evict pods from a node, it employs the following general mechanism:

- Pods in the **openshift-\*** and **kube-system** namespaces are never evicted.

- Critical pods with **priorityClassName** set to **system-cluster-critical** or **system-node-critical** are never evicted.

- Static, mirrored, or stand-alone pods that are not part of a replication controller, replica set, deployment, or job are never evicted because these pods will not be recreated.

- Pods associated with daemon sets are never evicted.

- Pods with local storage are never evicted.

- Best effort pods are evicted before burstable and guaranteed pods.

- All types of pods with the **descheduler.alpha.kubernetes.io/evict** annotation are eligible for eviction. This annotation is used to override checks that prevent eviction, and the user can select which pod is evicted. Users should know how and if the pod will be recreated.

- Pods subject to pod disruption budget (PDB) are not evicted if descheduling violates its pod disruption budget (PDB). The pods are evicted by using eviction subresource to handle PDB.

## 3.11.2. Descheduler profiles

The following descheduler profiles are available:

**AffinityAndTaints**

This profile evicts pods that violate inter-pod anti-affinity, node affinity, and node taints.
It enables the following strategies:

- **RemovePodsViolatingInterPodAntiAffinity**: removes pods that are violating inter-pod anti-affinity.

- **RemovePodsViolatingNodeAffinity**: removes pods that are violating node affinity.

- **RemovePodsViolatingNodeTaints**: removes pods that are violating **NoSchedule** taints on nodes.
  Pods with a node affinity type of **requiredDuringSchedulingIgnoredDuringExecution** are removed.

**TopologyAndDuplicates**

This profile evicts pods in an effort to evenly spread similar pods, or pods of the same topology domain, among nodes.
It enables the following strategies:

- **RemovePodsViolatingTopologySpreadConstraint**: finds unbalanced topology domains and tries to evict pods from larger ones when **DoNotSchedule** constraints are violated.

- **RemoveDuplicates**: ensures that there is only one pod associated with a replica set, replication controller, deployment, or job running on same node. If there are more, those duplicate pods are evicted for better pod distribution in a cluster.

**LifecycleAndUtilization**

This profile evicts long-running pods and balances resource usage between nodes.
It enables the following strategies:

- **RemovePodsHavingTooManyRestarts**: removes pods whose containers have been restarted too many times.
  Pods where the sum of restarts over all containers (including Init Containers) is more than 100.

- **LowNodeUtilization**: finds nodes that are underutilized and evicts pods, if possible, from overutilized nodes in the hope that recreation of evicted pods will be scheduled on these underutilized nodes.
  A node is considered underutilized if its usage is below 20% for all thresholds (CPU, memory, and number of pods).

  A node is considered overutilized if its usage is above 50% for any of the thresholds (CPU, memory, and number of pods).

- **PodLifeTime**: evicts pods that are too old.
  Pods that are older than 24 hours are removed.

## 3.11.3. Installing the descheduler

The descheduler is not available by default. To enable the descheduler, you must install the Kube Descheduler Operator from OperatorHub and enable one or more descheduler profiles.

**Prerequisites**

- Cluster administrator privileges.

- Access to the OpenShift Container Platform web console.

**Procedure**

1. Log in to the OpenShift Container Platform web console.

2. Create the required namespace for the Kube Descheduler Operator.

   a. Navigate to **Administration → Namespaces** and click **Create Namespace**.

   b. Enter **openshift-kube-descheduler-operator** in the **Name** field, enter **openshift.io/cluster-monitoring=true** in the **Labels** field to enable descheduler metrics, and click **Create**.

3. Install the Kube Descheduler Operator.

   a. Navigate to **Operators → OperatorHub**.

   b. Type **Kube Descheduler Operator** into the filter box.

   c. Select the **Kube Descheduler Operator** and click **Install**.

   d. On the **Install Operator** page, select **A specific namespace on the cluster**. Select **openshift-kube-descheduler-operator** from the drop-down menu.

   e. Adjust the values for the **Update Channel** and **Approval Strategy** to the desired values.

   f. Click **Install**.

4. Create a descheduler instance.

   a. From the **Operators → Installed Operators** page, click the **Kube Descheduler Operator**.

   b. Select the **Kube Descheduler** tab and click **Create KubeDescheduler**.

   c. Edit the settings as necessary.

      i. Expand the **Profiles** section to select one or more profiles to enable. The **AffinityAndTaints** profile is enabled by default. Click **Add Profile** to select additional profiles.

      ii. Optional: Use the **Descheduling Interval Seconds** field to change the number of seconds between descheduler runs. The default is **3600** seconds.

   d. Click **Create**.

You can also configure the profiles and settings for the descheduler later using the OpenShift CLI (**oc**). If you did not adjust the profiles when creating the descheduler instance from the web console, the **AffinityAndTaints** profile is enabled by default.

## 3.11.4. Configuring descheduler profiles

You can configure which profiles the descheduler uses to evict pods.

**Prerequisites**

- Cluster administrator privileges

**Procedure**

1. Edit the **KubeDescheduler** object:

   ```
   $ oc edit kubedeschedulers.operator.openshift.io cluster -n openshift-kube-descheduler-operator
   ```

2. Specify one or more profiles in the **spec.profiles** section.

   ```
   apiVersion: operator.openshift.io/v1
   kind: KubeDescheduler
   metadata:
     name: cluster
     namespace: openshift-kube-descheduler-operator
   spec:
     deschedulingIntervalSeconds: 3600
     logLevel: Normal
     managementState: Managed
     operatorLogLevel: Normal
     profiles:
     - AffinityAndTaints          1
     - TopologyAndDuplicates      2
     - LifecycleAndUtilization    3
   ```

   **1** Enable the **AffinityAndTaints** profile, which evicts pods that violate inter-pod anti-affinity, node affinity, and node taints.

   **2** Enable the **TopologyAndDuplicates** profile, which evicts pods in an effort to evenly spread similar pods, or pods of the same topology domain, among nodes.

   **3** Enable the **LifecycleAndUtilization** profile, which evicts long-running pods and balances resource usage between nodes.

   You can enable multiple profiles; the order that the profiles are specified in is not important.

3. Save the file to apply the changes.

## 3.11.5. Configuring the descheduler interval

You can configure the amount of time between descheduler runs. The default is 3600 seconds (one hour).

**Prerequisites**

- Cluster administrator privileges

**Procedure**

1. Edit the **KubeDescheduler** object:

   ```
   $ oc edit kubedeschedulers.operator.openshift.io cluster -n openshift-kube-descheduler-operator
   ```

2. Update the **deschedulingIntervalSeconds** field to the desired value:

   ```
   apiVersion: operator.openshift.io/v1
   kind: KubeDescheduler
   metadata:
     name: cluster
     namespace: openshift-kube-descheduler-operator
   spec:
     deschedulingIntervalSeconds: 3600     1
   ...
   ```

   **1**    Set the number of seconds between descheduler runs. A value of **0** in this field runs the descheduler once and exits.

3. Save the file to apply the changes.

### 3.11.6. Uninstalling the descheduler

You can remove the descheduler from your cluster by removing the descheduler instance and uninstalling the Kube Descheduler Operator. This procedure also cleans up the **KubeDescheduler** CRD and **openshift-kube-descheduler-operator** namespace.

**Prerequisites**

- Cluster administrator privileges.

- Access to the OpenShift Container Platform web console.

**Procedure**

1. Log in to the OpenShift Container Platform web console.

2. Delete the descheduler instance.

   a. From the **Operators → Installed Operators** page, click **Kube Descheduler Operator**.

   b. Select the **Kube Descheduler** tab.

   c. Click the Options menu ⋮ next to the **cluster** entry and select **Delete KubeDescheduler**.

   d. In the confirmation dialog, click **Delete**.

3. Uninstall the Kube Descheduler Operator.

   a. Navigate to **Operators → Installed Operators**,

b. Click the Options menu ⋮ next to the **Kube Descheduler Operator** entry and select **Uninstall Operator**.

c. In the confirmation dialog, click **Uninstall**.

4. Delete the **openshift-kube-descheduler-operator** namespace.

   a. Navigate to Administration → **Namespaces**.

   b. Enter **openshift-kube-descheduler-operator** into the filter box.

   c. Click the Options menu ⋮ next to the **openshift-kube-descheduler-operator** entry and select **Delete Namespace**.

   d. In the confirmation dialog, enter **openshift-kube-descheduler-operator** and click **Delete**.

5. Delete the **KubeDescheduler** CRD.

   a. Navigate to Administration → **Custom Resource Definitions**

   b. Enter **KubeDescheduler** into the filter box.

   c. Click the Options menu ⋮ next to the **KubeDescheduler** entry and select **Delete CustomResourceDefinition**.

   d. In the confirmation dialog, click **Delete**.

# CHAPTER 4. USING JOBS AND DAEMONSETS

## 4.1. RUNNING BACKGROUND TASKS ON NODES AUTOMATICALLY WITH DAEMON SETS

As an administrator, you can create and use daemon sets to run replicas of a pod on specific or all nodes in an OpenShift Container Platform cluster.

A daemon set ensures that all (or some) nodes run a copy of a pod. As nodes are added to the cluster, pods are added to the cluster. As nodes are removed from the cluster, those pods are removed through garbage collection. Deleting a daemon set will clean up the pods it created.

You can use daemon sets to create shared storage, run a logging pod on every node in your cluster, or deploy a monitoring agent on every node.

For security reasons, the cluster administrators and the project administrators can create daemon sets.

For more information on daemon sets, see the Kubernetes documentation.

> **IMPORTANT**
>
> Daemon set scheduling is incompatible with project's default node selector. If you fail to disable it, the daemon set gets restricted by merging with the default node selector. This results in frequent pod recreates on the nodes that got unselected by the merged node selector, which in turn puts unwanted load on the cluster.

### 4.1.1. Scheduled by default scheduler

A daemon set ensures that all eligible nodes run a copy of a pod. Normally, the node that a pod runs on is selected by the Kubernetes scheduler. However, previously daemon set pods are created and scheduled by the daemon set controller. That introduces the following issues:

- Inconsistent pod behavior: Normal pods waiting to be scheduled are created and in Pending state, but daemon set pods are not created in **Pending** state. This is confusing to the user.

- Pod preemption is handled by default scheduler. When preemption is enabled, the daemon set controller will make scheduling decisions without considering pod priority and preemption.

The **ScheduleDaemonSetPods** feature, enabled by default in OpenShift Container Platform, lets you to schedule daemon sets using the default scheduler instead of the daemon set controller, by adding the **NodeAffinity** term to the daemon set pods, instead of the **spec.nodeName** term. The default scheduler is then used to bind the pod to the target host. If node affinity of the daemon set pod already exists, it is replaced. The daemon set controller only performs these operations when creating or modifying daemon set pods, and no changes are made to the **spec.template** of the daemon set.

```
nodeAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    nodeSelectorTerms:
    - matchFields:
      - key: metadata.name
        operator: In
        values:
        - target-host-name
```

In addition, a **node.kubernetes.io/unschedulable:NoSchedule** toleration is added automatically to daemon set pods. The default scheduler ignores unschedulable Nodes when scheduling daemon set pods.

## 4.1.2. Creating daemonsets

When creating daemon sets, the **nodeSelector** field is used to indicate the nodes on which the daemon set should deploy replicas.

### Prerequisites

- Before you start using daemon sets, disable the default project-wide node selector in your namespace, by setting the namespace annotation **openshift.io/node-selector** to an empty string:

```
$ oc patch namespace myproject -p \
    '{"metadata": {"annotations": {"openshift.io/node-selector": ""}}}'
```

  **TIP**

  You can alternatively apply the following YAML to disable the default project-wide node selector for a namespace:

```
apiVersion: v1
kind: Namespace
metadata:
  name: <namespace>
  annotations:
    openshift.io/node-selector: ''
```

- If you are creating a new project, overwrite the default node selector:

```
$ oc adm new-project <name> --node-selector=""
```

### Procedure

To create a daemon set:

1. Define the daemon set yaml file:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: hello-daemonset
spec:
  selector:
    matchLabels:
      name: hello-daemonset 1
  template:
    metadata:
      labels:
        name: hello-daemonset 2
    spec:
```

```
    nodeSelector: 3
      role: worker
    containers:
    - image: openshift/hello-openshift
      imagePullPolicy: Always
      name: registry
      ports:
      - containerPort: 80
        protocol: TCP
      resources: {}
      terminationMessagePath: /dev/termination-log
    serviceAccount: default
    terminationGracePeriodSeconds: 10
```

[1] The label selector that determines which pods belong to the daemon set.

[2] The pod template's label selector. Must match the label selector above.

[3] The node selector that determines on which nodes pod replicas should be deployed. A matching label must be present on the node.

2. Create the daemon set object:

   ```
   $ oc create -f daemonset.yaml
   ```

3. To verify that the pods were created, and that each node has a pod replica:

   a. Find the daemonset pods:

      ```
      $ oc get pods
      ```

      **Example output**

      ```
      hello-daemonset-cx6md   1/1     Running   0       2m
      hello-daemonset-e3md9   1/1     Running   0       2m
      ```

   b. View the pods to verify the pod has been placed onto the node:

      ```
      $ oc describe pod/hello-daemonset-cx6md|grep Node
      ```

      **Example output**

      ```
      Node:        openshift-node01.hostname.com/10.14.20.134
      ```

      ```
      $ oc describe pod/hello-daemonset-e3md9|grep Node
      ```

      **Example output**

      ```
      Node:        openshift-node02.hostname.com/10.14.20.137
      ```

> **IMPORTANT**
>
> - If you update a daemon set pod template, the existing pod replicas are not affected.
>
> - If you delete a daemon set and then create a new daemon set with a different template but the same label selector, it recognizes any existing pod replicas as having matching labels and thus does not update them or create new replicas despite a mismatch in the pod template.
>
> - If you change node labels, the daemon set adds pods to nodes that match the new labels and deletes pods from nodes that do not match the new labels.
>
> To update a daemon set, force new pod replicas to be created by deleting the old replicas or nodes.

## 4.2. RUNNING TASKS IN PODS USING JOBS

A *job* executes a task in your OpenShift Container Platform cluster.

A job tracks the overall progress of a task and updates its status with information about active, succeeded, and failed pods. Deleting a job will clean up any pod replicas it created. Jobs are part of the Kubernetes API, which can be managed with **oc** commands like other object types.

**Sample Job specification**

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  parallelism: 1            1
  completions: 1            2
  activeDeadlineSeconds: 1800   3
  backoffLimit: 6           4
  template:                 5
   metadata:
     name: pi
   spec:
     containers:
     - name: pi
       image: perl
       command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
     restartPolicy: OnFailure      6
```

**1** The pod replicas a job should run in parallel.

**2** Successful pod completions are needed to mark a job completed.

**3** The maximum duration the job can run.

**4** The number of retries for a job.

**5** The template for the pod the controller creates.

**6** The restart policy of the pod.

See the Kubernetes documentation for more information about jobs.

## 4.2.1. Understanding jobs and cron jobs

A job tracks the overall progress of a task and updates its status with information about active, succeeded, and failed pods. Deleting a job cleans up any pods it created. Jobs are part of the Kubernetes API, which can be managed with **oc** commands like other object types.

There are two possible resource types that allow creating run-once objects in OpenShift Container Platform:

**Job**

A regular job is a run-once object that creates a task and ensures the job finishes.

There are three main types of task suitable to run as a job:

- Non-parallel jobs:

    - A job that starts only one pod, unless the pod fails.

    - The job is complete as soon as its pod terminates successfully.

- Parallel jobs with a fixed completion count:

    - a job that starts multiple pods.

    - The job represents the overall task and is complete when there is one successful pod for each value in the range **1** to the **completions** value.

- Parallel jobs with a work queue:

    - A job with multiple parallel worker processes in a given pod.

    - OpenShift Container Platform coordinates pods to determine what each should work on or use an external queue service.

    - Each pod is independently capable of determining whether or not all peer pods are complete and that the entire job is done.

    - When any pod from the job terminates with success, no new pods are created.

    - When at least one pod has terminated with success and all pods are terminated, the job is successfully completed.

    - When any pod has exited with success, no other pod should be doing any work for this task or writing any output. Pods should all be in the process of exiting.

For more information about how to make use of the different types of job, see Job Patterns in the Kubernetes documentation.

**Cron job**

A job can be scheduled to run multiple times, using a cron job.

A *cron job* builds on a regular job by allowing you to specify how the job should be run. Cron jobs are part of the Kubernetes API, which can be managed with **oc** commands like other object types.

Cron jobs are useful for creating periodic and recurring tasks, like running backups or sending emails. Cron jobs can also schedule individual tasks for a specific time, such as if you want to schedule a job for a low activity period. A cron job creates a **Job** object based on the timezone configured on the control plane node that runs the cronjob controller.

> **WARNING**
>
> A cron job creates a **Job** object approximately once per execution time of its schedule, but there are circumstances in which it fails to create a job or two jobs might be created. Therefore, jobs must be idempotent and you must configure history limits.

### 4.2.1.1. Understanding how to create jobs

Both resource types require a job configuration that consists of the following key parts:

- A pod template, which describes the pod that OpenShift Container Platform creates.

- The **parallelism** parameter, which specifies how many pods running in parallel at any point in time should execute a job.

  - For non-parallel jobs, leave unset. When unset, defaults to **1**.

- The **completions** parameter, specifying how many successful pod completions are needed to finish a job.

  - For non-parallel jobs, leave unset. When unset, defaults to **1**.

  - For parallel jobs with a fixed completion count, specify a value.

  - For parallel jobs with a work queue, leave unset. When unset defaults to the **parallelism** value.

### 4.2.1.2. Understanding how to set a maximum duration for jobs

When defining a job, you can define its maximum duration by setting the **activeDeadlineSeconds** field. It is specified in seconds and is not set by default. When not set, there is no maximum duration enforced.

The maximum duration is counted from the time when a first pod gets scheduled in the system, and defines how long a job can be active. It tracks overall time of an execution. After reaching the specified timeout, the job is terminated by OpenShift Container Platform.

### 4.2.1.3. Understanding how to set a job back off policy for pod failure

A job can be considered failed, after a set amount of retries due to a logical error in configuration or other similar reasons. Failed pods associated with the job are recreated by the controller with an exponential back off delay (**10s**, **20s**, **40s** ...) capped at six minutes. The limit is reset if no new failed pods appear between controller checks.

Use the **spec.backoffLimit** parameter to set the number of retries for a job.

### 4.2.1.4. Understanding how to configure a cron job to remove artifacts

Cron jobs can leave behind artifact resources such as jobs or pods. As a user it is important to configure history limits so that old jobs and their pods are properly cleaned. There are two fields within cron job's spec responsible for that:

- **.spec.successfulJobsHistoryLimit**. The number of successful finished jobs to retain (defaults to 3).

- **.spec.failedJobsHistoryLimit**. The number of failed finished jobs to retain (defaults to 1).

TIP

- Delete cron jobs that you no longer need:

  ```
  $ oc delete cronjob/<cron_job_name>
  ```

  Doing this prevents them from generating unnecessary artifacts.

- You can suspend further executions by setting the **spec.suspend** to true. All subsequent executions are suspended until you reset to **false**.

### 4.2.1.5. Known limitations

The job specification restart policy only applies to the *pods*, and not the *job controller*. However, the job controller is hard-coded to keep retrying jobs to completion.

As such, **restartPolicy: Never** or **--restart=Never** results in the same behavior as **restartPolicy: OnFailure** or **--restart=OnFailure**. That is, when a job fails it is restarted automatically until it succeeds (or is manually discarded). The policy only sets which subsystem performs the restart.

With the **Never** policy, the *job controller* performs the restart. With each attempt, the job controller increments the number of failures in the job status and create new pods. This means that with each failed attempt, the number of pods increases.

With the **OnFailure** policy, *kubelet* performs the restart. Each attempt does not increment the number of failures in the job status. In addition, kubelet will retry failed jobs starting pods on the same nodes.

### 4.2.2. Creating jobs

You create a job in OpenShift Container Platform by creating a job object.

#### Procedure

To create a job:

1. Create a YAML file similar to the following:

   ```
   apiVersion: batch/v1
   kind: Job
   metadata:
     name: pi
   spec:
   ```

```
parallelism: 1          1
completions: 1          2
activeDeadlineSeconds: 1800   3
backoffLimit: 6         4
template:               5
  metadata:
    name: pi
  spec:
    containers:
    - name: pi
      image: perl
      command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
    restartPolicy: OnFailure    6
```

**1**    Optional: Specify how many pod replicas a job should run in parallel; defaults to **1**.

- For non-parallel jobs, leave unset. When unset, defaults to **1**.

**2**    Optional: Specify how many successful pod completions are needed to mark a job completed.

- For non-parallel jobs, leave unset. When unset, defaults to **1**.

- For parallel jobs with a fixed completion count, specify the number of completions.

- For parallel jobs with a work queue, leave unset. When unset defaults to the **parallelism** value.

**3**    Optional: Specify the maximum duration the job can run.

**4**    Optional: Specify the number of retries for a job. This field defaults to six.

**5**    Specify the template for the pod the controller creates.

**6**    Specify the restart policy of the pod:

- **Never**. Do not restart the job.

- **OnFailure**. Restart the job only if it fails.

- **Always**. Always restart the job.
  For details on how OpenShift Container Platform uses restart policy with failed containers, see the Example States in the Kubernetes documentation.

2. Create the job:

```
$ oc create -f <file-name>.yaml
```

> **NOTE**
>
> You can also create and launch a job from a single command using **oc create job**. The following command creates and launches a job similar to the one specified in the previous example:
>
> ```
> $ oc create job pi --image=perl -- perl -Mbignum=bpi -wle 'print bpi(2000)'
> ```

### 4.2.3. Creating cron jobs

You create a cron job in OpenShift Container Platform by creating a job object.

**Procedure**

To create a cron job:

1. Create a YAML file similar to the following:

   ```
   apiVersion: batch/v1
   kind: CronJob
   metadata:
     name: pi
   spec:
     schedule: "*/1 * * * *"          1
     concurrencyPolicy: "Replace"     2
     startingDeadlineSeconds: 200     3
     suspend: true                    4
     successfulJobsHistoryLimit: 3    5
     failedJobsHistoryLimit: 1        6
     jobTemplate:                     7
       spec:
         template:
           metadata:
             labels:                  8
               parent: "cronjobpi"
           spec:
             containers:
             - name: pi
               image: perl
               command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
             restartPolicy: OnFailure 9
   ```

   **1**   Schedule for the job specified in cron format. In this example, the job will run every minute.

   **2**   An optional concurrency policy, specifying how to treat concurrent jobs within a cron job. Only one of the following concurrent policies may be specified. If not specified, this defaults to allowing concurrent executions.

   - **Allow** allows cron jobs to run concurrently.

   - **Forbid** forbids concurrent runs, skipping the next run if the previous has not finished yet.

   - **Replace** cancels the currently running job and replaces it with a new one.

**3** An optional deadline (in seconds) for starting the job if it misses its scheduled time for any reason. Missed jobs executions will be counted as failed ones. If not specified, there is no

**4** An optional flag allowing the suspension of a cron job. If set to **true**, all subsequent executions will be suspended.

**5** The number of successful finished jobs to retain (defaults to 3).

**6** The number of failed finished jobs to retain (defaults to 1).

**7** Job template. This is similar to the job example.

**8** Sets a label for jobs spawned by this cron job.

**9** The restart policy of the pod. This does not apply to the job controller.

> **NOTE**
>
> The **.spec.successfulJobsHistoryLimit** and **.spec.failedJobsHistoryLimit** fields are optional. These fields specify how many completed and failed jobs should be kept. By default, they are set to **3** and **1** respectively. Setting a limit to **0** corresponds to keeping none of the corresponding kind of jobs after they finish.

2. Create the cron job:

```
$ oc create -f <file-name>.yaml
```

> **NOTE**
>
> You can also create and launch a cron job from a single command using **oc create cronjob**. The following command creates and launches a cron job similar to the one specified in the previous example:
>
> ```
> $ oc create cronjob pi --image=perl --schedule='*/1 * * * *' -- perl -Mbignum=bpi -wle 'print bpi(2000)'
> ```
>
> With **oc create cronjob**, the **--schedule** option accepts schedules in  cron format.

# CHAPTER 5. WORKING WITH NODES

## 5.1. VIEWING AND LISTING THE NODES IN YOUR OPENSHIFT CONTAINER PLATFORM CLUSTER

You can list all the nodes in your cluster to obtain information such as status, age, memory usage, and details about the nodes.

When you perform node management operations, the CLI interacts with node objects that are representations of actual node hosts. The master uses the information from node objects to validate nodes with health checks.

### 5.1.1. About listing all the nodes in a cluster

You can get detailed information on the nodes in the cluster.

- The following command lists all nodes:

  ```
  $ oc get nodes
  ```

  The following example is a cluster with healthy nodes:

  ```
  $ oc get nodes
  ```

  **Example output**

  ```
  NAME              STATUS  ROLES   AGE     VERSION
  master.example.com   Ready    master   7h       v1.21.0
  node1.example.com    Ready    worker   7h       v1.21.0
  node2.example.com    Ready    worker   7h       v1.21.0
  ```

  The following example is a cluster with one unhealthy node:

  ```
  $ oc get nodes
  ```

  **Example output**

  ```
  NAME              STATUS                ROLES   AGE     VERSION
  master.example.com   Ready                  master   7h       v1.21.0
  node1.example.com    NotReady,SchedulingDisabled worker   7h        v1.21.0
  node2.example.com    Ready                  worker   7h       v1.21.0
  ```

  The conditions that trigger a **NotReady** status are shown later in this section.

- The **-o wide** option provides additional information on nodes.

  ```
  $ oc get nodes -o wide
  ```

  **Example output**

  ```
  NAME              STATUS  ROLES   AGE   VERSION       INTERNAL-IP   EXTERNAL-IP
  ```

```
OS-IMAGE                                    KERNEL-VERSION              CONTAINER-
RUNTIME
master.example.com  Ready   master  171m  v1.21.0+39c0afe  10.0.129.108  <none>
Red Hat Enterprise Linux CoreOS 48.83.202103210901-0 (Ootpa)   4.18.0-
240.15.1.el8_3.x86_64   cri-o://1.21.0-30.rhaos4.8.gitf2f339d.el8-dev
node1.example.com  Ready   worker  72m   v1.21.0+39c0afe  10.0.129.222  <none>
Red Hat Enterprise Linux CoreOS 48.83.202103210901-0 (Ootpa)   4.18.0-
240.15.1.el8_3.x86_64   cri-o://1.21.0-30.rhaos4.8.gitf2f339d.el8-dev
node2.example.com  Ready   worker  164m  v1.21.0+39c0afe  10.0.142.150  <none>
Red Hat Enterprise Linux CoreOS 48.83.202103210901-0 (Ootpa)   4.18.0-
240.15.1.el8_3.x86_64   cri-o://1.21.0-30.rhaos4.8.gitf2f339d.el8-dev
```

- The following command lists information about a single node:

```
$ oc get node <node>
```

For example:

```
$ oc get node node1.example.com
```

**Example output**

```
NAME               STATUS   ROLES   AGE     VERSION
node1.example.com    Ready    worker   7h       v1.21.0
```

- The following command provides more detailed information about a specific node, including the
  reason for the current condition:

```
$ oc describe node <node>
```

For example:

```
$ oc describe node node1.example.com
```

**Example output**

```
Name:             node1.example.com  1
Roles:            worker  2
Labels:           beta.kubernetes.io/arch=amd64  3
                  beta.kubernetes.io/instance-type=m4.large
                  beta.kubernetes.io/os=linux
                  failure-domain.beta.kubernetes.io/region=us-east-2
                  failure-domain.beta.kubernetes.io/zone=us-east-2a
                  kubernetes.io/hostname=ip-10-0-140-16
                  node-role.kubernetes.io/worker=
Annotations:      cluster.k8s.io/machine: openshift-machine-api/ahardin-worker-us-east-2a-
q5dzc  4
                  machineconfiguration.openshift.io/currentConfig: worker-
309c228e8b3a92e2235edd544c62fea8
                  machineconfiguration.openshift.io/desiredConfig: worker-
309c228e8b3a92e2235edd544c62fea8
                  machineconfiguration.openshift.io/state: Done
                  volumes.kubernetes.io/controller-managed-attach-detach: true
```

CreationTimestamp:  Wed, 13 Feb 2019 11:05:57 -0500
Taints:                <none> **5**
Unschedulable:    false
Conditions:             **6**
  Type            Status  LastHeartbeatTime                LastTransitionTime            Reason
Message
  ----            ------  -----------------                ------------------            ------            -------
  OutOfDisk       False   Wed, 13 Feb 2019 15:09:42 -0500   Wed, 13 Feb 2019 11:05:57 -
0500   KubeletHasSufficientDisk     kubelet has sufficient disk space available
  MemoryPressure  False   Wed, 13 Feb 2019 15:09:42 -0500   Wed, 13 Feb 2019 11:05:57
-0500   KubeletHasSufficientMemory   kubelet has sufficient memory available
  DiskPressure    False   Wed, 13 Feb 2019 15:09:42 -0500   Wed, 13 Feb 2019 11:05:57 -
0500   KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure     False   Wed, 13 Feb 2019 15:09:42 -0500   Wed, 13 Feb 2019 11:05:57 -
0500   KubeletHasSufficientPID      kubelet has sufficient PID available
  Ready           True    Wed, 13 Feb 2019 15:09:42 -0500   Wed, 13 Feb 2019 11:07:09 -0500
KubeletReady                kubelet is posting ready status
Addresses:        **7**
  InternalIP:   10.0.140.16
  InternalDNS:  ip-10-0-140-16.us-east-2.compute.internal
  Hostname:     ip-10-0-140-16.us-east-2.compute.internal
Capacity:         **8**
 attachable-volumes-aws-ebs:  39
 cpu:                 2
 hugepages-1Gi:          0
 hugepages-2Mi:          0
 memory:              8172516Ki
 pods:               250
Allocatable:
 attachable-volumes-aws-ebs:  39
 cpu:                1500m
 hugepages-1Gi:          0
 hugepages-2Mi:          0
 memory:              7558116Ki
 pods:               250
System Info:      **9**
 Machine ID:                  63787c9534c24fde9a0cde35c13f1f66
 System UUID:                 EC22BF97-A006-4A58-6AF8-0A38DEEA122A
 Boot ID:                     f24ad37d-2594-46b4-8830-7f7555918325
 Kernel Version:              3.10.0-957.5.1.el7.x86_64
 OS Image:                    Red Hat Enterprise Linux CoreOS 410.8.20190520.0 (Ootpa)
 Operating System:            linux
 Architecture:                amd64
 Container Runtime Version:   cri-o://1.16.0-0.6.dev.rhaos4.3.git9ad059b.el8-rc2
 Kubelet Version:             v1.21.0
 Kube-Proxy Version:          v1.21.0
PodCIDR:                      10.128.4.0/24
ProviderID:                   aws:///us-east-2a/i-04e87b31dc6b3e171
Non-terminated Pods:          (13 in total) **10**
  Namespace                Name                         CPU Requests  CPU Limits
Memory Requests  Memory Limits
  ---------                ----                         -----------  ----------  --------------  -------
------
  openshift-cluster-node-tuning-operator  tuned-hdl5q              0 (0%)     0 (0%)     0
(0%)        0 (0%)

```
  openshift-dns                    dns-default-l69zr                0 (0%)      0 (0%)      0 (0%)
0 (0%)
  openshift-image-registry         node-ca-9hmcg                    0 (0%)      0 (0%)      0
(0%)        0 (0%)
  openshift-ingress                router-default-76455c45c-c5ptv      0 (0%)      0 (0%)      0
(0%)        0 (0%)
  openshift-machine-config-operator    machine-config-daemon-cvqw9        20m (1%)      0
(0%)     50Mi (0%)      0 (0%)
  openshift-marketplace            community-operators-f67fh        0 (0%)      0 (0%)
0 (0%)        0 (0%)
  openshift-monitoring             alertmanager-main-0              50m (3%)     50m (3%)
210Mi (2%)      10Mi (0%)
  openshift-monitoring             grafana-78765ddcc7-hnjmm             100m (6%)     200m
(13%) 100Mi (1%)      200Mi (2%)
  openshift-monitoring             node-exporter-l7q8d              10m (0%)     20m (1%)
20Mi (0%)      40Mi (0%)
  openshift-monitoring             prometheus-adapter-75d769c874-hvb85    0 (0%)      0
(0%)     0 (0%)        0 (0%)
  openshift-multus                 multus-kw8w5                     0 (0%)      0 (0%)      0 (0%)
0 (0%)
  openshift-sdn                    ovs-t4dsn                        100m (6%)     0 (0%)     300Mi
(4%)      0 (0%)
  openshift-sdn                    sdn-g79hg                        100m (6%)     0 (0%)     200Mi
(2%)      0 (0%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource                Requests     Limits
  --------                --------     ------
  cpu                     380m (25%)   270m (18%)
  memory                  880Mi (11%)  250Mi (3%)
  attachable-volumes-aws-ebs  0           0
Events:       11
  Type     Reason                Age          From                  Message
  ----     ------                ----         ----                  -------
  Normal   NodeHasSufficientPID    6d (x5 over 6d)   kubelet, m01.example.com  Node
m01.example.com status is now: NodeHasSufficientPID
  Normal   NodeAllocatableEnforced 6d           kubelet, m01.example.com  Updated Node
Allocatable limit across pods
  Normal   NodeHasSufficientMemory 6d (x6 over 6d)   kubelet, m01.example.com  Node
m01.example.com status is now: NodeHasSufficientMemory
  Normal   NodeHasNoDiskPressure   6d (x6 over 6d)   kubelet, m01.example.com  Node
m01.example.com status is now: NodeHasNoDiskPressure
  Normal   NodeHasSufficientDisk   6d (x6 over 6d)   kubelet, m01.example.com  Node
m01.example.com status is now: NodeHasSufficientDisk
  Normal   NodeHasSufficientPID    6d           kubelet, m01.example.com  Node
m01.example.com status is now: NodeHasSufficientPID
  Normal   Starting                6d           kubelet, m01.example.com  Starting kubelet.
  ...
```

**1**  The name of the node.

**2**  The role of the node, either **master** or **worker**.

**3**  The labels applied to the node.

**4**  The annotations applied to the node.

**5** The taints applied to the node.

**6** The node conditions and status. The **conditions** stanza lists the **Ready**, **PIDPressure**, **PIDPressure**, **MemoryPressure**, **DiskPressure** and **OutOfDisk** status. These condition are described later in this section.

**7** The IP address and hostname of the node.

**8** The pod resources and allocatable resources.

**9** Information about the node host.

**10** The pods on the node.

**11** The events reported by the node.

Among the information shown for nodes, the following node conditions appear in the output of the commands shown in this section:

Table 5.1. Node Conditions

| Condition | Description |
|-----------|-------------|
| **Ready** | If **true**, the node is healthy and ready to accept pods. If **false**, the node is not healthy and is not accepting pods. If **unknown**, the node controller has not received a heartbeat from the node for the **node-monitor-grace-period** (the default is 40 seconds). |
| **DiskPressure** | If **true**, the disk capacity is low. |
| **MemoryPressure** | If **true**, the node memory is low. |
| **PIDPressure** | If **true**, there are too many processes on the node. |
| **OutOfDisk** | If **true**, the node has insufficient free space on the node for adding new pods. |
| **NetworkUnavailable** | If **true**, the network for the node is not correctly configured. |
| **NotReady** | If **true**, one of the underlying components, such as the container runtime or network, is experiencing issues or is not yet configured. |
| **SchedulingDisabled** | Pods cannot be scheduled for placement on the node. |

## 5.1.2. Listing pods on a node in your cluster

You can list all the pods on a specific node.

**Procedure**

- To list all or selected pods on one or more nodes:

```
$ oc describe node <node1> <node2>
```

For example:

```
$ oc describe node ip-10-0-128-218.ec2.internal
```

- To list all or selected pods on selected nodes:

```
$ oc describe --selector=<node_selector>
```

```
$ oc describe node  --selector=kubernetes.io/os
```

Or:

```
$ oc describe -l=<pod_selector>
```

```
$ oc describe node -l node-role.kubernetes.io/worker
```

- To list all pods on a specific node, including terminated pods:

```
$ oc get pod --all-namespaces --field-selector=spec.nodeName=<nodename>
```

### 5.1.3. Viewing memory and CPU usage statistics on your nodes

You can display usage statistics about nodes, which provide the runtime environments for containers. These usage statistics include CPU, memory, and storage consumption.

**Prerequisites**

- You must have **cluster-reader** permission to view the usage statistics.

- Metrics must be installed to view the usage statistics.

**Procedure**

- To view the usage statistics:

```
$ oc adm top nodes
```

**Example output**

```
NAME                          CPU(cores)  CPU%     MEMORY(bytes)  MEMORY%
ip-10-0-12-143.ec2.compute.internal   1503m      100%     4533Mi        61%
ip-10-0-132-16.ec2.compute.internal   76m        5%       1391Mi        18%
ip-10-0-140-137.ec2.compute.internal  398m       26%      2473Mi        33%
ip-10-0-142-44.ec2.compute.internal   656m       43%      6119Mi        82%
ip-10-0-146-165.ec2.compute.internal  188m       12%      3367Mi        45%
ip-10-0-19-62.ec2.compute.internal    896m       59%      5754Mi        77%
ip-10-0-44-193.ec2.compute.internal   632m       42%      5349Mi        72%
```

- To view the usage statistics for nodes with labels:

```
$ oc adm top node --selector="
```

You must choose the selector (label query) to filter on. Supports **=**, **==**, and **!=**.

## 5.2. WORKING WITH NODES

As an administrator, you can perform a number of tasks to make your clusters more efficient.

### 5.2.1. Understanding how to evacuate pods on nodes

Evacuating pods allows you to migrate all or selected pods from a given node or nodes.

You can only evacuate pods backed by a replication controller. The replication controller creates new pods on other nodes and removes the existing pods from the specified node(s).

Bare pods, meaning those not backed by a replication controller, are unaffected by default. You can evacuate a subset of pods by specifying a pod-selector. Pod selectors are based on labels, so all the pods with the specified label will be evacuated.

**Procedure**

1. Mark the nodes unschedulable before performing the pod evacuation.

    a. Mark the node as unschedulable:

    ```
    $ oc adm cordon <node1>
    ```

    **Example output**

    ```
    node/<node1> cordoned
    ```

    b. Check that the node status is **Ready,SchedulingDisabled**:

    ```
    $ oc get node <node1>
    ```

    **Example output**

    ```
    NAME       STATUS              ROLES    AGE    VERSION
    <node1>    Ready,SchedulingDisabled   worker   1d     v1.24.0
    ```

2. Evacuate the pods using one of the following methods:

    - Evacuate all or selected pods on one or more nodes:

    ```
    $ oc adm drain <node1> <node2> [--pod-selector=<pod_selector>]
    ```

    - Force the deletion of bare pods using the **--force** option. When set to **true**, deletion continues even if there are pods not managed by a replication controller, replica set, job, daemon set, or stateful set:

    ```
    $ oc adm drain <node1> <node2> --force=true
    ```

- Set a period of time in seconds for each pod to terminate gracefully, use **--grace-period**. If negative, the default value specified in the pod will be used:

  ```
  $ oc adm drain <node1> <node2> --grace-period=-1
  ```

- Ignore pods managed by daemon sets using the **--ignore-daemonsets** flag set to **true**:

  ```
  $ oc adm drain <node1> <node2> --ignore-daemonsets=true
  ```

- Set the length of time to wait before giving up using the **--timeout** flag. A value of **0** sets an infinite length of time:

  ```
  $ oc adm drain <node1> <node2> --timeout=5s
  ```

- Delete pods even if there are pods using **emptyDir** volumes by setting the **--delete-emptydir-data** flag to **true**. Local data is deleted when the node is drained:

  ```
  $ oc adm drain <node1> <node2> --delete-emptydir-data=true
  ```

- List objects that will be migrated without actually performing the evacuation, using the **--dry-run** option set to **true**:

  ```
  $ oc adm drain <node1> <node2>  --dry-run=true
  ```

  Instead of specifying specific node names (for example, **<node1> <node2>**), you can use the **--selector=<node_selector>** option to evacuate pods on selected nodes.

3. Mark the node as schedulable when done.

   ```
   $ oc adm uncordon <node1>
   ```

## 5.2.2. Understanding how to update labels on nodes

You can update any label on a node.

Node labels are not persisted after a node is deleted even if the node is backed up by a Machine.

> **NOTE**
>
> Any change to a **MachineSet** object is not applied to existing machines owned by the machine set. For example, labels edited or added to an existing **MachineSet** object are not propagated to existing machines and nodes associated with the machine set.

- The following command adds or updates labels on a node:

  ```
  $ oc label node <node> <key_1>=<value_1> ... <key_n>=<value_n>
  ```

  For example:

  ```
  $ oc label nodes webconsole-7f7f6 unhealthy=true
  ```

**TIP**

You can alternatively apply the following YAML to apply the label:

```
kind: Node
apiVersion: v1
metadata:
  name: webconsole-7f7f6
  labels:
    unhealthy: 'true'
```

- The following command updates all pods in the namespace:

```
$ oc label pods --all <key_1>=<value_1>
```

For example:

```
$ oc label pods --all status=unhealthy
```

### 5.2.3. Understanding how to mark nodes as unschedulable or schedulable

By default, healthy nodes with a **Ready** status are marked as schedulable, which means that you can place new pods on the node. Manually marking a node as unschedulable blocks any new pods from being scheduled on the node. Existing pods on the node are not affected.

- The following command marks a node or nodes as unschedulable:

**Example output**

```
$ oc adm cordon <node>
```

For example:

```
$ oc adm cordon node1.example.com
```

**Example output**

```
node/node1.example.com cordoned

NAME              LABELS                            STATUS
node1.example.com    kubernetes.io/hostname=node1.example.com
Ready,SchedulingDisabled
```

- The following command marks a currently unschedulable node or nodes as schedulable:

```
$ oc adm uncordon <node1>
```

Alternatively, instead of specifying specific node names (for example, **<node>**), you can use the **--selector=<node_selector>** option to mark selected nodes as schedulable or unschedulable.

### 5.2.4. Deleting nodes

### 5.2.4.1. Deleting nodes from a cluster

When you delete a node using the CLI, the node object is deleted in Kubernetes, but the pods that exist on the node are not deleted. Any bare pods not backed by a replication controller become inaccessible to OpenShift Container Platform. Pods backed by replication controllers are rescheduled to other available nodes. You must delete local manifest pods.

#### Procedure

To delete a node from the OpenShift Container Platform cluster, edit the appropriate **MachineSet** object:

> **NOTE**
>
> If you are running cluster on bare metal, you cannot delete a node by editing **MachineSet** objects. Machine sets are only available when a cluster is integrated with a cloud provider. Instead you must unschedule and drain the node before manually deleting it.

1. View the machine sets that are in the cluster:

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

   The machine sets are listed in the form of <clusterid>-worker-<aws-region-az>.

2. Scale the machine set:

   ```
   $ oc scale --replicas=2 machineset <machineset> -n openshift-machine-api
   ```

   Or:

   ```
   $ oc edit machineset <machineset> -n openshift-machine-api
   ```

   > **TIP**
   >
   > You can alternatively apply the following YAML to scale the machine set:
   >
   > ```yaml
   > apiVersion: machine.openshift.io/v1beta1
   > kind: MachineSet
   > metadata:
   >   name: <machineset>
   >   namespace: openshift-machine-api
   > spec:
   >   replicas: 2
   > ```

   For more information on scaling your cluster using a machine set, see *Manually scaling a machine set*.

#### Additional resources

- For more information on scaling your cluster using a MachineSet, see Manually scaling a MachineSet.

### 5.2.4.2. Deleting nodes from a bare metal cluster

When you delete a node using the CLI, the node object is deleted in Kubernetes, but the pods that exist on the node are not deleted. Any bare pods not backed by a replication controller become inaccessible to OpenShift Container Platform. Pods backed by replication controllers are rescheduled to other available nodes. You must delete local manifest pods.

### Procedure

Delete a node from an OpenShift Container Platform cluster running on bare metal by completing the following steps:

1. Mark the node as unschedulable:

   ```
   $ oc adm cordon <node_name>
   ```

2. Drain all pods on the node:

   ```
   $ oc adm drain <node_name> --force=true
   ```

   This step might fail if the node is offline or unresponsive. Even if the node does not respond, it might still be running a workload that writes to shared storage. To avoid data corruption, power down the physical hardware before you proceed.

3. Delete the node from the cluster:

   ```
   $ oc delete node <node_name>
   ```

   Although the node object is now deleted from the cluster, it can still rejoin the cluster after reboot or if the kubelet service is restarted. To permanently delete the node and all its data, you must decommission the node.

4. If you powered down the physical hardware, turn it back on so that the node can rejoin the cluster.

## 5.3. MANAGING NODES

OpenShift Container Platform uses a KubeletConfig custom resource (CR) to manage the configuration of nodes. By creating an instance of a **KubeletConfig** object, a managed machine config is created to override setting on the node.

> **NOTE**
>
> **Logging in to remote machines for the purpose of changing their configuration is not supported.**

### 5.3.1. Modifying nodes

To make configuration changes to a cluster, or machine pool, you must create a custom resource definition (CRD), or **kubeletConfig** object. OpenShift Container Platform uses the Machine Config Controller to watch for changes introduced through the CRD to apply the changes to the cluster.

NOTE

Because the fields in a **kubeletConfig** object are passed directly to the kubelet from upstream Kubernetes, the validation of those fields is handled directly by the kubelet itself. Please refer to the relevant Kubernetes documentation for the valid values for these fields. Invalid values in the **kubeletConfig** object can render cluster nodes unusable.

**Procedure**

1. Obtain the label associated with the static CRD, Machine Config Pool, for the type of node you want to configure. Perform one of the following steps:

   a. Check current labels of the desired machine config pool.
      For example:

      ```
      $ oc get machineconfigpool  --show-labels
      ```

   **Example output**

   ```
   NAME     CONFIG                                     UPDATED   UPDATING   DEGRADED
   LABELS
   master   rendered-master-e05b81f5ca4db1d249a1bf32f9ec24fd   True      False
   False    operator.machineconfiguration.openshift.io/required-for-upgrade=
   worker   rendered-worker-f50e78e1bc06d8e82327763145bfcf62   True      False
   False
   ```

   b. Add a custom label to the desired machine config pool.
      For example:

      ```
      $ oc label machineconfigpool worker custom-kubelet=enabled
      ```

2. Create a **kubeletconfig** custom resource (CR) for your configuration change.
   For example:

   **Sample configuration for a custom-config CR**

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: KubeletConfig
   metadata:
     name: custom-config 1
   spec:
     machineConfigPoolSelector:
       matchLabels:
         custom-kubelet: enabled 2
     kubeletConfig: 3
       podsPerCore: 10
       maxPods: 250
       systemReserved:
         cpu: 2000m
         memory: 1Gi
   ```

   **1**    Assign a name to CR.

**2** Specify the label to apply the configuration change, this is the label you added to the machine config pool.

**3** Specify the new value(s) you want to change.

3. Create the CR object.

```
$ oc create -f <file-name>
```

For example:

```
$ oc create -f master-kube-config.yaml
```

Most Kubelet Configuration options can be set by the user. The following options are not allowed to be overwritten:

- CgroupDriver

- ClusterDNS

- ClusterDomain

- RuntimeRequestTimeout

- StaticPodPath
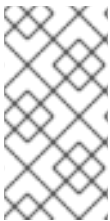
> **NOTE**
>
> If a single node contains more than 50 images, pod scheduling might be imbalanced across nodes. This is because the list of images on a node is shortened to 50 by default. You can disable the image limit by editing the **KubeletConfig** object and setting the value of **nodeStatusMaxImages** to **-1**.

## 5.3.2. Configuring control plane nodes as schedulable

You can configure control plane nodes (also known as the master nodes) to be schedulable, meaning that new pods are allowed for placement on the master nodes. By default, control plane nodes are not schedulable.

You can set the masters to be schedulable, but must retain the worker nodes.

> **NOTE**
>
> You can deploy OpenShift Container Platform with no worker nodes on a bare metal cluster. In this case, the control plane nodes are marked schedulable by default.

You can allow or disallow control plane nodes to be schedulable by configuring the **mastersSchedulable** field.

> **IMPORTANT**
>
> When you configure control plane nodes from the default unschedulable to schedulable, additional subscriptions are required. This is because control plane nodes then become worker nodes.

**Procedure**

1. Edit the **schedulers.config.openshift.io** resource.

   ```
   $ oc edit schedulers.config.openshift.io cluster
   ```

2. Configure the **mastersSchedulable** field.

   ```
   apiVersion: config.openshift.io/v1
   kind: Scheduler
   metadata:
     creationTimestamp: "2019-09-10T03:04:05Z"
     generation: 1
     name: cluster
     resourceVersion: "433"
     selfLink: /apis/config.openshift.io/v1/schedulers/cluster
     uid: a636d30a-d377-11e9-88d4-0a60097bee62
   spec:
     mastersSchedulable: false    1
     policy:
       name: ""
   status: {}
   ```

   **1**    Set to **true** to allow control plane nodes to be schedulable, or **false** to disallow control plane nodes to be schedulable.

3. Save the file to apply the changes.

## 5.3.3. Setting SELinux booleans

OpenShift Container Platform allows you to enable and disable an SELinux boolean on a Red Hat Enterprise Linux CoreOS (RHCOS) node. The following procedure explains how to modify SELinux booleans on nodes using the Machine Config Operator (MCO). This procedure uses **container_manage_cgroup** as the example boolean. You can modify this value to whichever boolean you need.

**Prerequisites**

- You have installed the OpenShift CLI (oc).

**Procedure**

1. Create a new YAML file with a **MachineConfig** object, displayed in the following example:

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: MachineConfig
   metadata:
   ```

```
      labels:
        machineconfiguration.openshift.io/role: worker
      name: 99-worker-setsebool
  spec:
    config:
      ignition:
        version: 2.2.0
      systemd:
        units:
        - contents: |
            [Unit]
            Description=Set SELinux booleans
            Before=kubelet.service

            [Service]
            Type=oneshot
            ExecStart=/sbin/setsebool container_manage_cgroup=on
            RemainAfterExit=true

            [Install]
            WantedBy=multi-user.target graphical.target
          enabled: true
          name: setsebool.service
```

2. Create the new **MachineConfig** object by running the following command:

```
$ oc create -f 99-worker-setsebool.yaml
```

> **NOTE**
>
> Applying any changes to the **MachineConfig** object causes all affected nodes to gracefully reboot after the change is applied.

## 5.3.4. Adding kernel arguments to nodes

In some special cases, you might want to add kernel arguments to a set of nodes in your cluster. This should only be done with caution and clear understanding of the implications of the arguments you set.

> **WARNING**
>
> Improper use of kernel arguments can result in your systems becoming unbootable.

Examples of kernel arguments you could set include:

- **enforcing=0**: Configures Security Enhanced Linux (SELinux) to run in permissive mode. In permissive mode, the system acts as if SELinux is enforcing the loaded security policy, including labeling objects and emitting access denial entries in the logs, but it does not actually deny any operations. While not supported for production systems, permissive mode can be helpful for debugging.

- **nosmt**: Disables symmetric multithreading (SMT) in the kernel. Multithreading allows multiple logical threads for each CPU. You could consider **nosmt** in multi-tenant environments to reduce risks from potential cross-thread attacks. By disabling SMT, you essentially choose security over performance.

See Kernel.org kernel parameters for a list and descriptions of kernel arguments.

In the following procedure, you create a **MachineConfig** object that identifies:

- A set of machines to which you want to add the kernel argument. In this case, machines with a worker role.

- Kernel arguments that are appended to the end of the existing kernel arguments.

- A label that indicates where in the list of machine configs the change is applied.

### Prerequisites

- Have administrative privilege to a working OpenShift Container Platform cluster.

### Procedure

1. List existing **MachineConfig** objects for your OpenShift Container Platform cluster to determine how to label your machine config:

   ```
   $ oc get MachineConfig
   ```

   **Example output**

   ```
   NAME                                               GENERATEDBYCONTROLLER
   IGNITIONVERSION   AGE
   00-master                                          52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0
   33m
   00-worker                                          52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0
   33m
   01-master-container-runtime                        52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
   3.2.0          33m
   01-master-kubelet                                  52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
   3.2.0          33m
   01-worker-container-runtime                        52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
   3.2.0          33m
   01-worker-kubelet                                  52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
   3.2.0          33m
   99-master-generated-registries                     52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
   3.2.0          33m
   99-master-ssh                                                                   3.2.0          40m
   99-worker-generated-registries                     52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
   3.2.0          33m
   99-worker-ssh                                                                   3.2.0          40m
   rendered-master-23e785de7587df95a4b517e0647e5ab7
   52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0          33m
   rendered-worker-5d596d9293ca3ea80c896a1191735bb1
   52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0          33m
   ```

2. Create a **MachineConfig** object file that identifies the kernel argument (for example, **05-worker-kernelarg-selinuxpermissive.yaml**)

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker ❶
  name: 05-worker-kernelarg-selinuxpermissive ❷
spec:
  config:
    ignition:
      version: 3.2.0
  kernelArguments:
    - enforcing=0 ❸
```

❶ Applies the new kernel argument only to worker nodes.

❷ Named to identify where it fits among the machine configs (05) and what it does (adds a kernel argument to configure SELinux permissive mode).

❸ Identifies the exact kernel argument as **enforcing=0**.

3. Create the new machine config:

```
$ oc create -f 05-worker-kernelarg-selinuxpermissive.yaml
```

4. Check the machine configs to see that the new one was added:

```
$ oc get MachineConfig
```

**Example output**

```
NAME                                GENERATEDBYCONTROLLER                        IGNITIONVERSION   AGE
00-master                           52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0    33m
00-worker                           52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0    33m
01-master-container-runtime          52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0    33m
01-master-kubelet                    52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0    33m
01-worker-container-runtime          52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0    33m
01-worker-kubelet                    52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0    33m
05-worker-kernelarg-selinuxpermissive                                          3.2.0    105s
99-master-generated-registries       52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0    33m
99-master-ssh                                                                   3.2.0    40m
99-worker-generated-registries       52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0    33m
99-worker-ssh                                                                   3.2.0    40m
```

```
rendered-master-23e785de7587df95a4b517e0647e5ab7
52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0          33m
rendered-worker-5d596d9293ca3ea80c896a1191735bb1
52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0          33m
```

5. Check the nodes:

```
$ oc get nodes
```

**Example output**

```
NAME                       STATUS              ROLES   AGE  VERSION
ip-10-0-136-161.ec2.internal  Ready                 worker  28m  v1.21.0
ip-10-0-136-243.ec2.internal  Ready                 master  34m  v1.21.0
ip-10-0-141-105.ec2.internal  Ready,SchedulingDisabled  worker  28m  v1.21.0
ip-10-0-142-249.ec2.internal  Ready                 master  34m  v1.21.0
ip-10-0-153-11.ec2.internal   Ready                 worker  28m  v1.21.0
ip-10-0-153-150.ec2.internal  Ready                 master  34m  v1.21.0
```

You can see that scheduling on each worker node is disabled as the change is being applied.

6. Check that the kernel argument worked by going to one of the worker nodes and listing the kernel command line arguments (in **/proc/cmdline** on the host):

```
$ oc debug node/ip-10-0-141-105.ec2.internal
```

**Example output**

```
Starting pod/ip-10-0-141-105ec2internal-debug ...
To use host binaries, run `chroot /host`

sh-4.2# cat /host/proc/cmdline
BOOT_IMAGE=/ostree/rhcos-... console=tty0 console=ttyS0,115200n8
rootflags=defaults,prjquota rw root=UUID=fd0... ostree=/ostree/boot.0/rhcos/16...
coreos.oem.id=qemu coreos.oem.id=ec2 ignition.platform.id=ec2 enforcing=0

sh-4.2# exit
```

You should see the **enforcing=0** argument added to the other kernel arguments.

## 5.4. MANAGING THE MAXIMUM NUMBER OF PODS PER NODE

In OpenShift Container Platform, you can configure the number of pods that can run on a node based on the number of processor cores on the node, a hard limit or both. If you use both options, the lower of the two limits the number of pods on a node.

Exceeding these values can result in:

- Increased CPU utilization by OpenShift Container Platform.

- Slow pod scheduling.

- Potential out-of-memory scenarios, depending on the amount of memory in the node.

- Exhausting the IP address pool.

- Resource overcommitting, leading to poor user application performance.

> **NOTE**
>
> A pod that is holding a single container actually uses two containers. The second container sets up networking prior to the actual container starting. As a result, a node running 10 pods actually has 20 containers running.

The **podsPerCore** parameter limits the number of pods the node can run based on the number of processor cores on the node. For example, if **podsPerCore** is set to **10** on a node with 4 processor cores, the maximum number of pods allowed on the node is 40.

The **maxPods** parameter limits the number of pods the node can run to a fixed value, regardless of the properties of the node.

## 5.4.1. Configuring the maximum number of pods per node

Two parameters control the maximum number of pods that can be scheduled to a node: **podsPerCore** and **maxPods**. If you use both options, the lower of the two limits the number of pods on a node.

For example, if **podsPerCore** is set to **10** on a node with 4 processor cores, the maximum number of pods allowed on the node will be 40.

**Prerequisites**

1. Obtain the label associated with the static **MachineConfigPool** CRD for the type of node you want to configure by entering the following command:

   ```
   $ oc edit machineconfigpool <name>
   ```

   For example:

   ```
   $ oc edit machineconfigpool worker
   ```

   **Example output**

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: MachineConfigPool
   metadata:
     creationTimestamp: "2022-11-16T15:34:25Z"
     generation: 4
     labels:
       pools.operator.machineconfiguration.openshift.io/worker: ""  ❶
     name: worker
   ```

   ❶ The label appears under Labels.

### TIP

If the label is not present, add a key/value pair such as:

```
$ oc label machineconfigpool worker custom-kubelet=small-pods
```

### Procedure

1. Create a custom resource (CR) for your configuration change.

   **Sample configuration for a max-pods CR**

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: KubeletConfig
   metadata:
     name: set-max-pods 1
   spec:
     machineConfigPoolSelector:
       matchLabels:
         pools.operator.machineconfiguration.openshift.io/worker: "" 2
     kubeletConfig:
       podsPerCore: 10 3
       maxPods: 250 4
   ```

   **1**    Assign a name to CR.

   **2**    Specify the label from the machine config pool.

   **3**    Specify the number of pods the node can run based on the number of processor cores on the node.

   **4**    Specify the number of pods the node can run to a fixed value, regardless of the properties of the node.

   ### NOTE

   Setting **podsPerCore** to **0** disables this limit.

   In the above example, the default value for **podsPerCore** is **10** and the default value for **maxPods** is **250**. This means that unless the node has 25 cores or more, by default, **podsPerCore** will be the limiting factor.

2. Run the following command to create the CR:

   ```
   $ oc create -f <file_name>.yaml
   ```

### Verification

1. List the **MachineConfigPool** CRDs to see if the change is applied. The **UPDATING** column reports **True** if the change is picked up by the Machine Config Controller:

   ```
   $ oc get machineconfigpools
   ```

### Example output

```
NAME     CONFIG                   UPDATED   UPDATING   DEGRADED
master   master-9cc2c72f205e103bb534   False     False      False
worker   worker-8cecd1236b33ee3f8a5e   False     True       False
```

Once the change is complete, the **UPDATED** column reports **True**.

```
$ oc get machineconfigpools
```

### Example output

```
NAME     CONFIG                   UPDATED   UPDATING   DEGRADED
master   master-9cc2c72f205e103bb534   False     True       False
worker   worker-8cecd1236b33ee3f8a5e   True      False      False
```

## 5.5. USING THE NODE TUNING OPERATOR

Learn about the Node Tuning Operator and how you can use it to manage node-level tuning by orchestrating the tuned daemon.

The Node Tuning Operator helps you manage node-level tuning by orchestrating the TuneD daemon. The majority of high-performance applications require some level of kernel tuning. The Node Tuning Operator provides a unified management interface to users of node-level sysctls and more flexibility to add custom tuning specified by user needs.

The Operator manages the containerized TuneD daemon for OpenShift Container Platform as a Kubernetes daemon set. It ensures the custom tuning specification is passed to all containerized TuneD daemons running in the cluster in the format that the daemons understand. The daemons run on all nodes in the cluster, one per node.

Node-level settings applied by the containerized TuneD daemon are rolled back on an event that triggers a profile change or when the containerized TuneD daemon is terminated gracefully by receiving and handling a termination signal.

The Node Tuning Operator is part of a standard OpenShift Container Platform installation in version 4.1 and later.

### 5.5.1. Accessing an example Node Tuning Operator specification

Use this process to access an example Node Tuning Operator specification.

**Procedure**

1. Run:

   ```
   $ oc get Tuned/default -o yaml -n openshift-cluster-node-tuning-operator
   ```

The default CR is meant for delivering standard node-level tuning for the OpenShift Container Platform platform and it can only be modified to set the Operator Management state. Any other custom changes to the default CR will be overwritten by the Operator. For custom tuning, create your own Tuned CRs. Newly created CRs will be combined with the default CR and custom tuning applied to OpenShift Container Platform nodes based on node or pod labels and profile priorities.

> **WARNING**
>
> While in certain situations the support for pod labels can be a convenient way of automatically delivering required tuning, this practice is discouraged and strongly advised against, especially in large-scale clusters. The default Tuned CR ships without pod label matching. If a custom profile is created with pod label matching, then the functionality will be enabled at that time. The pod label functionality might be deprecated in future versions of the Node Tuning Operator.

## 5.5.2. Custom tuning specification

The custom resource (CR) for the Operator has two major sections. The first section, **profile:**, is a list of TuneD profiles and their names. The second, **recommend:**, defines the profile selection logic.

Multiple custom tuning specifications can co-exist as multiple CRs in the Operator's namespace. The existence of new CRs or the deletion of old CRs is detected by the Operator. All existing custom tuning specifications are merged and appropriate objects for the containerized TuneD daemons are updated.

**Management state**

The Operator Management state is set by adjusting the default Tuned CR. By default, the Operator is in the Managed state and the **spec.managementState** field is not present in the default Tuned CR. Valid values for the Operator Management state are as follows:

- Managed: the Operator will update its operands as configuration resources are updated

- Unmanaged: the Operator will ignore changes to the configuration resources

- Removed: the Operator will remove its operands and resources the Operator provisioned

**Profile data**

The **profile:** section lists TuneD profiles and their names.

```
profile:
- name: tuned_profile_1
  data: |
    # TuneD profile specification
    [main]
    summary=Description of tuned_profile_1 profile

    [sysctl]
    net.ipv4.ip_forward=1
    # ... other sysctl's or other TuneD daemon plugins supported by the containerized TuneD

# ...

- name: tuned_profile_n
  data: |
    # TuneD profile specification
    [main]
```

```
summary=Description of tuned_profile_n profile

# tuned_profile_n profile settings
```

**Recommended profiles**

The **profile:** selection logic is defined by the **recommend:** section of the CR. The **recommend:** section is a list of items to recommend the profiles based on a selection criteria.

```
recommend:
<recommend-item-1>
# ...
<recommend-item-n>
```

The individual items of the list:

```
- machineConfigLabels:  1
    <mcLabels>  2
  match:  3
    <match>  4
  priority: <priority>  5
  profile: <tuned_profile_name>  6
  operand:  7
    debug: <bool>  8
```

**1**    Optional.

**2**    A dictionary of key/value **MachineConfig** labels. The keys must be unique.

**3**    If omitted, profile match is assumed unless a profile with a higher priority matches first or **machineConfigLabels** is set.

**4**    An optional list.

**5**    Profile ordering priority. Lower numbers mean higher priority (**0** is the highest priority).

**6**    A TuneD profile to apply on a match. For example **tuned_profile_1**.

**7**    Optional operand configuration.

**8**    Turn debugging on or off for the TuneD daemon. Options are **true** for on or **false** for off. The default is **false**.

**<match>** is an optional list recursively defined as follows:

```
- label: <label_name>  1
  value: <label_value>  2
  type: <label_type>  3
    <match>  4
```

**1**    Node or pod label name.

**2**    Optional node or pod label value. If omitted, the presence of **<label_name>** is enough to match.

**3**     Optional object type (**node** or **pod**). If omitted, **node** is assumed.

**4**     An optional **<match>** list.

If **<match>** is not omitted, all nested **<match>** sections must also evaluate to **true**. Otherwise, **false** is assumed and the profile with the respective **<match>** section will not be applied or recommended. Therefore, the nesting (child **<match>** sections) works as logical AND operator. Conversely, if any item of the **<match>** list matches, the entire **<match>** list evaluates to **true**. Therefore, the list acts as logical OR operator.

If **machineConfigLabels** is defined, machine config pool based matching is turned on for the given **recommend:** list item. **<mcLabels>** specifies the labels for a machine config. The machine config is created automatically to apply host settings, such as kernel boot parameters, for the profile **<tuned_profile_name>**. This involves finding all machine config pools with machine config selector matching **<mcLabels>** and setting the profile **<tuned_profile_name>** on all nodes that are assigned the found machine config pools. To target nodes that have both master and worker roles, you must use the master role.

The list items **match** and **machineConfigLabels** are connected by the logical OR operator. The **match** item is evaluated first in a short-circuit manner. Therefore, if it evaluates to **true**, the **machineConfigLabels** item is not considered.

> **IMPORTANT**
>
> When using machine config pool based matching, it is advised to group nodes with the same hardware configuration into the same machine config pool. Not following this practice might result in TuneD operands calculating conflicting kernel parameters for two or more nodes sharing the same machine config pool.

**Example: node or pod label based matching**

```
- match:
  - label: tuned.openshift.io/elasticsearch
    match:
    - label: node-role.kubernetes.io/master
    - label: node-role.kubernetes.io/infra
    type: pod
  priority: 10
  profile: openshift-control-plane-es
- match:
  - label: node-role.kubernetes.io/master
  - label: node-role.kubernetes.io/infra
  priority: 20
  profile: openshift-control-plane
- priority: 30
  profile: openshift-node
```

The CR above is translated for the containerized TuneD daemon into its **recommend.conf** file based on the profile priorities. The profile with the highest priority (**10**) is **openshift-control-plane-es** and, therefore, it is considered first. The containerized TuneD daemon running on a given node looks to see if there is a pod running on the same node with the **tuned.openshift.io/elasticsearch** label set. If not, the entire **<match>** section evaluates as **false**. If there is such a pod with the label, in order for the **<match>** section to evaluate to **true**, the node label also needs to be **node-role.kubernetes.io/master** or **node-role.kubernetes.io/infra**.

If the labels for the profile with priority **10** matched, **openshift-control-plane-es** profile is applied and no other profile is considered. If the node/pod label combination did not match, the second highest priority profile (**openshift-control-plane**) is considered. This profile is applied if the containerized TuneD pod runs on a node with labels **node-role.kubernetes.io/master** or **node-role.kubernetes.io/infra**.

Finally, the profile **openshift-node** has the lowest priority of **30**. It lacks the **<match>** section and, therefore, will always match. It acts as a profile catch-all to set **openshift-node** profile, if no other profile with higher priority matches on a given node.
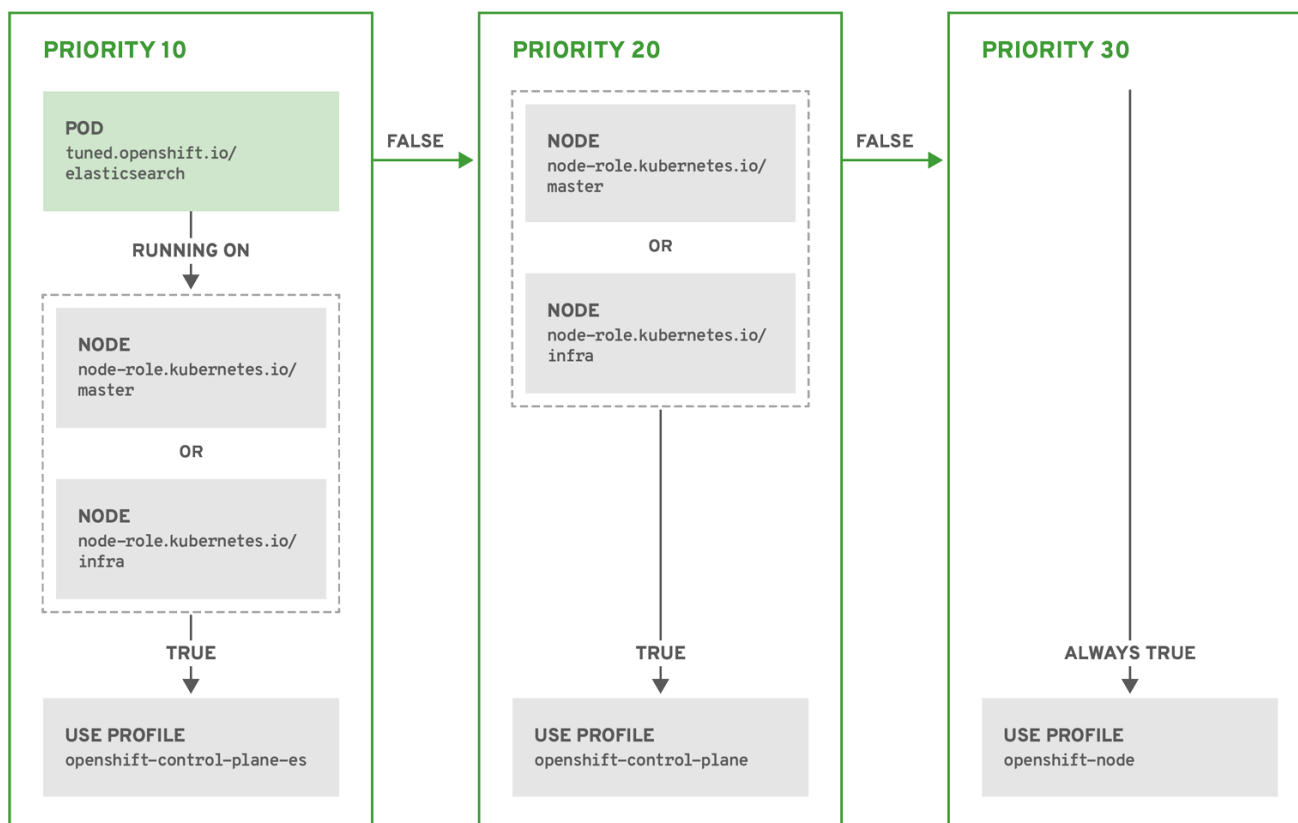


OPENSHIFT_10_0319

Example: machine config pool based matching

```
apiVersion: tuned.openshift.io/v1
kind: Tuned
metadata:
  name: openshift-node-custom
  namespace: openshift-cluster-node-tuning-operator
spec:
  profile:
  - data: |
      [main]
      summary=Custom OpenShift node profile with an additional kernel parameter
      include=openshift-node
      [bootloader]
      cmdline_openshift_node_custom=+skew_tick=1
    name: openshift-node-custom

  recommend:
  - machineConfigLabels:
```

```
      machineconfiguration.openshift.io/role: "worker-custom"
    priority: 20
    profile: openshift-node-custom
```

To minimize node reboots, label the target nodes with a label the machine config pool's node selector will match, then create the Tuned CR above and finally create the custom machine config pool itself.

## 5.5.3. Default profiles set on a cluster

The following are the default profiles set on a cluster.

```
apiVersion: tuned.openshift.io/v1
kind: Tuned
metadata:
  name: default
  namespace: openshift-cluster-node-tuning-operator
spec:
  profile:
  - name: "openshift"
    data: |
      [main]
      summary=Optimize systems running OpenShift (parent profile)
      include=${f:virt_check:virtual-guest:throughput-performance}

      [selinux]
      avc_cache_threshold=8192

      [net]
      nf_conntrack_hashsize=131072

      [sysctl]
      net.ipv4.ip_forward=1
      kernel.pid_max=>4194304
      net.netfilter.nf_conntrack_max=1048576
      net.ipv4.conf.all.arp_announce=2
      net.ipv4.neigh.default.gc_thresh1=8192
      net.ipv4.neigh.default.gc_thresh2=32768
      net.ipv4.neigh.default.gc_thresh3=65536
      net.ipv6.neigh.default.gc_thresh1=8192
      net.ipv6.neigh.default.gc_thresh2=32768
      net.ipv6.neigh.default.gc_thresh3=65536
      vm.max_map_count=262144

      [sysfs]
      /sys/module/nvme_core/parameters/io_timeout=4294967295
      /sys/module/nvme_core/parameters/max_retries=10

  - name: "openshift-control-plane"
    data: |
      [main]
      summary=Optimize systems running OpenShift control plane
      include=openshift

      [sysctl]
      # ktune sysctl settings, maximizing i/o throughput
```

```
    #
    # Minimal preemption granularity for CPU-bound tasks:
    # (default: 1 msec#  (1 + ilog(ncpus)), units: nanoseconds)
    kernel.sched_min_granularity_ns=10000000
    # The total time the scheduler will consider a migrated process
    # "cache hot" and thus less likely to be re-migrated
    # (system default is 500000, i.e. 0.5 ms)
    kernel.sched_migration_cost_ns=5000000
    # SCHED_OTHER wake-up granularity.
    #
    # Preemption granularity when tasks wake up.  Lower the value to
    # improve wake-up latency and throughput for latency critical tasks.
    kernel.sched_wakeup_granularity_ns=4000000

  - name: "openshift-node"
    data: |
      [main]
      summary=Optimize systems running OpenShift nodes
      include=openshift

      [sysctl]
      net.ipv4.tcp_fastopen=3
      fs.inotify.max_user_watches=65536
      fs.inotify.max_user_instances=8192

  recommend:
  - profile: "openshift-control-plane"
    priority: 30
    match:
    - label: "node-role.kubernetes.io/master"
    - label: "node-role.kubernetes.io/infra"

  - profile: "openshift-node"
    priority: 40
```

### 5.5.4. Supported TuneD daemon plugins

Excluding the **[main]** section, the following TuneD plugins are supported when using custom profiles defined in the **profile:** section of the Tuned CR:

- audio

- cpu

- disk

- eeepc_she

- modules

- mounts

- net

- scheduler

- scsi_host

- selinux

- sysctl

- sysfs

- usb

- video

- vm

There is some dynamic tuning functionality provided by some of these plugins that is not supported. The following TuneD plugins are currently not supported:

- bootloader

- script

- systemd

See Available TuneD Plugins and Getting Started with TuneD for more information.

## 5.6. REMEDIATING NODES WITH THE POISON PILL OPERATOR

You can use the Poison Pill Operator to automatically reboot unhealthy nodes. This remediation strategy minimizes downtime for stateful applications and ReadWriteOnce (RWO) volumes, and restores compute capacity if transient failures occur.

### 5.6.1. About the Poison Pill Operator

The Poison Pill Operator runs on the cluster nodes and reboots nodes that are identified as unhealthy. The Operator uses the **MachineHealthCheck** controller to detect the health of a node in the cluster. When a node is identified as unhealthy, the **MachineHealthCheck** resource creates the **PoisonPillRemediation** custom resource (CR), which triggers the Poison Pill Operator.

The Poison Pill Operator provides the following capabilities:

- Minimizes downtime for stateful applications and restores compute capacity if transient failures occur.

- Independent of any management interface, such as IPMI or an API to provision a node.

#### 5.6.1.1. Understanding the Poison Pill Operator configuration

The Poison Pill Operator creates the **PoisonPillConfig** CR with the name **poison-pill-config** in the Poison Pill Operator's namespace. You can edit this CR. However, you cannot create a new CR for the Poison Pill Operator.

A change in the **PoisonPillConfig** CR re-creates the Poison Pill daemon set.

The **PoisonPillConfig** CR resembles the following YAML file:

```
apiVersion: poison-pill.medik8s.io/v1alpha1
```

```
kind: PoisonPillConfig
metadata:
  name: poison-pill-config
  namespace: openshift-operators
spec:
  safeTimeToAssumeNodeRebootedSeconds: 180 1
  watchdogFilePath: /test/watchdog1 2
```

**1** Specify the timeout duration for the surviving peer, after which the Operator can assume that an unhealthy node has been rebooted. The Operator automatically calculates the lower limit for this value. However, if different nodes have different watchdog timeouts, you must change this value to a higher value.

**2** Specify the file path of the watchdog device in the nodes. If a watchdog device is unavailable, the **PoisonPillConfig** CR uses a software reboot.

## 5.6.2. Installing the Poison Pill Operator by using the web console

You can use the OpenShift Container Platform web console to install the Poison Pill Operator.

### Prerequisites

- Log in as a user with **cluster-admin** privileges.

### Procedure

1. In the OpenShift Container Platform web console, navigate to **Operators → OperatorHub**.

2. Search for the Poison Pill Operator from the list of available Operators, and then click **Install**.

3. Keep the default selection of **Installation mode** and **namespace** to ensure that the Operator is installed to the **poison-pill** namespace.

4. Click **Install**.

### Verification

To confirm that the installation is successful:

1. Navigate to the **Operators → Installed Operators** page.

2. Check that the Operator is installed in the **poison-pill** namespace and its status is **Succeeded**.

If the Operator is not installed successfully:

1. Navigate to the **Operators → Installed Operators** page and inspect the **Status** column for any errors or failures.

2. Navigate to the **Workloads → Pods** page and check the logs in any pods in the **poison-pill-controller-manager** project that are reporting issues.

## 5.6.3. Installing the Poison Pill Operator by using the CLI

You can use the OpenShift CLI (**oc**) to install the Poison Pill Operator.

Prerequisites

- Install the OpenShift CLI (**oc**).

- Log in as a user with **cluster-admin** privileges.

Procedure

1. Create a **Namespace** custom resource (CR) for the Poison Pill Operator:

    a. Define the **Namespace** CR and save the YAML file, for example, **poison-pill-namespace.yaml**:

    ```
    apiVersion: v1
    kind: Namespace
    metadata:
      name: poison-pill
    ```

    b. To create the **Namespace** CR, run the following command:

    ```
    $ oc create -f poison-pill-namespace.yaml
    ```

2. Create an **OperatorGroup** CR:

    a. Define the **OperatorGroup** CR and save the YAML file, for example, **poison-pill-operator-group.yaml**:

    ```
    apiVersion: operators.coreos.com/v1
    kind: OperatorGroup
    metadata:
      name: poison-pill-manager
      namespace: poison-pill
    spec:
      targetNamespaces:
      - poison-pill
    ```

    b. To create the **OperatorGroup** CR, run the following command:

    ```
    $ oc create -f poison-pill-operator-group.yaml
    ```

3. Create a **Subscription** CR:

    a. Define the **Subscription** CR and save the YAML file, for example, **poison-pill-subscription.yaml**:

    ```
    apiVersion: operators.coreos.com/v1alpha1
    kind: Subscription
    metadata:
        name: poison-pill-manager
        namespace: poison-pill
    spec:
        channel: alpha
        name: poison-pill-manager
    ```

```
      source: redhat-operators
      sourceNamespace: openshift-marketplace
      package: poison-pill-manager
```

b. To create the **Subscription** CR, run the following command:

```
$ oc create -f poison-pill-subscription.yaml
```

## Verification

1. Verify that the installation succeeded by inspecting the CSV resource:

```
$ oc get csv -n poison-pill
```

**Example output**

```
NAME             DISPLAY              VERSION  REPLACES   PHASE
poison-pill.v0.1.4   Poison Pill Operator   0.1.4               Succeeded
```

2. Verify that the Poison Pill Operator is up and running:

```
$ oc get deploy -n poison-pill
```

**Example output**

```
NAME                       READY  UP-TO-DATE  AVAILABLE  AGE
poison-pill-controller-manager    1/1    1           1          10d
```

3. Verify that the Poison Pill Operator created the **PoisonPillConfig** CR:

```
$ oc get PoisonPillConfig -n poison-pill
```

**Example output**

```
NAME             AGE
poison-pill-config   10d
```

4. Verify that each poison pill pod is scheduled and running on each worker node:

```
$ oc get daemonset -n poison-pill
```

**Example output**

```
NAME            DESIRED  CURRENT  READY  UP-TO-DATE  AVAILABLE  NODE
SELECTOR   AGE
poison-pill-ds  2        2        2      2           2          <none>       10d
```

> **NOTE**
>
> This command is unsupported for the control plane nodes.

### 5.6.4. Configuring machine health checks to use the Poison Pill Operator

Use the following procedure to configure the machine health checks to use the Poison Pill Operator as a remediation provider.

**Prerequisites**

- Install the OpenShift CLI (**oc**).

- Log in as a user with **cluster-admin** privileges.

**Procedure**

1. Create a **PoisonPillRemediationTemplate** CR:

   a. Define the **PoisonPillRemediationTemplate** CR:

   ```
   apiVersion: poison-pill.medik8s.io/v1alpha1
   kind: PoisonPillRemediationTemplate
   metadata:
     namespace: openshift-machine-api
     name: poisonpillremediationtemplate-sample
   spec:
     template:
       spec: {}
   ```

   b. To create the **PoisonPillRemediationTemplate** CR, run the following command:

   ```
   $ oc create -f <ppr-name>.yaml
   ```

2. Create or update the **MachineHealthCheck** CR to point to the **PoisonPillRemediationTemplate** CR:

   a. Define or update the **MachineHealthCheck** CR:

   ```
   apiVersion: machine.openshift.io/v1beta1
   kind: MachineHealthCheck
   metadata:
     name: machine-health-check
     namespace: openshift-machine-api
   spec:
     selector:
       matchLabels:
         machine.openshift.io/cluster-api-machine-role: "worker"
         machine.openshift.io/cluster-api-machine-type: "worker"
     unhealthyConditions:
     - type:    "Ready"
       timeout: "300s"
       status: "False"
     - type:    "Ready"
       timeout: "300s"
       status: "Unknown"
     maxUnhealthy: "40%"
     nodeStartupTimeout: "10m"
     remediationTemplate: ❶
   ```

```
kind: PoisonPillRemediationTemplate
apiVersion: poison-pill.medik8s.io/v1alpha1
name: <poison-pill-remediation-template-sample>
```

**1** Specify the details for the remediation template.

b. To create a **MachineHealthCheck** CR, run the following command:

```
$ oc create -f <file-name>.yaml
```

c. To update a **MachineHealthCheck** CR, run the following command:

```
$ oc apply -f <file-name>.yaml
```

## 5.6.5. Troubleshooting the Poison Pill Operator

### 5.6.5.1. General troubleshooting

**Issue**

You want to troubleshoot issues with the Poison Pill Operator.

**Resolution**

Check the Operator logs.

### 5.6.5.2. Checking the daemon set

**Issue**

The Poison Pill Operator is installed but the daemon set is not available.

**Resolution**

Check the Operator logs for errors or warnings.

### 5.6.5.3. Unsuccessful remediation

**Issue**

An unhealthy node was not remediated.

**Resolution**

Verify that the **PoisonPillRemediation** CR was created by running the following command:

```
$ oc get ppr -A
```

If the **MachineHealthCheck** controller did not create the **PoisonPillRemediation** CR when the node turned unhealthy, check the logs of the **MachineHealthCheck** controller. Additionally, ensure that the **MachineHealthCheck** CR includes the required specification to use the remediation template.

If the **PoisonPillRemediation** CR was created, ensure that its name matches the unhealthy node or the machine object.

### 5.6.5.4. Daemon set and other Poison Pill Operator resources exist even after uninstalling the Poison Pill Operator

**Issue**

The Poison Pill Operator resources, such as the daemon set, configuration CR, and the remediation template CR, exist even after after uninstalling the Operator.

**Resolution**

To remove the Poison Pill Operator resources, delete the resources by running the following commands for each resource type:

```
$ oc delete ds <poison-pill-ds> -n <namespace>
```

```
$ oc delete ppc <poison-pill-config> -n <namespace>
```

```
$ oc delete pprt <poison-pill-remediation-template> -n <namespace>
```

## 5.6.6. Additional resources

- The Poison Pill Operator is supported in a restricted network environment. For more information, see Using Operator Lifecycle Manager on restricted networks .

- Deleting Operators from a cluster

# 5.7. UNDERSTANDING NODE REBOOTING

To reboot a node without causing an outage for applications running on the platform, it is important to first evacuate the pods. For pods that are made highly available by the routing tier, nothing else needs to be done. For other pods needing storage, typically databases, it is critical to ensure that they can remain in operation with one pod temporarily going offline. While implementing resiliency for stateful pods is different for each application, in all cases it is important to configure the scheduler to use node anti-affinity to ensure that the pods are properly spread across available nodes.

Another challenge is how to handle nodes that are running critical infrastructure such as the router or the registry. The same node evacuation process applies, though it is important to understand certain edge cases.

## 5.7.1. About rebooting nodes running critical infrastructure

When rebooting nodes that host critical OpenShift Container Platform infrastructure components, such as router pods, registry pods, and monitoring pods, ensure that there are at least three nodes available to run these components.

The following scenario demonstrates how service interruptions can occur with applications running on OpenShift Container Platform when only two nodes are available:

- Node A is marked unschedulable and all pods are evacuated.

- The registry pod running on that node is now redeployed on node B. Node B is now running both registry pods.

- Node B is now marked unschedulable and is evacuated.

- The service exposing the two pod endpoints on node B loses all endpoints, for a brief period of time, until they are redeployed to node A.

When using three nodes for infrastructure components, this process does not result in a service disruption. However, due to pod scheduling, the last node that is evacuated and brought back into rotation does not have a registry pod. One of the other nodes has two registry pods. To schedule the third registry pod on the last node, use pod anti-affinity to prevent the scheduler from locating two registry pods on the same node.

### Additional information

- For more information on pod anti-affinity, see Placing pods relative to other pods using affinity and anti-affinity rules.

## 5.7.2. Rebooting a node using pod anti-affinity

Pod anti-affinity is slightly different than node anti-affinity. Node anti-affinity can be violated if there are no other suitable locations to deploy a pod. Pod anti-affinity can be set to either required or preferred.

With this in place, if only two infrastructure nodes are available and one is rebooted, the container image registry pod is prevented from running on the other node. **oc get pods** reports the pod as unready until a suitable node is available. Once a node is available and all pods are back in ready state, the next node can be restarted.

### Procedure

To reboot a node using pod anti-affinity:

1. Edit the node specification to configure pod anti-affinity:

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-antiaffinity
spec:
  affinity:
    podAntiAffinity: 1
      preferredDuringSchedulingIgnoredDuringExecution: 2
      - weight: 100 3
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: registry 4
              operator: In 5
              values:
              - default
          topologyKey: kubernetes.io/hostname
```

**1**    Stanza to configure pod anti-affinity.

**2**    Defines a preferred rule.

**3**    Specifies a weight for a preferred rule. The node with the highest weight is preferred.

**4** Description of the pod label that determines when the anti-affinity rule applies. Specify a key and value for the label.

**5** The operator represents the relationship between the label on the existing pod and the set of values in the **matchExpression** parameters in the specification for the new pod. Can be **In**, **NotIn**, **Exists**, or **DoesNotExist**.

This example assumes the container image registry pod has a label of **registry=default**. Pod anti-affinity can use any Kubernetes match expression.

2. Enable the **MatchInterPodAffinity** scheduler predicate in the scheduling policy file.

3. Perform a graceful restart of the node.

### 5.7.3. Understanding how to reboot nodes running routers

In most cases, a pod running an OpenShift Container Platform router exposes a host port.

The **PodFitsPorts** scheduler predicate ensures that no router pods using the same port can run on the same node, and pod anti-affinity is achieved. If the routers are relying on IP failover for high availability, there is nothing else that is needed.

For router pods relying on an external service such as AWS Elastic Load Balancing for high availability, it is that service's responsibility to react to router pod restarts.

In rare cases, a router pod may not have a host port configured. In those cases, it is important to follow the recommended restart process for infrastructure nodes.

### 5.7.4. Rebooting a node gracefully

Before rebooting a node, it is recommended to backup etcd data to avoid any data loss on the node.

#### Procedure

To perform a graceful restart of a node:

1. Mark the node as unschedulable:

   ```
   $ oc adm cordon <node1>
   ```

2. Drain the node to remove all the running pods:

   ```
   $ oc adm drain <node1> --ignore-daemonsets --delete-emptydir-data
   ```

   You might receive errors that pods associated with custom pod disruption budgets (PDB) cannot be evicted.

   **Example error**

   ```
   error when evicting pods/"rails-postgresql-example-1-72v2w" -n "rails" (will retry after 5s):
   Cannot evict pod as it would violate the pod's disruption budget.
   ```

   In this case, run the drain command again, adding the **disable-eviction** flag, which bypasses the PDB checks:

```
$ oc adm drain <node1> --ignore-daemonsets --delete-emptydir-data --force --disable-
eviction
```

3. Access the node in debug mode:

```
$ oc debug node/<node1>
```

4. Change your root directory to **/host**:

```
$ chroot /host
```

5. Restart the node:

```
$ systemctl reboot
```

In a moment, the node enters the **NotReady** state.

6. After the reboot is complete, mark the node as schedulable by running the following command:

```
$ oc adm uncordon <node1>
```

7. Verify that the node is ready:

```
$ oc get node <node1>
```

**Example output**

```
NAME    STATUS ROLES   AGE    VERSION
<node1> Ready   worker  6d22h  v1.18.3+b0068a8
```

## Additional information

For information on etcd data backup, see Backing up etcd data.

## 5.8. FREEING NODE RESOURCES USING GARBAGE COLLECTION

As an administrator, you can use OpenShift Container Platform to ensure that your nodes are running efficiently by freeing up resources through garbage collection.

The OpenShift Container Platform node performs two types of garbage collection:

- Container garbage collection: Removes terminated containers.

- Image garbage collection: Removes images not referenced by any running pods.

### 5.8.1. Understanding how terminated containers are removed through garbage collection

Container garbage collection can be performed using eviction thresholds.

When eviction thresholds are set for garbage collection, the node tries to keep any container for any pod accessible from the API. If the pod has been deleted, the containers will be as well. Containers are preserved as long the pod is not deleted and the eviction threshold is not reached. If the node is under

disk pressure, it will remove containers and their logs will no longer be accessible using **oc logs**.

- **eviction-soft** - A soft eviction threshold pairs an eviction threshold with a required administrator-specified grace period.

- **eviction-hard** - A hard eviction threshold has no grace period, and if observed, OpenShift Container Platform takes immediate action.

The following table lists the eviction thresholds:

**Table 5.2. Variables for configuring container garbage collection**

| Node condition | Eviction signal | Description |
| --- | --- | --- |
| MemoryPressure | **memory.available** | The available memory on the node. |
| DiskPressure | <br>- **nodefs.available**<br><br>- **nodefs.inodesFree**<br><br>- **imagefs.available**<br><br>- **imagefs.inodesFree** | The available disk space or inodes on the node root file system, **nodefs**, or image file system, **imagefs**. |

> **NOTE**
>
> For **evictionHard** you must specify all of these parameters. If you do not specify all parameters, only the specified parameters are applied and the garbage collection will not function properly.

If a node is oscillating above and below a soft eviction threshold, but not exceeding its associated grace period, the corresponding node would constantly oscillate between **true** and **false**. As a consequence, the scheduler could make poor scheduling decisions.

To protect against this oscillation, use the **eviction-pressure-transition-period** flag to control how long OpenShift Container Platform must wait before transitioning out of a pressure condition. OpenShift Container Platform will not set an eviction threshold as being met for the specified pressure condition for the period specified before toggling the condition back to false.

## 5.8.2. Understanding how images are removed through garbage collection

Image garbage collection relies on disk usage as reported by **cAdvisor** on the node to decide which images to remove from the node.

The policy for image garbage collection is based on two conditions:

- The percent of disk usage (expressed as an integer) which triggers image garbage collection. The default is **85**.

- The percent of disk usage (expressed as an integer) to which image garbage collection attempts to free. Default is **80**.

For image garbage collection, you can modify any of the following variables using a custom resource.

Table 5.3. Variables for configuring image garbage collection

| Setting | Description |
| --- | --- |
| **imageMinimumGCAge** | The minimum age for an unused image before the image is removed by garbage collection. The default is **2m**. |
| **imageGCHighThresholdPercent** | The percent of disk usage, expressed as an integer, which triggers image garbage collection. The default is **85**. |
| **imageGCLowThresholdPercent** | The percent of disk usage, expressed as an integer, to which image garbage collection attempts to free. The default is **80**. |

Two lists of images are retrieved in each garbage collector run:

1. A list of images currently running in at least one pod.

2. A list of images available on a host.

As new containers are run, new images appear. All images are marked with a time stamp. If the image is running (the first list above) or is newly detected (the second list above), it is marked with the current time. The remaining images are already marked from the previous spins. All images are then sorted by the time stamp.

Once the collection starts, the oldest images get deleted first until the stopping criterion is met.

## 5.8.3. Configuring garbage collection for containers and images

As an administrator, you can configure how OpenShift Container Platform performs garbage collection by creating a **kubeletConfig** object for each machine config pool.

> **NOTE**
>
> OpenShift Container Platform supports only one **kubeletConfig** object for each machine config pool.

You can configure any combination of the following:

- Soft eviction for containers

- Hard eviction for containers

- Eviction for images

**Prerequisites**

1. Obtain the label associated with the static **MachineConfigPool** CRD for the type of node you want to configure by entering the following command:

   ```
   $ oc edit machineconfigpool <name>
   ```

For example:

```
$ oc edit machineconfigpool worker
```

## Example output

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  creationTimestamp: "2022-11-16T15:34:25Z"
  generation: 4
  labels:
    pools.operator.machineconfiguration.openshift.io/worker: ""
  name: worker
```

**1** The label appears under Labels.

### TIP

If the label is not present, add a key/value pair such as:

```
$ oc label machineconfigpool worker custom-kubelet=small-pods
```

## Procedure

1. Create a custom resource (CR) for your configuration change.

   ### IMPORTANT

   If there is one file system, or if **/var/lib/kubelet** and **/var/lib/containers/** are in the same file system, the settings with the highest values trigger evictions, as those are met first. The file system triggers the eviction.

   ### Sample configuration for a container garbage collection CR:

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: worker-kubeconfig 1
spec:
  machineConfigPoolSelector:
    matchLabels:
      pools.operator.machineconfiguration.openshift.io/worker: "" 2
  kubeletConfig:
    evictionSoft: 3
      memory.available: "500Mi" 4
      nodefs.available: "10%"
      nodefs.inodesFree: "5%"
      imagefs.available: "15%"
      imagefs.inodesFree: "10%"
    evictionSoftGracePeriod: 5
```

```
      memory.available: "1m30s"
      nodefs.available: "1m30s"
      nodefs.inodesFree: "1m30s"
      imagefs.available: "1m30s"
      imagefs.inodesFree: "1m30s"
    evictionHard: 6
      memory.available: "200Mi"
      nodefs.available: "5%"
      nodefs.inodesFree: "4%"
      imagefs.available: "10%"
      imagefs.inodesFree: "5%"
    evictionPressureTransitionPeriod: 0s 7
    imageMinimumGCAge: 5m 8
    imageGCHighThresholdPercent: 80 9
    imageGCLowThresholdPercent: 75 10
```

**1** Name for the object.

**2** Specify the label from the machine config pool.

**3** Type of eviction: **evictionSoft** or **evictionHard**.

**4** Eviction thresholds based on a specific eviction trigger signal.

**5** Grace periods for the soft eviction. This parameter does not apply to **eviction-hard**.

**6** Eviction thresholds based on a specific eviction trigger signal. For **evictionHard** you must specify all of these parameters. If you do not specify all parameters, only the specified parameters are applied and the garbage collection will not function properly.

**7** The duration to wait before transitioning out of an eviction pressure condition.

**8** The minimum age for an unused image before the image is removed by garbage collection.

**9** The percent of disk usage (expressed as an integer) that triggers image garbage collection.

**10** The percent of disk usage (expressed as an integer) that image garbage collection attempts to free.

2. Run the following command to create the CR:

```
$ oc create -f <file_name>.yaml
```

For example:

```
$ oc create -f gc-container.yaml
```

**Example output**

```
kubeletconfig.machineconfiguration.openshift.io/gc-container created
```

**Verification**

1. Verify that garbage collection is active by entering the following command. The Machine Config Pool you specified in the custom resource appears with **UPDATING** as 'true` until the change is fully implemented:

```
$ oc get machineconfigpool
```

**Example output**

```
NAME     CONFIG                                UPDATED   UPDATING
master   rendered-master-546383f80705bd5aeaba93   True      False
worker   rendered-worker-b4c51bb33ccaae6fc4a6a5   False     True
```

# 5.9. ALLOCATING RESOURCES FOR NODES IN AN OPENSHIFT CONTAINER PLATFORM CLUSTER

To provide more reliable scheduling and minimize node resource overcommitment, reserve a portion of the CPU and memory resources for use by the underlying node components, such as **kubelet** and **kube-proxy**, and the remaining system components, such as **sshd** and **NetworkManager**. By specifying the resources to reserve, you provide the scheduler with more information about the remaining CPU and memory resources that a node has available for use by pods. You can allow OpenShift Container Platform to automatically determine the optimal **system-reserved** CPU and memory resources for your nodes or you can manually determine and set the best resources for your nodes.

## 5.9.1. Understanding how to allocate resources for nodes

CPU and memory resources reserved for node components in OpenShift Container Platform are based on two node settings:

| Setting | Description |
| --- | --- |
| **kube-reserved** | This setting is not used with OpenShift Container Platform. Add the CPU and memory resources that you planned to reserve to the **system-reserved** setting. |
| **system-reserved** | This setting identifies the resources to reserve for the node components and system components, such as CRI-O and Kubelet. The default settings depend on the OpenShift Container Platform and Machine Config Operator versions. Confirm the default **systemReserved** parameter on the **machine-config-operator** repository. |

If a flag is not set, the defaults are used. If none of the flags are set, the allocated resource is set to the node's capacity as it was before the introduction of allocatable resources.

> **NOTE**
>
> Any CPUs specifically reserved using the **reservedSystemCPUs** parameter are not available for allocation using **kube-reserved** or **system-reserved**.

### 5.9.1.1. How OpenShift Container Platform computes allocated resources

An allocated amount of a resource is computed based on the following formula:

> [Allocatable] = [Node Capacity] - [system-reserved] - [Hard-Eviction-Thresholds]

**NOTE**

The withholding of **Hard-Eviction-Thresholds** from **Allocatable** improves system reliability because the value for **Allocatable** is enforced for pods at the node level.

If **Allocatable** is negative, it is set to **0**.

Each node reports the system resources that are used by the container runtime and kubelet. To simplify configuring the **system-reserved** parameter, view the resource use for the node by using the node summary API. The node summary is available at **/api/v1/nodes/<node>/proxy/stats/summary**.

### 5.9.1.2. How nodes enforce resource constraints

The node is able to limit the total amount of resources that pods can consume based on the configured allocatable value. This feature significantly improves the reliability of the node by preventing pods from using CPU and memory resources that are needed by system services such as the container runtime and node agent. To improve node reliability, administrators should reserve resources based on a target for resource use.

The node enforces resource constraints by using a new cgroup hierarchy that enforces quality of service. All pods are launched in a dedicated cgroup hierarchy that is separate from system daemons.

Administrators should treat system daemons similar to pods that have a guaranteed quality of service. System daemons can burst within their bounding control groups and this behavior must be managed as part of cluster deployments. Reserve CPU and memory resources for system daemons by specifying the amount of CPU and memory resources in **system-reserved**.

Enforcing **system-reserved** limits can prevent critical system services from receiving CPU and memory resources. As a result, a critical system service can be ended by the out-of-memory killer. The recommendation is to enforce **system-reserved** only if you have profiled the nodes exhaustively to determine precise estimates and you are confident that critical system services can recover if any process in that group is ended by the out-of-memory killer.

### 5.9.1.3. Understanding Eviction Thresholds

If a node is under memory pressure, it can impact the entire node and all pods running on the node. For example, a system daemon that uses more than its reserved amount of memory can trigger an out-of-memory event. To avoid or reduce the probability of system out-of-memory events, the node provides out-of-resource handling.

You can reserve some memory using the **--eviction-hard** flag. The node attempts to evict pods whenever memory availability on the node drops below the absolute value or percentage. If system daemons do not exist on a node, pods are limited to the memory **capacity - eviction-hard**. For this reason, resources set aside as a buffer for eviction before reaching out of memory conditions are not available for pods.

The following is an example to illustrate the impact of node allocatable for memory:

- Node capacity is **32Gi**

- --system-reserved is **3Gi**

- --eviction-hard is set to **100Mi**.

For this node, the effective node allocatable value is **28.9Gi**. If the node and system components use all their reservation, the memory available for pods is **28.9Gi**, and kubelet evicts pods when it exceeds this threshold.

If you enforce node allocatable, **28.9Gi**, with top-level cgroups, then pods can never exceed **28.9Gi**. Evictions are not performed unless system daemons consume more than **3.1Gi** of memory.

If system daemons do not use up all their reservation, with the above example, pods would face memcg OOM kills from their bounding cgroup before node evictions kick in. To better enforce QoS under this situation, the node applies the hard eviction thresholds to the top-level cgroup for all pods to be **Node Allocatable + Eviction Hard Thresholds**.

If system daemons do not use up all their reservation, the node will evict pods whenever they consume more than **28.9Gi** of memory. If eviction does not occur in time, a pod will be OOM killed if pods consume **29Gi** of memory.

### 5.9.1.4. How the scheduler determines resource availability

The scheduler uses the value of **node.Status.Allocatable** instead of **node.Status.Capacity** to decide if a node will become a candidate for pod scheduling.

By default, the node will report its machine capacity as fully schedulable by the cluster.

## 5.9.2. Automatically allocating resources for nodes

OpenShift Container Platform can automatically determine the optimal **system-reserved** CPU and memory resources for nodes associated with a specific machine config pool and update the nodes with those values when the nodes start. By default, the **system-reserved** CPU is **500m** and **system-reserved** memory is **1Gi**.

To automatically determine and allocate the **system-reserved** resources on nodes, create a **KubeletConfig** custom resource (CR) to set the **autoSizingReserved: true** parameter. A script on each node calculates the optimal values for the respective reserved resources based on the installed CPU and memory capacity on each node. The script takes into account that increased capacity requires a corresponding increase in the reserved resources.

Automatically determining the optimal **system-reserved** settings ensures that your cluster is running efficiently and prevents node failure due to resource starvation of system components, such as CRI-O and kubelet, without your needing to manually calculate and update the values.

This feature is disabled by default.

**Prerequisites**

1. Obtain the label associated with the static **MachineConfigPool** object for the type of node you want to configure by entering the following command:

   ```
   $ oc edit machineconfigpool <name>
   ```

   For example:

   ```
   $ oc edit machineconfigpool worker
   ```

   **Example output**

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
 creationTimestamp: "2022-11-16T15:34:25Z"
 generation: 4
 labels:
  pools.operator.machineconfiguration.openshift.io/worker: ""  1
 name: worker
...
```

**1**     The label appears under **Labels**.

### TIP

If the label is not present, add a key/value pair such as:

```
$ oc label machineconfigpool worker custom-kubelet=small-pods
```

### Procedure

1. Create a custom resource (CR) for your configuration change:

   **Sample configuration for a resource allocation CR**

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: KubeletConfig
   metadata:
    name: dynamic-node  1
   spec:
    autoSizingReserved: true  2
    machineConfigPoolSelector:
     matchLabels:
      pools.operator.machineconfiguration.openshift.io/worker: ""  3
   ```

   **1**     Assign a name to CR.

   **2**     Add the **autoSizingReserved** parameter set to **true** to allow OpenShift Container Platform to automatically determine and allocate the **system-reserved** resources on the nodes associated with the specified label. To disable automatic allocation on those nodes, set this parameter to **false**.

   **3**     Specify the label from the machine config pool.

   The previous example enables automatic resource allocation on all worker nodes. OpenShift Container Platform drains the nodes, applies the kubelet config, and restarts the nodes.

2. Create the CR by entering the following command:

   ```
   $ oc create -f <file_name>.yaml
   ```

### Verification

1. Log in to a node you configured by entering the following command:

   ```
   $ oc debug node/<node_name>
   ```

2. Set /**host** as the root directory within the debug shell:

   ```
   # chroot /host
   ```

3. View the /**etc**/**node-sizing.env** file:

   **Example output**

   ```
   SYSTEM_RESERVED_MEMORY=3Gi
   SYSTEM_RESERVED_CPU=0.08
   ```

   The kubelet uses the **system-reserved** values in the /**etc**/**node-sizing.env** file. In the previous example, the worker nodes are allocated **0.08** CPU and 3 Gi of memory. It can take several minutes for the optimal values to appear.

## 5.9.3. Manually allocating resources for nodes

OpenShift Container Platform supports the CPU and memory resource types for allocation. The **ephemeral-resource** resource type is supported as well. For the **cpu** type, the resource quantity is specified in units of cores, such as **200m**, **0.5**, or **1**. For **memory** and **ephemeral-storage**, it is specified in units of bytes, such as **200Ki**, **50Mi**, or **5Gi**. By default, the **system-reserved** CPU is **500m** and **system-reserved** memory is **1Gi**.

As an administrator, you can set these using a custom resource (CR) through a set of **<resource_type>=<resource_quantity>** pairs (e.g., **cpu=200m,memory=512Mi**).

For details on the recommended **system-reserved** values, refer to the recommended system-reserved values.

**Prerequisites**

1. Obtain the label associated with the static **MachineConfigPool** CRD for the type of node you want to configure by entering the following command:

   ```
   $ oc edit machineconfigpool <name>
   ```

   For example:

   ```
   $ oc edit machineconfigpool worker
   ```

   **Example output**

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: MachineConfigPool
   metadata:
     creationTimestamp: "2022-11-16T15:34:25Z"
     generation: 4
   ```

```
        labels:
          pools.operator.machineconfiguration.openshift.io/worker: "" 1
        name: worker
```

**1**     The label appears under Labels.

**TIP**

If the label is not present, add a key/value pair such as:

```
$ oc label machineconfigpool worker custom-kubelet=small-pods
```

**Procedure**

1. Create a custom resource (CR) for your configuration change.

   **Sample configuration for a resource allocation CR**

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: KubeletConfig
   metadata:
     name: set-allocatable 1
   spec:
     machineConfigPoolSelector:
       matchLabels:
         pools.operator.machineconfiguration.openshift.io/worker: "" 2
     kubeletConfig:
       systemReserved: 3
         cpu: 1000m
         memory: 1Gi
   ```

   **1**     Assign a name to CR.

   **2**     Specify the label from the machine config pool.

   **3**     Specify the resources to reserve for the node components and system components.

2. Run the following command to create the CR:

   ```
   $ oc create -f <file_name>.yaml
   ```

# 5.10. ALLOCATING SPECIFIC CPUS FOR NODES IN A CLUSTER

When using the static CPU Manager policy, you can reserve specific CPUs for use by specific nodes in your cluster. For example, on a system with 24 CPUs, you could reserve CPUs numbered 0 – 3 for the control plane allowing the compute nodes to use CPUs 4 – 23.

## 5.10.1. Reserving CPUs for nodes

To explicitly define a list of CPUs that are reserved for specific nodes, create a **KubeletConfig** custom resource (CR) to define the **reservedSystemCPUs** parameter. This list supersedes the CPUs that might be reserved using the **systemReserved** and **kubeReserved** parameters.

**Procedure**

1. Obtain the label associated with the machine config pool (MCP) for the type of node you want to configure:

   ```
   $ oc describe machineconfigpool <name>
   ```

   For example:

   ```
   $ oc describe machineconfigpool worker
   ```

   **Example output**

   ```
   Name:        worker
   Namespace:
   Labels:      machineconfiguration.openshift.io/mco-built-in=
                pools.operator.machineconfiguration.openshift.io/worker=  1
   Annotations: <none>
   API Version: machineconfiguration.openshift.io/v1
   Kind:        MachineConfigPool
   ...
   ```

   **1**  Get the MCP label.

2. Create a YAML file for the **KubeletConfig** CR:

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: KubeletConfig
   metadata:
     name: set-reserved-cpus  1
   spec:
     kubeletConfig:
       reservedSystemCPUs: "0,1,2,3"  2
     machineConfigPoolSelector:
       matchLabels:
         pools.operator.machineconfiguration.openshift.io/worker: ""  3
   ```

   **1**  Specify a name for the CR.

   **2**  Specify the core IDs of the CPUs you want to reserve for the nodes associated with the MCP.

   **3**  Specify the label from the MCP.

3. Create the CR object:

   ```
   $ oc create -f <file_name>.yaml
   ```

**Additional resources**

- For more information on the **systemReserved** and **kubeReserved** parameters, see Allocating resources for nodes in an OpenShift Container Platform cluster.

# 5.11. ENABLING TLS SECURITY PROFILES FOR THE KUBELET

You can use a TLS (Transport Layer Security) security profile to define which TLS ciphers are required by the kubelet when it is acting as an HTTP server. The kubelet uses its HTTP/GRPC server to communicate with the Kubernetes API server, which sends commands to pods, gathers logs, and run exec commands on pods through the kubelet.

A TLS security profile defines the TLS ciphers that the Kubernetes API server must use when connecting with the kubelet to protect communication between the kubelet and the Kubernetes API server.

> **NOTE**
>
> By default, when the kubelet acts as a client with the Kubernetes API server, it automatically negotiates the TLS parameters with the API server.

## 5.11.1. Understanding TLS security profiles

You can use a TLS (Transport Layer Security) security profile to define which TLS ciphers are required by various OpenShift Container Platform components. The OpenShift Container Platform TLS security profiles are based on Mozilla recommended configurations.

You can specify one of the following TLS security profiles for each component:

Table 5.4. TLS security profiles

| Profile | Description |
| --- | --- |
| **Old** | This profile is intended for use with legacy clients or libraries. The profile is based on the Old backward compatibility recommended configuration.<br><br>The **Old** profile requires a minimum TLS version of 1.0.<br><br>> **NOTE**<br>><br>> For the Ingress Controller, the minimum TLS version is converted from 1.0 to 1.1. |
| **Intermediate** | This profile is the recommended configuration for the majority of clients. It is the default TLS security profile for the Ingress Controller, kubelet, and control plane. The profile is based on the Intermediate compatibility recommended configuration.<br><br>The **Intermediate** profile requires a minimum TLS version of 1.2. |

| Profile | Description |
| --- | --- |
| **Modern** | This profile is intended for use with modern clients that have no need for backwards compatibility. This profile is based on the Modern compatibility recommended configuration.<br><br>The **Modern** profile requires a minimum TLS version of 1.3.<br><br>**NOTE**<br><br>In OpenShift Container Platform 4.6, 4.7, and 4.8, the **Modern** profile is unsupported. If selected, the **Intermediate** profile is enabled.<br><br>**IMPORTANT**<br><br>The **Modern** profile is currently not supported. |
| **Custom** | This profile allows you to define the TLS version and ciphers to use.<br><br>**WARNING**<br><br>Use caution when using a **Custom** profile, because invalid configurations can cause problems.<br><br>**NOTE**<br><br>OpenShift Container Platform router enables Red Hat-distributed OpenSSL default set of TLS **1.3** cipher suites. Your cluster might accept TLS **1.3** connections and cipher suites, even though TLS **1.3** is unsupported in OpenShift Container Platform 4.6, 4.7, and 4.8. |

**NOTE**

When using one of the predefined profile types, the effective profile configuration is subject to change between releases. For example, given a specification to use the Intermediate profile deployed on release X.Y.Z, an upgrade to release X.Y.Z+1 might cause a new profile configuration to be applied, resulting in a rollout.

## 5.11.2. Configuring the TLS security profile for the kubelet

To configure a TLS security profile for the kubelet when it is acting as an HTTP server, create a **KubeletConfig** custom resource (CR) to specify a predefined or custom TLS security profile for specific nodes. If a TLS security profile is not configured, the default TLS security profile is **Intermediate**.

**Sample KubeletConfig CR that configures the Old TLS security profile on worker nodes**

```
apiVersion: config.openshift.io/v1
kind: KubeletConfig
 ...
spec:
 tlsSecurityProfile:
   old: {}
   type: Old
 machineConfigPoolSelector:
   matchLabels:
     pools.operator.machineconfiguration.openshift.io/worker: ""
```

You can see the ciphers and the minimum TLS version of the configured TLS security profile in the **kubelet.conf** file on a configured node.

**Prerequisites**

- You have access to the cluster as a user with the **cluster-admin** role.

**Procedure**

1. Create a **KubeletConfig** CR to configure the TLS security profile:

   **Sample KubeletConfig CR for a Custom profile**

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: KubeletConfig
   metadata:
     name: set-kubelet-tls-security-profile
   spec:
    tlsSecurityProfile:
      type: Custom ❶
      custom: ❷
        ciphers: ❸
        - ECDHE-ECDSA-CHACHA20-POLY1305
        - ECDHE-RSA-CHACHA20-POLY1305
        - ECDHE-RSA-AES128-GCM-SHA256
        - ECDHE-ECDSA-AES128-GCM-SHA256
        minTLSVersion: VersionTLS11
    machineConfigPoolSelector:
      matchLabels:
        pools.operator.machineconfiguration.openshift.io/worker: "" ❹
   ```

   ❶ Specify the TLS security profile type (**Old**, **Intermediate**, or **Custom**). The default is **Intermediate**.

   ❷ Specify the appropriate field for the selected type:

      - **old: {}**

      - **intermediate: {}**

      - **custom:**

**3** For the **custom** type, specify a list of TLS ciphers and minimum accepted TLS version.

**4** Optional: Specify the machine config pool label for the nodes you want to apply the TLS security profile.

2. Create the **KubeletConfig** object:

```
$ oc create -f <filename>
```

Depending on the number of worker nodes in the cluster, wait for the configured nodes to be rebooted one by one.

## Verification

To verify that the profile is set, perform the following steps after the nodes are in the **Ready** state:

1. Start a debug session for a configured node:

```
$ oc debug node/<node_name>
```

2. Set /**host** as the root directory within the debug shell:

```
sh-4.4# chroot /host
```

3. View the **kubelet.conf** file:

```
sh-4.4# cat /etc/kubernetes/kubelet.conf
```

**Example output**

```
kind: KubeletConfiguration
apiVersion: kubelet.config.k8s.io/v1beta1
 ...
  "tlsCipherSuites": [
    "TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256",
    "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256",
    "TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384",
    "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384",
    "TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256",
    "TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256"
  ],
  "tlsMinVersion": "VersionTLS12",
```

# 5.12. MACHINE CONFIG DAEMON METRICS

The Machine Config Daemon is a part of the Machine Config Operator. It runs on every node in the cluster. The Machine Config Daemon manages configuration changes and updates on each of the nodes.

## 5.12.1. Machine Config Daemon metrics

Beginning with OpenShift Container Platform 4.3, the Machine Config Daemon provides a set of metrics. These metrics can be accessed using the Prometheus Cluster Monitoring stack.

The following table describes this set of metrics.
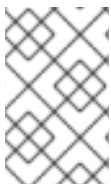
> **NOTE**
>
> Metrics marked with **\*** in the *Name* and **Description** columns represent serious errors that might cause performance problems. Such problems might prevent updates and upgrades from proceeding.
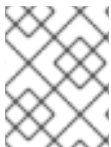
> **NOTE**
>
> While some entries contain commands for getting specific logs, the most comprehensive set of logs is available using the **oc adm must-gather** command.

Table 5.5. MCO metrics

| Name | Format | Description | Notes |
|------|--------|-------------|-------|
| **mcd_host_os_and_version** | **[]string{"os", "version"}** | Shows the OS that MCD is running on, such as RHCOS or RHEL. In case of RHCOS, the version is provided. | |
| **ssh_accessed** | **counter** | Shows the number of successful SSH authentications into the node. | The non-zero value shows that someone might have made manual changes to the node. Such changes might cause irreconcilable errors due to the differences between the state on the disk and the state defined in the machine configuration. |
| **mcd_drain\*** | **{"drain_time", "err"}** | Logs errors received during failed drain. * | While drains might need multiple tries to succeed, terminal failed drains prevent updates from proceeding. The **drain_time** metric, which shows how much time the drain took, might help with troubleshooting.<br><br>For further investigation, see the logs by running:<br><br>**$ oc logs -f -n openshift-machine-config-operator machine-config-daemon-\<hash\> -c machine-config-daemon** |

| Name | Format | Description | Notes |
|------|--------|-------------|-------|
| **mcd_pivot_err\*** | **[]string{"pivot_target", "err"}** | Logs errors encountered during pivot. * | Pivot errors might prevent OS upgrades from proceeding.<br><br>For further investigation, run this command to access the node and see all its logs:<br><br>**$ oc debug node/<node> — chroot /host journalctl -u pivot.service**<br><br>Alternatively, you can run this command to only see the logs from the **machine-config-daemon** container:<br><br>**$ oc logs -f -n openshift-machine-config-operator machine-config-daemon-<hash> -c machine-config-daemon** |
| **mcd_state** | **[]string{"state", "reason"}** | State of Machine Config Daemon for the indicated node. Possible states are "Done", "Working", and "Degraded". In case of "Degraded", the reason is included. | For further investigation, see the logs by running:<br><br>**$ oc logs -f -n openshift-machine-config-operator machine-config-daemon-<hash> -c machine-config-daemon** |
| **mcd_kubelet_state\*** | **[]string{"err"}** | Logs kubelet health failures. * | This is expected to be empty, with failure count of 0. If failure count exceeds 2, the error indicating threshold is exceeded. This indicates a possible issue with the health of the kubelet.<br><br>For further investigation, run this command to access the node and see all its logs:<br><br>**$ oc debug node/<node> — chroot /host journalctl -u kubelet** |

| Name | Format | Description | Notes |
|------|--------|-------------|-------|
| **mcd_reboot_err*** | **[]string{"message", "err"}** | Logs the failed reboots and the corresponding errors. * | This is expected to be empty, which indicates a successful reboot. <br><br> For further investigation, see the logs by running: <br><br> **$ oc logs -f -n openshift-machine-config-operator machine-config-daemon-\<hash\> -c machine-config-daemon** |
| **mcd_update_state** | **[]string{"config", "err"}** | Logs success or failure of configuration updates and the corresponding errors. | The expected value is **rendered-master/rendered-worker-XXXX**. If the update fails, an error is present. <br><br> For further investigation, see the logs by running: <br><br> **$ oc logs -f -n openshift-machine-config-operator machine-config-daemon-\<hash\> -c machine-config-daemon** |

**Additional resources**

- See Monitoring overview.

- See the documentation on gathering data about your cluster .

## 5.13. CREATING INFRASTRUCTURE NODES
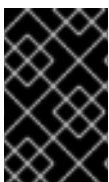


**IMPORTANT**

This process is not applicable for clusters with manually provisioned machines. You can use the advanced machine management and scaling capabilities only in clusters where the Machine API is operational.

You can use infrastructure machine sets to create machines that host only infrastructure components, such as the default router, the integrated container image registry, and the components for cluster metrics and monitoring. These infrastructure machines are not counted toward the total number of subscriptions that are required to run the environment.

In a production deployment, it is recommended that you deploy at least three machine sets to hold infrastructure components. Both OpenShift Logging and Red Hat OpenShift Service Mesh deploy Elasticsearch, which requires three instances to be installed on different nodes. Each of these nodes can

be deployed to different availability zones for high availability. This configuration requires three different machine sets, one for each availability zone. In global Azure regions that do not have multiple availability zones, you can use availability sets to ensure high availability.

## 5.13.1. OpenShift Container Platform infrastructure components

The following infrastructure workloads do not incur OpenShift Container Platform worker subscriptions:

- Kubernetes and OpenShift Container Platform control plane services that run on masters

- The default router

- The integrated container image registry

- The HAProxy-based Ingress Controller

- The cluster metrics collection, or monitoring service, including components for monitoring user-defined projects

- Cluster aggregated logging

- Service brokers

- Red Hat Quay

- Red Hat OpenShift Container Storage

- Red Hat Advanced Cluster Manager

- Red Hat Advanced Cluster Security for Kubernetes

- Red Hat OpenShift GitOps

- Red Hat OpenShift Pipelines

Any node that runs any other container, pod, or component is a worker node that your subscription must cover.

For information about infrastructure nodes and which components can run on infrastructure nodes, see the "Red Hat OpenShift control plane and infrastructure nodes" section in the OpenShift sizing and subscription guide for enterprise Kubernetes document.

To create an infrastructure node, you can use a machine set, label the node, or use a machine config pool.

### 5.13.1.1. Creating an infrastructure node

> **IMPORTANT**
>
> See Creating infrastructure machine sets for installer-provisioned infrastructure environments or for any cluster where the control plane nodes (also known as the master nodes) are managed by the machine API.

Requirements of the cluster dictate that infrastructure, also called **infra** nodes, be provisioned. The installer only provides provisions for control plane and worker nodes. Worker nodes can be designated as infrastructure nodes or application, also called **app**, nodes through labeling.

**Procedure**

1. Add a label to the worker node that you want to act as application node:

   ```
   $ oc label node <node-name> node-role.kubernetes.io/app=""
   ```

2. Add a label to the worker nodes that you want to act as infrastructure nodes:

   ```
   $ oc label node <node-name> node-role.kubernetes.io/infra=""
   ```

3. Check to see if applicable nodes now have the **infra** role and **app** roles:

   ```
   $ oc get nodes
   ```

4. Create a default cluster-wide node selector. The default node selector is applied to pods created in all namespaces. This creates an intersection with any existing node selectors on a pod, which additionally constrains the pod's selector.

   > **IMPORTANT**
   >
   > If the default node selector key conflicts with the key of a pod's label, then the default node selector is not applied.
   >
   > However, do not set a default node selector that might cause a pod to become unschedulable. For example, setting the default node selector to a specific node role, such as **node-role.kubernetes.io/infra=""**, when a pod's label is set to a different node role, such as **node-role.kubernetes.io/master=""**, can cause the pod to become unschedulable. For this reason, use caution when setting the default node selector to specific node roles.
   >
   > You can alternatively use a project node selector to avoid cluster-wide node selector key conflicts.

   a. Edit the **Scheduler** object:

      ```
      $ oc edit scheduler cluster
      ```

   b. Add the **defaultNodeSelector** field with the appropriate node selector:

      ```
      apiVersion: config.openshift.io/v1
      kind: Scheduler
      metadata:
        name: cluster
      ...
      spec:
        defaultNodeSelector: topology.kubernetes.io/region=us-east-1    1
      ...
      ```

      **1**    This example node selector deploys pods on nodes in the **us-east-1** region by default.

   c. Save the file to apply the changes.

You can now move infrastructure resources to the newly labeled **infra** nodes.

## Additional resources

- [Moving resources to infrastructure machine sets](#)

# CHAPTER 6. WORKING WITH CONTAINERS

## 6.1. UNDERSTANDING CONTAINERS

The basic units of OpenShift Container Platform applications are called *containers*. Linux container technologies are lightweight mechanisms for isolating running processes so that they are limited to interacting with only their designated resources.

Many application instances can be running in containers on a single host without visibility into each others' processes, files, network, and so on. Typically, each container provides a single service (often called a "micro-service"), such as a web server or a database, though containers can be used for arbitrary workloads.

The Linux kernel has been incorporating capabilities for container technologies for years. OpenShift Container Platform and Kubernetes add the ability to orchestrate containers across multi-host installations.

**About containers and RHEL kernel memory**

Due to Red Hat Enterprise Linux (RHEL) behavior, a container on a node with high CPU usage might seem to consume more memory than expected. The higher memory consumption could be caused by the **kmem_cache** in the RHEL kernel. The RHEL kernel creates a **kmem_cache** for each cgroup. For added performance, the **kmem_cache** contains a **cpu_cache**, and a node cache for any NUMA nodes. These caches all consume kernel memory.

The amount of memory stored in those caches is proportional to the number of CPUs that the system uses. As a result, a higher number of CPUs results in a greater amount of kernel memory being held in these caches. Higher amounts of kernel memory in these caches can cause OpenShift Container Platform containers to exceed the configured memory limits, resulting in the container being killed.

To avoid losing containers due to kernel memory issues, ensure that the containers request sufficient memory. You can use the following formula to estimate the amount of memory consumed by the **kmem_cache**, where **nproc** is the number of processing units available that are reported by the **nproc** command. The lower limit of container requests should be this value plus the container memory requirements:

> $(nproc) X 1/2 MiB

## 6.2. USING INIT CONTAINERS TO PERFORM TASKS BEFORE A POD IS DEPLOYED

OpenShift Container Platform provides *init containers*, which are specialized containers that run before application containers and can contain utilities or setup scripts not present in an app image.

### 6.2.1. Understanding Init Containers

You can use an Init Container resource to perform tasks before the rest of a pod is deployed.

A pod can have Init Containers in addition to application containers. Init containers allow you to reorganize setup scripts and binding code.

An Init Container can:

- Contain and run utilities that are not desirable to include in the app Container image for security reasons.

- Contain utilities or custom code for setup that is not present in an app image. For example, there is no requirement to make an image FROM another image just to use a tool like sed, awk, python, or dig during setup.

- Use Linux namespaces so that they have different filesystem views from app containers, such as access to secrets that application containers are not able to access.

Each Init Container must complete successfully before the next one is started. So, Init Containers provide an easy way to block or delay the startup of app containers until some set of preconditions are met.

For example, the following are some ways you can use Init Containers:

- Wait for a service to be created with a shell command like:

  > for i in {1..100}; do sleep 1; if dig myservice; then exit 0; fi; done; exit 1

- Register this pod with a remote server from the downward API with a command like:

  > $ curl -X POST
  > http://$MANAGEMENT_SERVICE_HOST:$MANAGEMENT_SERVICE_PORT/register -d
  > 'instance=$()&ip=$()'

- Wait for some time before starting the app Container with a command like **sleep 60**.

- Clone a git repository into a volume.

- Place values into a configuration file and run a template tool to dynamically generate a configuration file for the main app Container. For example, place the POD_IP value in a configuration and generate the main app configuration file using Jinja.

See the Kubernetes documentation for more information.

### 6.2.2. Creating Init Containers

The following example outlines a simple pod which has two Init Containers. The first waits for **myservice** and the second waits for **mydb**. After both containers complete, the pod begins.

**Procedure**

1. Create a YAML file for the Init Container:

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: myapp-pod
     labels:
       app: myapp
   spec:
     containers:
     - name: myapp-container
       image: registry.access.redhat.com/ubi8/ubi:latest
       command: ['sh', '-c', 'echo The app is running! && sleep 3600']
     initContainers:
     - name: init-myservice
   ```

```
image: registry.access.redhat.com/ubi8/ubi:latest
command: ['sh', '-c', 'until getent hosts myservice; do echo waiting for myservice; sleep 2;
done;']
  - name: init-mydb
    image: registry.access.redhat.com/ubi8/ubi:latest
    command: ['sh', '-c', 'until getent hosts mydb; do echo waiting for mydb; sleep 2; done;']
```

2. Create a YAML file for the **myservice** service.

```
kind: Service
apiVersion: v1
metadata:
  name: myservice
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

3. Create a YAML file for the **mydb** service.

```
kind: Service
apiVersion: v1
metadata:
  name: mydb
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9377
```

4. Run the following command to create the **myapp-pod**:

```
$ oc create -f myapp.yaml
```

**Example output**

```
pod/myapp-pod created
```

5. View the status of the pod:

```
$ oc get pods
```

**Example output**

```
NAME                 READY    STATUS        RESTARTS  AGE
myapp-pod            0/1      Init:0/2       0         5s
```

Note that the pod status indicates it is waiting

6. Run the following commands to create the services:

```
$ oc create -f mydb.yaml
```

```
$ oc create -f myservice.yaml
```

7. View the status of the pod:

```
$ oc get pods
```

**Example output**

```
NAME                    READY    STATUS        RESTARTS  AGE
myapp-pod               1/1      Running       0         2m
```

## 6.3. USING VOLUMES TO PERSIST CONTAINER DATA

Files in a container are ephemeral. As such, when a container crashes or stops, the data is lost. You can use *volumes* to persist the data used by the containers in a pod. A volume is directory, accessible to the Containers in a pod, where data is stored for the life of the pod.

### 6.3.1. Understanding volumes

Volumes are mounted file systems available to pods and their containers which may be backed by a number of host-local or network attached storage endpoints. Containers are not persistent by default; on restart, their contents are cleared.

To ensure that the file system on the volume contains no errors and, if errors are present, to repair them when possible, OpenShift Container Platform invokes the **fsck** utility prior to the **mount** utility. This occurs when either adding a volume or updating an existing volume.

The simplest volume type is **emptyDir**, which is a temporary directory on a single machine. Administrators may also allow you to request a persistent volume that is automatically attached to your pods.

> **NOTE**
>
> **emptyDir** volume storage may be restricted by a quota based on the pod's FSGroup, if the FSGroup parameter is enabled by your cluster administrator.

### 6.3.2. Working with volumes using the OpenShift Container Platform CLI

You can use the CLI command **oc set volume** to add and remove volumes and volume mounts for any object that has a pod template like replication controllers or deployment configs. You can also list volumes in pods or any object that has a pod template.

The **oc set volume** command uses the following general syntax:

```
$ oc set volume <object_selection> <operation> <mandatory_parameters> <options>
```

**Object selection**

Specify one of the following for the **object_selection** parameter in the **oc set volume** command:

**Table 6.1. Object Selection**

| Syntax | Description | Example |
|--------|-------------|---------|
| *<object_type> <name>* | Selects *<name>* of type *<object_type>*. | **deploymentConfig registry** |
| *<object_type>*/*<name>* | Selects *<name>* of type *<object_type>*. | **deploymentConfig/registry** |
| *<object_type>*--selector=*<object_label_selector>* | Selects resources of type *<object_type>* that matched the given label selector. | **deploymentConfig--selector="name=registry"** |
| *<object_type>* --all | Selects all resources of type *<object_type>*. | **deploymentConfig --all** |
| **-f** or **--filename=***<file_name>* | File name, directory, or URL to file to use to edit the resource. | **-f registry-deployment-config.json** |

**Operation**

Specify **--add** or **--remove** for the **operation** parameter in the **oc set volume** command.

**Mandatory parameters**

Any mandatory parameters are specific to the selected operation and are discussed in later sections.

**Options**

Any options are specific to the selected operation and are discussed in later sections.

### 6.3.3. Listing volumes and volume mounts in a pod

You can list volumes and volume mounts in pods or pod templates:

**Procedure**

To list volumes:

```
$ oc set volume <object_type>/<name> [options]
```

List volume supported options:

| Option | Description | Default |
|--------|-------------|---------|
| **--name** | Name of the volume. | |
| **-c, --containers** | Select containers by name. It can also take wildcard **'*'** that matches any character. | **'*'** |

For example:

- To list all volumes for pod **p1**:
    -

```
$ oc set volume pod/p1
```

- To list volume **v1** defined on all deployment configs:

```
$ oc set volume dc --all --name=v1
```

### 6.3.4. Adding volumes to a pod

You can add volumes and volume mounts to a pod.

#### Procedure

To add a volume, a volume mount, or both to pod templates:

```
$ oc set volume <object_type>/<name> --add [options]
```

Table 6.2. Supported Options for Adding Volumes

| Option | Description | Default |
| --- | --- | --- |
| **--name** | Name of the volume. | Automatically generated, if not specified. |
| **-t, --type** | Name of the volume source. Supported values: **emptyDir**, **hostPath**, **secret**, **configmap**, **persistentVolumeClaim** or **projected**. | **emptyDir** |
| **-c, --containers** | Select containers by name. It can also take wildcard **'*'** that matches any character. | **'*'** |
| **-m, --mount-path** | Mount path inside the selected containers. Do not mount to the container root, /, or any path that is the same in the host and the container. This can corrupt your host system if the container is sufficiently privileged, such as the host **/dev/pts** files. It is safe to mount the host by using **/host**. | |

| Option | Description | Default |
|--------|-------------|---------|
| **--path** | Host path. Mandatory parameter for **--type=hostPath**. Do not mount to the container root, /, or any path that is the same in the host and the container. This can corrupt your host system if the container is sufficiently privileged, such as the host **/dev/pts** files. It is safe to mount the host by using **/host**. | |
| **--secret-name** | Name of the secret. Mandatory parameter for **--type=secret**. | |
| **--configmap-name** | Name of the configmap. Mandatory parameter for **--type=configmap**. | |
| **--claim-name** | Name of the persistent volume claim. Mandatory parameter for **--type=persistentVolumeClaim**. | |
| **--source** | Details of volume source as a JSON string. Recommended if the desired volume source is not supported by **--type**. | |
| **-o, --output** | Display the modified objects instead of updating them on the server. Supported values: **json**, **yaml**. | |
| **--output-version** | Output the modified objects with the given version. | **api-version** |

For example:

- To add a new volume source **emptyDir** to the **registry DeploymentConfig** object:

  ```
  $ oc set volume dc/registry --add
  ```

TIP

You can alternatively apply the following YAML to add the volume:

Example 6.1. Sample deployment config with an added volume

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: registry
  namespace: registry
spec:
  replicas: 3
  selector:
    app: httpd
  template:
    metadata:
      labels:
        app: httpd
    spec:
      volumes: ❶
        - name: volume-pppsw
          emptyDir: {}
      containers:
        - name: httpd
          image: >-
            image-registry.openshift-image-registry.svc:5000/openshift/httpd:latest
          ports:
            - containerPort: 8080
              protocol: TCP
```

❶   Add the volume source **emptyDir**.

- To add volume **v1** with secret **secret1** for replication controller **r1** and mount inside the containers at */data*:

```
$ oc set volume rc/r1 --add --name=v1 --type=secret --secret-name='secret1' --mount-path=/data
```

**TIP**

You can alternatively apply the following YAML to add the volume:

**Example 6.2. Sample replication controller with added volume and secret**

```
kind: ReplicationController
apiVersion: v1
metadata:
  name: example-1
  namespace: example
spec:
  replicas: 0
  selector:
    app: httpd
    deployment: example-1
    deploymentconfig: example
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: httpd
        deployment: example-1
        deploymentconfig: example
    spec:
      volumes: 1
      - name: v1
        secret:
          secretName: secret1
          defaultMode: 420
      containers:
      - name: httpd
        image: >-
          image-registry.openshift-image-registry.svc:5000/openshift/httpd:latest
        volumeMounts: 2
        - name: v1
          mountPath: /data
```

**1**    Add the volume and secret.

**2**    Add the container mount path.

- To add existing persistent volume **v1** with claim name **pvc1** to deployment configuration *dc.json* on disk, mount the volume on container **c1** at */data*, and update the **DeploymentConfig** object on the server:

```
$ oc set volume -f dc.json --add --name=v1 --type=persistentVolumeClaim \
  --claim-name=pvc1 --mount-path=/data --containers=c1
```

**TIP**

You can alternatively apply the following YAML to add the volume:

**Example 6.3. Sample deployment config with persistent volume added**

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example
  namespace: example
spec:
  replicas: 3
  selector:
    app: httpd
  template:
    metadata:
      labels:
        app: httpd
    spec:
      volumes:
        - name: volume-pppsw
          emptyDir: {}
        - name: v1                                    1
          persistentVolumeClaim:
            claimName: pvc1
      containers:
        - name: httpd
          image: >-
            image-registry.openshift-image-registry.svc:5000/openshift/httpd:latest
          ports:
            - containerPort: 8080
              protocol: TCP
          volumeMounts:                               2
            - name: v1
              mountPath: /data
```

**1**   Add the persistent volume claim named `pvc1`.

**2**   Add the container mount path.

- To add a volume **v1** based on Git repository **https://github.com/namespace1/project1** with revision **5125c45f9f563** for all replication controllers:

```
$ oc set volume rc --all --add --name=v1 \
  --source='{"gitRepo": {
        "repository": "https://github.com/namespace1/project1",
        "revision": "5125c45f9f563"
      }}'
```

## 6.3.5. Updating volumes and volume mounts in a pod

You can modify the volumes and volume mounts in a pod.

## Procedure

Updating existing volumes using the **--overwrite** option:

```
$ oc set volume <object_type>/<name> --add --overwrite [options]
```

For example:

- To replace existing volume **v1** for replication controller **r1** with existing persistent volume claim **pvc1**:

  ```
  $ oc set volume rc/r1 --add --overwrite --name=v1 --type=persistentVolumeClaim --claim-name=pvc1
  ```

TIP

You can alternatively apply the following YAML to replace the volume:

**Example 6.4. Sample replication controller with persistent volume claim named pvc1**

```
kind: ReplicationController
apiVersion: v1
metadata:
  name: example-1
  namespace: example
spec:
  replicas: 0
  selector:
    app: httpd
    deployment: example-1
    deploymentconfig: example
  template:
    metadata:
      labels:
        app: httpd
        deployment: example-1
        deploymentconfig: example
    spec:
      volumes:
        - name: v1     1
          persistentVolumeClaim:
            claimName: pvc1
      containers:
        - name: httpd
          image: >-
            image-registry.openshift-image-registry.svc:5000/openshift/httpd:latest
          ports:
            - containerPort: 8080
              protocol: TCP
          volumeMounts:
            - name: v1
              mountPath: /data
```

**1**  Set persistent volume claim to **pvc1**.

- To change the **DeploymentConfig** object **d1** mount point to */opt* for volume **v1**:

```
$ oc set volume dc/d1 --add --overwrite --name=v1 --mount-path=/opt
```

**TIP**

You can alternatively apply the following YAML to change the mount point:

**Example 6.5. Sample deployment config with mount point set to opt.**

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example
  namespace: example
spec:
  replicas: 3
  selector:
    app: httpd
  template:
    metadata:
      labels:
        app: httpd
    spec:
      volumes:
        - name: volume-pppsw
          emptyDir: {}
        - name: v2
          persistentVolumeClaim:
            claimName: pvc1
        - name: v1
          persistentVolumeClaim:
            claimName: pvc1
      containers:
        - name: httpd
          image: >-
            image-registry.openshift-image-registry.svc:5000/openshift/httpd:latest
          ports:
            - containerPort: 8080
              protocol: TCP
          volumeMounts: ❶
            - name: v1
              mountPath: /opt
```

❶ Set the mount point to **/opt**.

## 6.3.6. Removing volumes and volume mounts from a pod

You can remove a volume or volume mount from a pod.

**Procedure**

To remove a volume from pod templates:

```
$ oc set volume <object_type>/<name> --remove [options]
```

**Table 6.3. Supported options for removing volumes**

| Option | Description | Default |
|--------|-------------|---------|
| **--name** | Name of the volume. | |
| **-c, --containers** | Select containers by name. It can also take wildcard **'*'** that matches any character. | **'*'** |
| **--confirm** | Indicate that you want to remove multiple volumes at once. | |
| **-o, --output** | Display the modified objects instead of updating them on the server. Supported values: **json**, **yaml**. | |
| **--output-version** | Output the modified objects with the given version. | **api-version** |

For example:

- To remove a volume **v1** from the **DeploymentConfig** object **d1**:

  ```
  $ oc set volume dc/d1 --remove --name=v1
  ```

- To unmount volume **v1** from container **c1** for the **DeploymentConfig** object **d1** and remove the volume **v1** if it is not referenced by any containers on **d1**:

  ```
  $ oc set volume dc/d1 --remove --name=v1 --containers=c1
  ```

- To remove all volumes for replication controller **r1**:

  ```
  $ oc set volume rc/r1 --remove --confirm
  ```

## 6.3.7. Configuring volumes for multiple uses in a pod

You can configure a volume to allows you to share one volume for multiple uses in a single pod using the **volumeMounts.subPath** property to specify a **subPath** value inside a volume instead of the volume's root.

### Procedure

1. View the list of files in the volume, run the **oc rsh** command:

   ```
   $ oc rsh <pod>
   ```

   **Example output**

   ```
   sh-4.2$ ls /path/to/volume/subpath/mount
   example_file1 example_file2 example_file3
   ```

2. Specify the **subPath**:

**Example Pod spec with subPath parameter**

```
apiVersion: v1
kind: Pod
metadata:
  name: my-site
spec:
    containers:
    - name: mysql
      image: mysql
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: site-data
        subPath: mysql 1
    - name: php
      image: php
      volumeMounts:
      - mountPath: /var/www/html
        name: site-data
        subPath: html 2
    volumes:
    - name: site-data
      persistentVolumeClaim:
        claimName: my-site-data
```

**1**    Databases are stored in the **mysql** folder.

**2**    HTML content is stored in the **html** folder.

## 6.4. MAPPING VOLUMES USING PROJECTED VOLUMES

A *projected volume* maps several existing volume sources into the same directory.

The following types of volume sources can be projected:

- Secrets

- Config Maps

- Downward API

> **NOTE**
>
> All sources are required to be in the same namespace as the pod.

### 6.4.1. Understanding projected volumes

Projected volumes can map any combination of these volume sources into a single directory, allowing the user to:

- automatically populate a single volume with the keys from multiple secrets, config maps, and with downward API information, so that I can synthesize a single directory with various sources of information;

- populate a single volume with the keys from multiple secrets, config maps, and with downward API information, explicitly specifying paths for each item, so that I can have full control over the contents of that volume.

> **IMPORTANT**
>
> When the **RunAsUser** permission is set in the security context of a Linux-based pod, the projected files have the correct permissions set, including container user ownership. However, when the Windows equivalent **RunAsUsername** permission is set in a Windows pod, the kubelet is unable to correctly set ownership on the files in the projected volume.
>
> Therefore, the **RunAsUsername** permission set in the security context of a Windows pod is not honored for Windows projected volumes running in OpenShift Container Platform.

The following general scenarios show how you can use projected volumes.

**Config map, secrets, Downward API.**

Projected volumes allow you to deploy containers with configuration data that includes passwords. An application using these resources could be deploying Red Hat OpenStack Platform (RHOSP) on Kubernetes. The configuration data might have to be assembled differently depending on if the services are going to be used for production or for testing. If a pod is labeled with production or testing, the downward API selector **metadata.labels** can be used to produce the correct RHOSP configs.

**Config map + secrets.**

Projected volumes allow you to deploy containers involving configuration data and passwords. For example, you might execute a config map with some sensitive encrypted tasks that are decrypted using a vault password file.

**ConfigMap + Downward API.**

Projected volumes allow you to generate a config including the pod name (available via the **metadata.name** selector). This application can then pass the pod name along with requests to easily determine the source without using IP tracking.

**Secrets + Downward API.**

Projected volumes allow you to use a secret as a public key to encrypt the namespace of the pod (available via the **metadata.namespace** selector). This example allows the Operator to use the application to deliver the namespace information securely without using an encrypted transport.

### 6.4.1.1. Example Pod specs

The following are examples of **Pod** specs for creating projected volumes.

### Pod with a secret, a Downward API, and a config map

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
```

```
  image: busybox
  volumeMounts: 1
  - name: all-in-one
    mountPath: "/projected-volume" 2
    readOnly: true 3
volumes: 4
- name: all-in-one 5
  projected:
    defaultMode: 0400 6
    sources:
    - secret:
        name: mysecret 7
        items:
          - key: username
            path: my-group/my-username 8
    - downwardAPI: 9
        items:
          - path: "labels"
            fieldRef:
              fieldPath: metadata.labels
          - path: "cpu_limit"
            resourceFieldRef:
              containerName: container-test
              resource: limits.cpu
    - configMap: 10
        name: myconfigmap
        items:
          - key: config
            path: my-group/my-config
            mode: 0777 11
```

| | |
|---|---|
| **1** | Add a **volumeMounts** section for each container that needs the secret. |
| **2** | Specify a path to an unused directory where the secret will appear. |
| **3** | Set **readOnly** to **true**. |
| **4** | Add a **volumes** block to list each projected volume source. |
| **5** | Specify any name for the volume. |
| **6** | Set the execute permission on the files. |
| **7** | Add a secret. Enter the name of the secret object. Each secret you want to use must be listed. |
| **8** | Specify the path to the secrets file under the **mountPath**. Here, the secrets file is in */projected-volume/my-group/my-username*. |
| **9** | Add a Downward API source. |
| **10** | Add a ConfigMap source. |
| **11** | Set the mode for the specific projection |

> **NOTE**
>
> If there are multiple containers in the pod, each container needs a **volumeMounts** section, but only one **volumes** section is needed.

## Pod with multiple secrets with a non-default permission mode set

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      defaultMode: 0755
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/my-username
      - secret:
          name: mysecret2
          items:
            - key: password
              path: my-group/my-password
              mode: 511
```

> **NOTE**
>
> The **defaultMode** can only be specified at the projected level and not for each volume source. However, as illustrated above, you can explicitly set the **mode** for each individual projection.

### 6.4.1.2. Pathing Considerations

#### Collisions Between Keys when Configured Paths are Identical

If you configure any keys with the same path, the pod spec will not be accepted as valid. In the following example, the specified path for **mysecret** and **myconfigmap** are the same:
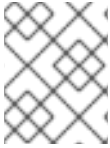
```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
```

```
    - name: container-test
      image: busybox
      volumeMounts:
      - name: all-in-one
        mountPath: "/projected-volume"
        readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/data
      - configMap:
          name: myconfigmap
          items:
            - key: config
              path: my-group/data
```

Consider the following situations related to the volume file paths.

**Collisions Between Keys without Configured Paths**

The only run–time validation that can occur is when all the paths are known at pod creation, similar to the above scenario. Otherwise, when a conflict occurs the most recent specified resource will overwrite anything preceding it (this is true for resources that are updated after pod creation as well).

**Collisions when One Path is Explicit and the Other is Automatically Projected**

In the event that there is a collision due to a user specified path matching data that is automatically projected, the latter resource will overwrite anything preceding it as before

## 6.4.2. Configuring a Projected Volume for a Pod

When creating projected volumes, consider the volume file path situations described in *Understanding projected volumes.*

The following example shows how to use a projected volume to mount an existing secret volume source. The steps can be used to create a user name and password secrets from local files. You then create a pod that runs one container, using a projected volume to mount the secrets into the same shared directory.

### Procedure

To use a projected volume to mount an existing secret volume source.

1. Create files containing the secrets, entering the following, replacing the password and user information as appropriate:

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: mysecret
   type: Opaque
   ```

```
data:
  pass: MWYyZDFlMmU2N2Rm
  user: YWRtaW4=
```

The **user** and **pass** values can be any valid string that is  **base64** encoded.

The following example shows **admin** in base64:

```
$ echo -n "admin" | base64
```

### Example output

```
YWRtaW4=
```

The following example shows the password **1f2d1e2e67df** in base64:.

```
$ echo -n "1f2d1e2e67df" | base64
```

### Example output

```
MWYyZDFlMmU2N2Rm
```

2. Use the following command to create the secrets:

```
$ oc create -f <secrets-filename>
```

For example:

```
$ oc create -f secret.yaml
```

### Example output

```
secret "mysecret" created
```

3. You can check that the secret was created using the following commands:

```
$ oc get secret <secret-name>
```

For example:

```
$ oc get secret mysecret
```

### Example output

```
NAME       TYPE     DATA    AGE
mysecret   Opaque   2       17h
```

```
$ oc get secret <secret-name> -o yaml
```

For example:

```
$ oc get secret mysecret -o yaml
```

```
apiVersion: v1
data:
  pass: MWYyZDFlMmU2N2Rm
  user: YWRtaW4=
kind: Secret
metadata:
  creationTimestamp: 2017-05-30T20:21:38Z
  name: mysecret
  namespace: default
  resourceVersion: "2107"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: 959e0424-4575-11e7-9f97-fa163e4bd54c
type: Opaque
```

4. Create a pod configuration file similar to the following that includes a **volumes** section:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-projected-volume
spec:
  containers:
  - name: test-projected-volume
    image: busybox
    args:
    - sleep
    - "86400"
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:          ❶
          name: user
      - secret:          ❷
          name: pass
```

❶ ❷ The name of the secret you created.

5. Create the pod from the configuration file:

```
$ oc create -f <your_yaml_file>.yaml
```

For example:

```
$ oc create -f secret-pod.yaml
```

**Example output**

> pod "test-projected-volume" created

6. Verify that the pod container is running, and then watch for changes to the pod:

> $ oc get pod <name>

For example:

> $ oc get pod test-projected-volume

The output should appear similar to the following:

**Example output**

```
NAME                  READY    STATUS   RESTARTS  AGE
test-projected-volume  1/1     Running  0         14s
```

7. In another terminal, use the **oc exec** command to open a shell to the running container:

> $ oc exec -it <pod> <command>

For example:

> $ oc exec -it test-projected-volume -- /bin/sh

8. In your shell, verify that the **projected-volumes** directory contains your projected sources:

> / # ls

**Example output**

```
bin        home          root        tmp
dev         proc          run         usr
etc        projected-volume  sys           var
```

# 6.5. ALLOWING CONTAINERS TO CONSUME API OBJECTS

The *Downward API* is a mechanism that allows containers to consume information about API objects without coupling to OpenShift Container Platform. Such information includes the pod's name, namespace, and resource values. Containers can consume information from the downward API using environment variables or a volume plugin.

## 6.5.1. Expose pod information to Containers using the Downward API

The Downward API contains such information as the pod's name, project, and resource values. Containers can consume information from the downward API using environment variables or a volume plugin.

Fields within the pod are selected using the **FieldRef** API type. **FieldRef** has two fields:

| Field | Description |
|-------|-------------|
| **fieldPath** | The path of the field to select, relative to the pod. |
| **apiVersion** | The API version to interpret the **fieldPath** selector within. |

Currently, the valid selectors in the v1 API include:

| Selector | Description |
|----------|-------------|
| **metadata.name** | The pod's name. This is supported in both environment variables and volumes. |
| **metadata.namespace** | The pod's namespace.This is supported in both environment variables and volumes. |
| **metadata.labels** | The pod's labels. This is only supported in volumes and not in environment variables. |
| **metadata.annotations** | The pod's annotations. This is only supported in volumes and not in environment variables. |
| **status.podIP** | The pod's IP. This is only supported in environment variables and not volumes. |

The **apiVersion** field, if not specified, defaults to the API version of the enclosing pod template.

## 6.5.2. Understanding how to consume container values using the downward API

You containers can consume API values using environment variables or a volume plugin. Depending on the method you choose, containers can consume:

- Pod name
- Pod project/namespace
- Pod annotations
- Pod labels

Annotations and labels are available using only a volume plugin.

### 6.5.2.1. Consuming container values using environment variables

When using a container's environment variables, use the **EnvVar** type's **valueFrom** field (of type **EnvVarSource**) to specify that the variable's value should come from a **FieldRef** source instead of the literal value specified by the **value** field.

Only constant attributes of the pod can be consumed this way, as environment variables cannot be updated once a process is started in a way that allows the process to be notified that the value of a variable has changed. The fields supported using environment variables are:

- Pod name

- Pod project/namespace

## Procedure

To use environment variables

1. Create a **pod.yaml** file:

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: dapi-env-test-pod
   spec:
     containers:
       - name: env-test-container
         image: gcr.io/google_containers/busybox
         command: [ "/bin/sh", "-c", "env" ]
         env:
           - name: MY_POD_NAME
             valueFrom:
               fieldRef:
                 fieldPath: metadata.name
           - name: MY_POD_NAMESPACE
             valueFrom:
               fieldRef:
                 fieldPath: metadata.namespace
     restartPolicy: Never
   ```

2. Create the pod from the **pod.yaml** file:

   ```
   $ oc create -f pod.yaml
   ```

3. Check the container's logs for the **MY_POD_NAME** and **MY_POD_NAMESPACE** values:

   ```
   $ oc logs -p dapi-env-test-pod
   ```

### 6.5.2.2. Consuming container values using a volume plugin

You containers can consume API values using a volume plugin.

Containers can consume:

- Pod name

- Pod project/namespace

- Pod annotations

- Pod labels

## Procedure

To use the volume plugin:

1. Create a **volume-pod.yaml** file:

```
kind: Pod
apiVersion: v1
metadata:
 labels:
   zone: us-east-coast
   cluster: downward-api-test-cluster1
   rack: rack-123
 name: dapi-volume-test-pod
 annotations:
   annotation1: "345"
   annotation2: "456"
spec:
 containers:
   - name: volume-test-container
     image: gcr.io/google_containers/busybox
     command: ["sh", "-c", "cat /tmp/etc/pod_labels /tmp/etc/pod_annotations"]
     volumeMounts:
       - name: podinfo
         mountPath: /tmp/etc
         readOnly: false
 volumes:
 - name: podinfo
   downwardAPI:
     defaultMode: 420
     items:
     - fieldRef:
         fieldPath: metadata.name
       path: pod_name
     - fieldRef:
         fieldPath: metadata.namespace
       path: pod_namespace
     - fieldRef:
         fieldPath: metadata.labels
       path: pod_labels
     - fieldRef:
         fieldPath: metadata.annotations
       path: pod_annotations
 restartPolicy: Never
```

2. Create the pod from the **volume-pod.yaml** file:

```
$ oc create -f volume-pod.yaml
```

3. Check the container's logs and verify the presence of the configured fields:

```
$ oc logs -p dapi-volume-test-pod
```

**Example output**

```
cluster=downward-api-test-cluster1
rack=rack-123
zone=us-east-coast
annotation1=345
annotation2=456
kubernetes.io/config.source=api
```

### 6.5.3. Understanding how to consume container resources using the Downward API

When creating pods, you can use the Downward API to inject information about computing resource requests and limits so that image and application authors can correctly create an image for specific environments.

You can do this using environment variable or a volume plugin.

#### 6.5.3.1. Consuming container resources using environment variables

When creating pods, you can use the Downward API to inject information about computing resource requests and limits using environment variables.

#### Procedure

To use environment variables:

1. When creating a pod configuration, specify environment variables that correspond to the contents of the **resources** field in the **spec.container** field:

```
....
spec:
 containers:
   - name: test-container
    image: gcr.io/google_containers/busybox:1.24
    command: [ "/bin/sh", "-c", "env" ]
    resources:
      requests:
        memory: "32Mi"
        cpu: "125m"
      limits:
        memory: "64Mi"
        cpu: "250m"
    env:
      - name: MY_CPU_REQUEST
        valueFrom:
          resourceFieldRef:
            resource: requests.cpu
      - name: MY_CPU_LIMIT
        valueFrom:
          resourceFieldRef:
            resource: limits.cpu
      - name: MY_MEM_REQUEST
        valueFrom:
          resourceFieldRef:
            resource: requests.memory
      - name: MY_MEM_LIMIT
        valueFrom:
```

```
      resourceFieldRef:
        resource: limits.memory
....
```

If the resource limits are not included in the container configuration, the downward API defaults to the node's CPU and memory allocatable values.

2. Create the pod from the ***pod.yaml*** file:

```
$ oc create -f pod.yaml
```

### 6.5.3.2. Consuming container resources using a volume plugin

When creating pods, you can use the Downward API to inject information about computing resource requests and limits using a volume plugin.

### Procedure

To use the Volume Plugin:

1. When creating a pod configuration, use the **spec.volumes.downwardAPI.items** field to describe the desired resources that correspond to the **spec.resources** field:

```
....
spec:
 containers:
  - name: client-container
    image: gcr.io/google_containers/busybox:1.24
    command: ["sh", "-c", "while true; do echo; if [[ -e /etc/cpu_limit ]]; then cat /etc/cpu_limit;
fi; if [[ -e /etc/cpu_request ]]; then cat /etc/cpu_request; fi; if [[ -e /etc/mem_limit ]]; then cat
/etc/mem_limit; fi; if [[ -e /etc/mem_request ]]; then cat /etc/mem_request; fi; sleep 5; done"]
    resources:
      requests:
        memory: "32Mi"
        cpu: "125m"
      limits:
        memory: "64Mi"
        cpu: "250m"
    volumeMounts:
      - name: podinfo
        mountPath: /etc
        readOnly: false
 volumes:
  - name: podinfo
    downwardAPI:
      items:
        - path: "cpu_limit"
          resourceFieldRef:
            containerName: client-container
            resource: limits.cpu
        - path: "cpu_request"
          resourceFieldRef:
            containerName: client-container
            resource: requests.cpu
        - path: "mem_limit"
          resourceFieldRef:
```

```
          containerName: client-container
          resource: limits.memory
        - path: "mem_request"
          resourceFieldRef:
            containerName: client-container
            resource: requests.memory
    ....
```

If the resource limits are not included in the container configuration, the Downward API defaults to the node's CPU and memory allocatable values.

2. Create the pod from the ***volume-pod.yaml*** file:

```
$ oc create -f volume-pod.yaml
```

## 6.5.4. Consuming secrets using the Downward API

When creating pods, you can use the downward API to inject secrets so image and application authors can create an image for specific environments.

**Procedure**

1. Create a **secret.yaml** file:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
data:
  password: cGFzc3dvcmQ=
  username: ZGV2ZWxvcGVy
type: kubernetes.io/basic-auth
```

2. Create a **Secret** object from the **secret.yaml** file:

```
$ oc create -f secret.yaml
```

3. Create a **pod.yaml** file that references the **username** field from the above **Secret** object:

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_SECRET_USERNAME
          valueFrom:
            secretKeyRef:
```

```
        name: mysecret
        key: username
  restartPolicy: Never
```

4. Create the pod from the **pod.yaml** file:

```
$ oc create -f pod.yaml
```

5. Check the container's logs for the **MY_SECRET_USERNAME** value:

```
$ oc logs -p dapi-env-test-pod
```

### 6.5.5. Consuming configuration maps using the Downward API

When creating pods, you can use the Downward API to inject configuration map values so image and application authors can create an image for specific environments.

**Procedure**

1. Create a *configmap.yaml* file:

   ```
   apiVersion: v1
   kind: ConfigMap
   metadata:
     name: myconfigmap
   data:
     mykey: myvalue
   ```

2. Create a **ConfigMap** object from the *configmap.yaml* file:

   ```
   $ oc create -f configmap.yaml
   ```

3. Create a *pod.yaml* file that references the above **ConfigMap** object:

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: dapi-env-test-pod
   spec:
     containers:
       - name: env-test-container
         image: gcr.io/google_containers/busybox
         command: [ "/bin/sh", "-c", "env" ]
         env:
           - name: MY_CONFIGMAP_VALUE
             valueFrom:
               configMapKeyRef:
                 name: myconfigmap
                 key: mykey
     restartPolicy: Always
   ```

4. Create the pod from the *pod.yaml* file:

```
$ oc create -f pod.yaml
```

5. Check the container's logs for the **MY_CONFIGMAP_VALUE** value:

```
$ oc logs -p dapi-env-test-pod
```

## 6.5.6. Referencing environment variables

When creating pods, you can reference the value of a previously defined environment variable by using the **$()** syntax. If the environment variable reference can not be resolved, the value will be left as the provided string.

**Procedure**

1. Create a *pod.yaml* file that references an existing **environment variable**:

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
 containers:
   - name: env-test-container
     image: gcr.io/google_containers/busybox
     command: [ "/bin/sh", "-c", "env" ]
     env:
       - name: MY_EXISTING_ENV
         value: my_value
       - name: MY_ENV_VAR_REF_ENV
         value: $(MY_EXISTING_ENV)
 restartPolicy: Never
```

2. Create the pod from the *pod.yaml* file:

```
$ oc create -f pod.yaml
```

3. Check the container's logs for the **MY_ENV_VAR_REF_ENV** value:

```
$ oc logs -p dapi-env-test-pod
```

## 6.5.7. Escaping environment variable references

When creating a pod, you can escape an environment variable reference by using a double dollar sign. The value will then be set to a single dollar sign version of the provided value.

**Procedure**

1. Create a *pod.yaml* file that references an existing **environment variable**:

```
apiVersion: v1
kind: Pod
metadata:
```

```
    name: dapi-env-test-pod
  spec:
    containers:
      - name: env-test-container
        image: gcr.io/google_containers/busybox
        command: [ "/bin/sh", "-c", "env" ]
        env:
          - name: MY_NEW_ENV
            value: $$(SOME_OTHER_ENV)
    restartPolicy: Never
```

2. Create the pod from the ***pod.yaml*** file:

   ```
   $ oc create -f pod.yaml
   ```

3. Check the container's logs for the **MY_NEW_ENV** value:

   ```
   $ oc logs -p dapi-env-test-pod
   ```

# 6.6. COPYING FILES TO OR FROM AN OPENSHIFT CONTAINER PLATFORM CONTAINER

You can use the CLI to copy local files to or from a remote directory in a container using the **rsync** command.

## 6.6.1. Understanding how to copy files

The **oc rsync** command, or remote sync, is a useful tool for copying database archives to and from your pods for backup and restore purposes. You can also use **oc rsync** to copy source code changes into a running pod for development debugging, when the running pod supports hot reload of source files.

```
$ oc rsync <source> <destination> [-c <container>]
```

### 6.6.1.1. Requirements

**Specifying the Copy Source**

The source argument of the **oc rsync** command must point to either a local directory or a pod directory. Individual files are not supported.

When specifying a pod directory the directory name must be prefixed with the pod name:

```
<pod name>:<dir>
```

If the directory name ends in a path separator (/), only the contents of the directory are copied to the destination. Otherwise, the directory and its contents are copied to the destination.

**Specifying the Copy Destination**

The destination argument of the **oc rsync** command must point to a directory. If the directory does not exist, but **rsync** is used for copy, the directory is created for you.

**Deleting Files at the Destination**

The **--delete** flag may be used to delete any files in the remote directory that are not in the local directory.

**Continuous Syncing on File Change**

Using the **--watch** option causes the command to monitor the source path for any file system changes, and synchronizes changes when they occur. With this argument, the command runs forever.

Synchronization occurs after short quiet periods to ensure a rapidly changing file system does not result in continuous synchronization calls.

When using the **--watch** option, the behavior is effectively the same as manually invoking **oc rsync** repeatedly, including any arguments normally passed to **oc rsync**. Therefore, you can control the behavior via the same flags used with manual invocations of **oc rsync**, such as **--delete**.

## 6.6.2. Copying files to and from containers

Support for copying local files to or from a container is built into the CLI.

### Prerequisites

When working with **oc rsync**, note the following:

**rsync must be installed**

The **oc rsync** command uses the local **rsync** tool if present on the client machine and the remote container.

If **rsync** is not found locally or in the remote container, a **tar** archive is created locally and sent to the container where the **tar** utility is used to extract the files. If **tar** is not available in the remote container, the copy will fail.

The **tar** copy method does not provide the same functionality as **oc rsync**. For example, **oc rsync** creates the destination directory if it does not exist and only sends files that are different between the source and the destination.
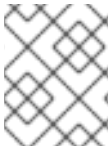
> **NOTE**
>
> In Windows, the **cwRsync** client should be installed and added to the PATH for use with the **oc rsync** command.

### Procedure

- To copy a local directory to a pod directory:

  ```
  $ oc rsync <local-dir> <pod-name>:/<remote-dir> -c <container-name>
  ```

  For example:

  ```
  $ oc rsync /home/user/source devpod1234:/src -c user-container
  ```

- To copy a pod directory to a local directory:

  ```
  $ oc rsync devpod1234:/src /home/user/source
  ```

  **Example output**

```
$ oc rsync devpod1234:/src/status.txt /home/user/
```

## 6.6.3. Using advanced Rsync features

The **oc rsync** command exposes fewer command line options than standard **rsync**. In the case that you want to use a standard **rsync** command line option that is not available in **oc rsync**, for example the **--exclude-from=FILE** option, it might be possible to use standard **rsync** 's **--rsh** (**-e**) option or **RSYNC_RSH** environment variable as a workaround, as follows:

```
$ rsync --rsh='oc rsh' --exclude-from=FILE SRC POD:DEST
```

or:

Export the **RSYNC_RSH** variable:

```
$ export RSYNC_RSH='oc rsh'
```

Then, run the rsync command:

```
$ rsync --exclude-from=FILE SRC POD:DEST
```

Both of the above examples configure standard **rsync** to use **oc rsh** as its remote shell program to enable it to connect to the remote pod, and are an alternative to running **oc rsync**.

# 6.7. EXECUTING REMOTE COMMANDS IN AN OPENSHIFT CONTAINER PLATFORM CONTAINER

You can use the CLI to execute remote commands in an OpenShift Container Platform container.

## 6.7.1. Executing remote commands in containers

Support for remote container command execution is built into the CLI.

### Procedure

To run a command in a container:

```
$ oc exec <pod> [-c <container>] <command> [<arg_1> ... <arg_n>]
```

For example:

```
$ oc exec mypod date
```

### Example output

```
Thu Apr  9 02:21:53 UTC 2015
```

> **IMPORTANT**
>
> For security purposes, the **oc exec** command does not work when accessing privileged containers except when the command is executed by a **cluster-admin** user.

## 6.7.2. Protocol for initiating a remote command from a client

Clients initiate the execution of a remote command in a container by issuing a request to the Kubernetes API server:

```
/proxy/nodes/<node_name>/exec/<namespace>/<pod>/<container>?command=<command>
```

In the above URL:

- **<node_name>** is the FQDN of the node.

- **<namespace>** is the project of the target pod.

- **<pod>** is the name of the target pod.

- **<container>** is the name of the target container.

- **<command>** is the desired command to be executed.

For example:

```
/proxy/nodes/node123.openshift.com/exec/myns/mypod/mycontainer?command=date
```

Additionally, the client can add parameters to the request to indicate if:

- the client should send input to the remote container's command (stdin).

- the client's terminal is a TTY.

- the remote container's command should send output from stdout to the client.

- the remote container's command should send output from stderr to the client.

After sending an **exec** request to the API server, the client upgrades the connection to one that supports multiplexed streams; the current implementation uses **HTTP/2**.

The client creates one stream each for stdin, stdout, and stderr. To distinguish among the streams, the client sets the **streamType** header on the stream to one of  **stdin**, **stdout**, or **stderr**.

The client closes all streams, the upgraded connection, and the underlying connection when it is finished with the remote command execution request.

## 6.8. USING PORT FORWARDING TO ACCESS APPLICATIONS IN A CONTAINER

OpenShift Container Platform supports port forwarding to pods.

### 6.8.1. Understanding port forwarding

You can use the CLI to forward one or more local ports to a pod. This allows you to listen on a given or random port locally, and have data forwarded to and from given ports in the pod.

Support for port forwarding is built into the CLI:

```
$ oc port-forward <pod> [<local_port>:]<remote_port> [...[<local_port_n>:]<remote_port_n>]
```

The CLI listens on each local port specified by the user, forwarding using the protocol described below.

Ports may be specified using the following formats:

| | |
|---|---|
| **5000** | The client listens on port 5000 locally and forwards to 5000 in the pod. |
| **6000:5000** | The client listens on port 6000 locally and forwards to 5000 in the pod. |
| **:5000** or **0:5000** | The client selects a free local port and forwards to 5000 in the pod. |

OpenShift Container Platform handles port-forward requests from clients. Upon receiving a request, OpenShift Container Platform upgrades the response and waits for the client to create port-forwarding streams. When OpenShift Container Platform receives a new stream, it copies data between the stream and the pod's port.

Architecturally, there are options for forwarding to a pod's port. The supported OpenShift Container Platform implementation invokes **nsenter** directly on the node host to enter the pod's network namespace, then invokes **socat** to copy data between the stream and the pod's port. However, a custom implementation could include running a *helper* pod that then runs **nsenter** and **socat**, so that those binaries are not required to be installed on the host.

## 6.8.2. Using port forwarding

You can use the CLI to port-forward one or more local ports to a pod.

### Procedure

Use the following command to listen on the specified port in a pod:

```
$ oc port-forward <pod> [<local_port>:]<remote_port> [...[<local_port_n>:]<remote_port_n>]
```

For example:

- Use the following command to listen on ports **5000** and **6000** locally and forward data to and from ports **5000** and **6000** in the pod:

  ```
  $ oc port-forward <pod> 5000 6000
  ```

  **Example output**

  ```
  Forwarding from 127.0.0.1:5000 -> 5000
  Forwarding from [::1]:5000 -> 5000
  Forwarding from 127.0.0.1:6000 -> 6000
  Forwarding from [::1]:6000 -> 6000
  ```

- Use the following command to listen on port **8888** locally and forward to **5000** in the pod:

  ```
  $ oc port-forward <pod> 8888:5000
  ```

  **Example output**

> Forwarding from 127.0.0.1:8888 -> 5000
> Forwarding from [::1]:8888 -> 5000

- Use the following command to listen on a free port locally and forward to **5000** in the pod:

> $ oc port-forward <pod> :5000

**Example output**

> Forwarding from 127.0.0.1:42390 -> 5000
> Forwarding from [::1]:42390 -> 5000

Or:

> $ oc port-forward <pod> 0:5000

### 6.8.3. Protocol for initiating port forwarding from a client

Clients initiate port forwarding to a pod by issuing a request to the Kubernetes API server:

> /proxy/nodes/<node_name>/portForward/<namespace>/<pod>

In the above URL:

- **<node_name>** is the FQDN of the node.

- **<namespace>** is the namespace of the target pod.

- **<pod>** is the name of the target pod.

For example:

> /proxy/nodes/node123.openshift.com/portForward/myns/mypod

After sending a port forward request to the API server, the client upgrades the connection to one that supports multiplexed streams; the current implementation uses **Hyptertext Transfer Protocol Version 2 (HTTP/2)**.

The client creates a stream with the **port** header containing the target port in the pod. All data written to the stream is delivered via the kubelet to the target pod and port. Similarly, all data sent from the pod for that forwarded connection is delivered back to the same stream in the client.

The client closes all streams, the upgraded connection, and the underlying connection when it is finished with the port forwarding request.

## 6.9. USING SYSCTLS IN CONTAINERS

Sysctl settings are exposed via Kubernetes, allowing users to modify certain kernel parameters at runtime for namespaces within a container. Only sysctls that are namespaced can be set independently on pods. If a sysctl is not namespaced, called *node-level*, you must use another method of setting the sysctl, such as the Node Tuning Operator. Moreover, only those sysctls considered *safe* are whitelisted by default; you can manually enable other *unsafe* sysctls on the node to be available to the user.

## 6.9.1. About sysctls

In Linux, the sysctl interface allows an administrator to modify kernel parameters at runtime. Parameters are available via the */proc/sys/* virtual process file system. The parameters cover various subsystems, such as:

- kernel (common prefix: *kernel.*)

- networking (common prefix: *net.*)

- virtual memory (common prefix: *vm.*)

- MDADM (common prefix: *dev.*)

More subsystems are described in [Kernel documentation](). To get a list of all parameters, run:

```
$ sudo sysctl -a
```

### 6.9.1.1. Namespaced versus node-level sysctls

A number of sysctls are *namespaced* in the Linux kernels. This means that you can set them independently for each pod on a node. Being namespaced is a requirement for sysctls to be accessible in a pod context within Kubernetes.

The following sysctls are known to be namespaced:

- *kernel.shm\**

- *kernel.msg\**

- *kernel.sem*

- *fs.mqueue.\**

Additionally, most of the sysctls in the **net.*** group are known to be namespaced. Their namespace adoption differs based on the kernel version and distributor.

Sysctls that are not namespaced are called *node-level* and must be set manually by the cluster administrator, either by means of the underlying Linux distribution of the nodes, such as by modifying the */etc/sysctls.conf* file, or by using a daemon set with privileged containers. You can use the Node Tuning Operator to set *node-level* sysctls.

> **NOTE**
>
> Consider marking nodes with special sysctls as tainted. Only schedule pods onto them that need those sysctl settings. Use the taints and toleration feature to mark the nodes.

### 6.9.1.2. Safe versus unsafe sysctls

Sysctls are grouped into *safe* and *unsafe* sysctls.

For a sysctl to be considered safe, it must use proper namespacing and must be properly isolated between pods on the same node. This means that if you set a sysctl for one pod it must not:

- Influence any other pod on the node

- Harm the node's health

- Gain CPU or memory resources outside of the resource limits of a pod

OpenShift Container Platform supports, or whitelists, the following sysctls in the safe set:

- *kernel.shm_rmid_forced*

- *net.ipv4.ip_local_port_range*

- *net.ipv4.tcp_syncookies*

- *net.ipv4.ping_group_range*

All safe sysctls are enabled by default. You can use a sysctl in a pod by modifying the **Pod** spec.

Any sysctl not whitelisted by OpenShift Container Platform is considered unsafe for OpenShift Container Platform. Note that being namespaced alone is not sufficient for the sysctl to be considered safe.

All unsafe sysctls are disabled by default, and the cluster administrator must manually enable them on a per-node basis. Pods with disabled unsafe sysctls are scheduled but do not launch.

```
$ oc get pod
```

**Example output**

```
NAME       READY  STATUS          RESTARTS  AGE
hello-pod  0/1    SysctlForbidden  0          14s
```

## 6.9.2. Setting sysctls for a pod

You can set sysctls on pods using the pod's **securityContext**. The **securityContext** applies to all containers in the same pod.

Safe sysctls are allowed by default. A pod with unsafe sysctls fails to launch on any node unless the cluster administrator explicitly enables unsafe sysctls for that node. As with node-level sysctls, use the taints and toleration feature or labels on nodes to schedule those pods onto the right nodes.

The following example uses the pod **securityContext** to set a safe sysctl **kernel.shm_rmid_forced** and two unsafe sysctls, **net.core.somaxconn** and **kernel.msgmax**. There is no distinction between *safe* and *unsafe* sysctls in the specification.

> **WARNING**
>
> To avoid destabilizing your operating system, modify sysctl parameters only after you understand their effects.

**Procedure**

To use safe and unsafe sysctls:

1. Modify the YAML file that defines the pod and add the **securityContext** spec, as shown in the following example:

```
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example
spec:
  securityContext:
    sysctls:
    - name: kernel.shm_rmid_forced
      value: "0"
    - name: net.core.somaxconn
      value: "1024"
    - name: kernel.msgmax
      value: "65536"
  ...
```

2. Create the pod:

```
$ oc apply -f <file-name>.yaml
```

If the unsafe sysctls are not allowed for the node, the pod is scheduled, but does not deploy:

```
$ oc get pod
```

**Example output**

```
NAME       READY  STATUS         RESTARTS  AGE
hello-pod  0/1    SysctlForbidden  0         14s
```

### 6.9.3. Enabling unsafe sysctls

A cluster administrator can allow certain unsafe sysctls for very special situations such as high performance or real-time application tuning.

If you want to use unsafe sysctls, a cluster administrator must enable them individually for a specific type of node. The sysctls must be namespaced.

You can further control which sysctls are set in pods by specifying lists of sysctls or sysctl patterns in the **allowedUnsafeSysctls** field of the Security Context Constraints.

- The **allowedUnsafeSysctls** option controls specific needs such as high performance or real-time application tuning.

> **WARNING**
>
> Due to their nature of being unsafe, the use of unsafe sysctls is at-your-own-risk and can lead to severe problems, such as improper behavior of containers, resource shortage, or breaking a node.

**Procedure**

1. Add a label to the machine config pool where the containers where containers with the unsafe sysctls will run:

   ```
   $ oc edit machineconfigpool worker
   ```

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: MachineConfigPool
   metadata:
     creationTimestamp: 2019-02-08T14:52:39Z
     generation: 1
     labels:
       custom-kubelet: sysctl ❶
   ```

   ❶    Add a **key: pair** label.

2. Create a **KubeletConfig** custom resource (CR):

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: KubeletConfig
   metadata:
     name: custom-kubelet
   spec:
     machineConfigPoolSelector:
       matchLabels:
         custom-kubelet: sysctl ❶
     kubeletConfig:
       allowedUnsafeSysctls: ❷
         - "kernel.msg*"
         - "net.core.somaxconn"
   ```

   ❶    Specify the label from the machine config pool.

   ❷    List the unsafe sysctls you want to allow.

3. Create the object:

   ```
   $ oc apply -f set-sysctl-worker.yaml
   ```

   A new **MachineConfig** object named in the **99-worker-XXXXXX-XXXXX-XXXX-XXXXX-kubelet** format is created.

4. Wait for the cluster to reboot using the **machineconfigpool** object **status** fields:
   For example:

```
status:
  conditions:
    - lastTransitionTime: '2019-08-11T15:32:00Z'
      message: >-
        All nodes are updating to
        rendered-worker-ccbfb5d2838d65013ab36300b7b3dc13
      reason: ''
      status: 'True'
      type: Updating
```

A message similar to the following appears when the cluster is ready:

```
    - lastTransitionTime: '2019-08-11T16:00:00Z'
      message: >-
        All nodes are updated with
        rendered-worker-ccbfb5d2838d65013ab36300b7b3dc13
      reason: ''
      status: 'True'
      type: Updated
```

5. When the cluster is ready, check for the merged **KubeletConfig** object in the new **MachineConfig** object:

```
$ oc get machineconfig 99-worker-XXXXXX-XXXXX-XXXX-XXXXX-kubelet -o json | grep ownerReference -A7
```

```
"ownerReferences": [
  {
      "apiVersion": "machineconfiguration.openshift.io/v1",
      "blockOwnerDeletion": true,
      "controller": true,
      "kind": "KubeletConfig",
      "name": "custom-kubelet",
      "uid": "3f64a766-bae8-11e9-abe8-0a1a2a4813f2"
  }
]
```

You can now add unsafe sysctls to pods as needed.

# CHAPTER 7. WORKING WITH CLUSTERS

## 7.1. VIEWING SYSTEM EVENT INFORMATION IN AN OPENSHIFT CONTAINER PLATFORM CLUSTER

Events in OpenShift Container Platform are modeled based on events that happen to API objects in an OpenShift Container Platform cluster.

### 7.1.1. Understanding events

Events allow OpenShift Container Platform to record information about real-world events in a resource-agnostic manner. They also allow developers and administrators to consume information about system components in a unified way.

### 7.1.2. Viewing events using the CLI

You can get a list of events in a given project using the CLI.

**Procedure**

- To view events in a project use the following command:

  ```
  $ oc get events [-n <project>] 1
  ```

  **1** The name of the project.

  For example:

  ```
  $ oc get events -n openshift-config
  ```

**Example output**

```
LAST SEEN  TYPE     REASON              OBJECT                 MESSAGE
97m        Normal   Scheduled           pod/dapi-env-test-pod     Successfully assigned
openshift-config/dapi-env-test-pod to ip-10-0-171-202.ec2.internal
97m        Normal   Pulling             pod/dapi-env-test-pod     pulling image
"gcr.io/google_containers/busybox"
97m        Normal   Pulled              pod/dapi-env-test-pod     Successfully pulled image
"gcr.io/google_containers/busybox"
97m        Normal   Created             pod/dapi-env-test-pod     Created container
9m5s       Warning  FailedCreatePodSandBox  pod/dapi-volume-test-pod   Failed create
pod sandbox: rpc error: code = Unknown desc = failed to create pod network sandbox
k8s_dapi-volume-test-pod_openshift-config_6bc60c1f-452e-11e9-9140-
0eec59c23068_0(748c7a40db3d08c07fb4f9eba774bd5effe5f0d5090a242432a73eee66ba9e22
): Multus: Err adding pod to network "openshift-sdn": cannot set "openshift-sdn" ifname to
"eth0": no netns: failed to Statfs "/proc/33366/ns/net": no such file or directory
8m31s      Normal   Scheduled           pod/dapi-volume-test-pod   Successfully assigned
openshift-config/dapi-volume-test-pod to ip-10-0-171-202.ec2.internal
```

- To view events in your project from the OpenShift Container Platform console.

  1. Launch the OpenShift Container Platform console.

2. Click **Home** → **Events** and select your project.

3. Move to resource that you want to see events. For example: **Home** → **Projects** → \<project-name\> → \<resource-name\>.
   Many objects, such as pods and deployments, have their own **Events** tab as well, which shows events related to that object.

### 7.1.3. List of events

This section describes the events of OpenShift Container Platform.

Table 7.1. Configuration events

| Name | Description |
| --- | --- |
| **FailedValidation** | Failed pod configuration validation. |

Table 7.2. Container events

| Name | Description |
| --- | --- |
| **BackOff** | Back-off restarting failed the container. |
| **Created** | Container created. |
| **Failed** | Pull/Create/Start failed. |
| **Killing** | Killing the container. |
| **Started** | Container started. |
| **Preempting** | Preempting other pods. |
| **ExceededGrace Period** | Container runtime did not stop the pod within specified grace period. |

Table 7.3. Health events

| Name | Description |
| --- | --- |
| **Unhealthy** | Container is unhealthy. |

Table 7.4. Image events

| Name | Description |
| --- | --- |
| **BackOff** | Back off Ctr Start, image pull. |

| Name | Description |
|------|-------------|
| **ErrImageNeverPull** | The image's **NeverPull Policy** is violated. |
| **Failed** | Failed to pull the image. |
| **InspectFailed** | Failed to inspect the image. |
| **Pulled** | Successfully pulled the image or the container image is already present on the machine. |
| **Pulling** | Pulling the image. |

Table 7.5. Image Manager events

| Name | Description |
|------|-------------|
| **FreeDiskSpaceFailed** | Free disk space failed. |
| **InvalidDiskCapacity** | Invalid disk capacity. |

Table 7.6. Node events

| Name | Description |
|------|-------------|
| **FailedMount** | Volume mount failed. |
| **HostNetworkNotSupported** | Host network not supported. |
| **HostPortConflict** | Host/port conflict. |
| **KubeletSetupFailed** | Kubelet setup failed. |
| **NilShaper** | Undefined shaper. |
| **NodeNotReady** | Node is not ready. |
| **NodeNotSchedulable** | Node is not schedulable. |
| **NodeReady** | Node is ready. |

| Name | Description |
| --- | --- |
| **NodeSchedulable** | Node is schedulable. |
| **NodeSelectorMismatching** | Node selector mismatch. |
| **OutOfDisk** | Out of disk. |
| **Rebooted** | Node rebooted. |
| **Starting** | Starting kubelet. |
| **FailedAttachVolume** | Failed to attach volume. |
| **FailedDetachVolume** | Failed to detach volume. |
| **VolumeResizeFailed** | Failed to expand/reduce volume. |
| **VolumeResizeSuccessful** | Successfully expanded/reduced volume. |
| **FileSystemResizeFailed** | Failed to expand/reduce file system. |
| **FileSystemResizeSuccessful** | Successfully expanded/reduced file system. |
| **FailedUnMount** | Failed to unmount volume. |
| **FailedMapVolume** | Failed to map a volume. |
| **FailedUnmapDevice** | Failed unmaped device. |
| **AlreadyMountedVolume** | Volume is already mounted. |
| **SuccessfulDetachVolume** | Volume is successfully detached. |
| **SuccessfulMountVolume** | Volume is successfully mounted. |

| Name | Description |
|---|---|
| **SuccessfulUnMountVolume** | Volume is successfully unmounted. |
| **ContainerGCFailed** | Container garbage collection failed. |
| **ImageGCFailed** | Image garbage collection failed. |
| **FailedNodeAllocatableEnforcement** | Failed to enforce System Reserved Cgroup limit. |
| **NodeAllocatableEnforced** | Enforced System Reserved Cgroup limit. |
| **UnsupportedMountOption** | Unsupported mount option. |
| **SandboxChanged** | Pod sandbox changed. |
| **FailedCreatePodSandBox** | Failed to create pod sandbox. |
| **FailedPodSandBoxStatus** | Failed pod sandbox status. |

Table 7.7. Pod worker events

| Name | Description |
|---|---|
| **FailedSync** | Pod sync failed. |

Table 7.8. System Events

| Name | Description |
|---|---|
| **SystemOOM** | There is an OOM (out of memory) situation on the cluster. |

Table 7.9. Pod events

| Name | Description |
|---|---|
| **FailedKillPod** | Failed to stop a pod. |

| Name | Description |
| --- | --- |
| **FailedCreatePodContainer** | Failed to create a pod container. |
| **Failed** | Failed to make pod data directories. |
| **NetworkNotReady** | Network is not ready. |
| **FailedCreate** | Error creating: **<error-msg>**. |
| **SuccessfulCreate** | Created pod: **<pod-name>**. |
| **FailedDelete** | Error deleting: **<error-msg>**. |
| **SuccessfulDelete** | Deleted pod: **<pod-id>**. |

Table 7.10. Horizontal Pod AutoScaler events

| Name | Description |
| --- | --- |
| SelectorRequired | Selector is required. |
| **InvalidSelector** | Could not convert selector into a corresponding internal selector object. |
| **FailedGetObjectMetric** | HPA was unable to compute the replica count. |
| **InvalidMetricSourceType** | Unknown metric source type. |
| **ValidMetricFound** | HPA was able to successfully calculate a replica count. |
| **FailedConvertHPA** | Failed to convert the given HPA. |
| **FailedGetScale** | HPA controller was unable to get the target's current scale. |
| **SucceededGetScale** | HPA controller was able to get the target's current scale. |
| **FailedComputeMetricsReplicas** | Failed to compute desired number of replicas based on listed metrics. |

| Name | Description |
|------|-------------|
| **FailedRescale** | New size: **\<size\>**; reason:**\<msg\>**; error:**\<error-msg\>**. |
| **SuccessfulRescale** | New size: **\<size\>**; reason:**\<msg\>**. |
| **FailedUpdateStatus** | Failed to update status. |

Table 7.11. Network events (openshift-sdn)

| Name | Description |
|------|-------------|
| **Starting** | Starting OpenShift SDN. |
| **NetworkFailed** | The pod's network interface has been lost and the pod will be stopped. |

Table 7.12. Network events (kube-proxy)

| Name | Description |
|------|-------------|
| **NeedPods** | The service-port **\<serviceName\>:\<port\>** needs pods. |

Table 7.13. Volume events

| Name | Description |
|------|-------------|
| **FailedBinding** | There are no persistent volumes available and no storage class is set. |
| **VolumeMismatch** | Volume size or class is different from what is requested in claim. |
| **VolumeFailedRecycle** | Error creating recycler pod. |
| **VolumeRecycled** | Occurs when volume is recycled. |
| **RecyclerPod** | Occurs when pod is recycled. |
| **VolumeDelete** | Occurs when volume is deleted. |
| **VolumeFailedDelete** | Error when deleting the volume. |

| Name | Description |
| --- | --- |
| **ExternalProvisi oning** | Occurs when volume for the claim is provisioned either manually or via external software. |
| **ProvisioningFail ed** | Failed to provision volume. |
| **ProvisioningCle anupFailed** | Error cleaning provisioned volume. |
| **ProvisioningSu cceeded** | Occurs when the volume is provisioned successfully. |
| **WaitForFirstCo nsumer** | Delay binding until pod scheduling. |

### Table 7.14. Lifecycle hooks

| Name | Description |
| --- | --- |
| **FailedPostStart Hook** | Handler failed for pod start. |
| **FailedPreStopH ook** | Handler failed for pre-stop. |
| **UnfinishedPreSt opHook** | Pre-stop hook unfinished. |

### Table 7.15. Deployments

| Name | Description |
| --- | --- |
| **DeploymentCan cellationFailed** | Failed to cancel deployment. |
| **DeploymentCan celled** | Canceled deployment. |
| **DeploymentCre ated** | Created new replication controller. |
| **IngressIPRange Full** | No available Ingress IP to allocate to service. |

Table 7.16. Scheduler events

| Name | Description |
|------|-------------|
| **FailedScheduling** | Failed to schedule pod: **&lt;pod-namespace&gt;**/**&lt;pod-name&gt;**. This event is raised for multiple reasons, for example: **AssumePodVolumes** failed, Binding rejected etc. |
| **Preempted** | By **&lt;preemptor-namespace&gt;**/**&lt;preemptor-name&gt;** on node **&lt;node-name&gt;**. |
| **Scheduled** | Successfully assigned **&lt;pod-name&gt;** to **&lt;node-name&gt;**. |

Table 7.17. Daemon set events

| Name | Description |
|------|-------------|
| **SelectingAll** | This daemon set is selecting all pods. A non-empty selector is required. |
| **FailedPlacement** | Failed to place pod on **&lt;node-name&gt;**. |
| **FailedDaemonPod** | Found failed daemon pod **&lt;pod-name&gt;** on node **&lt;node-name&gt;**, will try to kill it. |

Table 7.18. LoadBalancer service events

| Name | Description |
|------|-------------|
| **CreatingLoadBalancerFailed** | Error creating load balancer. |
| **DeletingLoadBalancer** | Deleting load balancer. |
| **EnsuringLoadBalancer** | Ensuring load balancer. |
| **EnsuredLoadBalancer** | Ensured load balancer. |
| **UnAvailableLoadBalancer** | There are no available nodes for **LoadBalancer** service. |
| **LoadBalancerSourceRanges** | Lists the new **LoadBalancerSourceRanges**. For example, **&lt;old-source-range&gt;** → **&lt;new-source-range&gt;**. |
| **LoadbalancerIP** | Lists the new IP address. For example, **&lt;old-ip&gt;** → **&lt;new-ip&gt;**. |
| **ExternalIP** | Lists external IP address. For example, **Added: &lt;external-ip&gt;**. |

| Name | Description |
| --- | --- |
| **UID** | Lists the new UID. For example, **\<old-service-uid\>** → **\<new-service-uid\>**. |
| **ExternalTrafficPolicy** | Lists the new **ExternalTrafficPolicy**. For example,**\<old-policy\>** → **\<new-policy\>**. |
| **HealthCheckNodePort** | Lists the new **HealthCheckNodePort**. For example,**\<old-node-port\>** → **new-node-port\>**. |
| **UpdatedLoadBalancer** | Updated load balancer with new hosts. |
| **LoadBalancerUpdateFailed** | Error updating load balancer with new hosts. |
| **DeletingLoadBalancer** | Deleting load balancer. |
| **DeletingLoadBalancerFailed** | Error deleting load balancer. |
| **DeletedLoadBalancer** | Deleted load balancer. |

## 7.2. ESTIMATING THE NUMBER OF PODS YOUR OPENSHIFT CONTAINER PLATFORM NODES CAN HOLD

As a cluster administrator, you can use the cluster capacity tool to view the number of pods that can be scheduled to increase the current resources before they become exhausted, and to ensure any future pods can be scheduled. This capacity comes from an individual node host in a cluster, and includes CPU, memory, disk space, and others.

### 7.2.1. Understanding the OpenShift Container Platform cluster capacity tool

The cluster capacity tool simulates a sequence of scheduling decisions to determine how many instances of an input pod can be scheduled on the cluster before it is exhausted of resources to provide a more accurate estimation.

> **NOTE**
>
> The remaining allocatable capacity is a rough estimation, because it does not count all of the resources being distributed among nodes. It analyzes only the remaining resources and estimates the available capacity that is still consumable in terms of a number of instances of a pod with given requirements that can be scheduled in a cluster.
>
> Also, pods might only have scheduling support on particular sets of nodes based on its selection and affinity criteria. As a result, the estimation of which remaining pods a cluster can schedule can be difficult.

You can run the cluster capacity analysis tool as a stand-alone utility from the command line, or as a job in a pod inside an OpenShift Container Platform cluster. Running it as job inside of a pod enables you to run it multiple times without intervention.

## 7.2.2. Running the cluster capacity tool on the command line

You can run the OpenShift Container Platform cluster capacity tool from the command line to estimate the number of pods that can be scheduled onto your cluster.

### Prerequisites

- Run the OpenShift Cluster Capacity Tool, which is available as a container image from the Red Hat Ecosystem Catalog.

- Create a sample **Pod** spec file, which the tool uses for estimating resource usage. The **podspec** specifies its resource requirements as **limits** or **requests**. The cluster capacity tool takes the pod's resource requirements into account for its estimation analysis.
  An example of the **Pod** spec input is:

  ```
  apiVersion: v1
  kind: Pod
  metadata:
    name: small-pod
    labels:
      app: guestbook
      tier: frontend
  spec:
    containers:
    - name: php-redis
      image: gcr.io/google-samples/gb-frontend:v4
      imagePullPolicy: Always
      resources:
        limits:
          cpu: 150m
          memory: 100Mi
        requests:
          cpu: 150m
          memory: 100Mi
  ```

### Procedure

To use the cluster capacity tool on the command line:

1. From the terminal, log in to the Red Hat Registry:

   ```
   $ podman login registry.redhat.io
   ```

2. Pull the cluster capacity tool image:

   ```
   $ podman pull registry.redhat.io/openshift4/ose-cluster-capacity
   ```

3. Run the cluster capacity tool:

```
$ podman run -v $HOME/.kube:/kube:Z -v $(pwd):/cc:Z  ose-cluster-capacity \
/bin/cluster-capacity --kubeconfig /kube/config --podspec /cc/pod-spec.yaml \
--verbose 1
```

**1**  You can also add the **--verbose** option to output a detailed description of how many pods can be scheduled on each node in the cluster.

**Example output**

```
small-pod pod requirements:
 - CPU: 150m
 - Memory: 100Mi

The cluster can schedule 88 instance(s) of the pod small-pod.

Termination reason: Unschedulable: 0/5 nodes are available: 2 Insufficient cpu,
3 node(s) had taint {node-role.kubernetes.io/master: }, that the pod didn't
tolerate.

Pod distribution among nodes:
small-pod
 - 192.168.124.214: 45 instance(s)
 - 192.168.124.120: 43 instance(s)
```

In the above example, the number of estimated pods that can be scheduled onto the cluster is 88.

### 7.2.3. Running the cluster capacity tool as a job inside a pod

Running the cluster capacity tool as a job inside of a pod has the advantage of being able to be run multiple times without needing user intervention. Running the cluster capacity tool as a job involves using a **ConfigMap** object.

#### Prerequisites

Download and install the cluster capacity tool.

#### Procedure

To run the cluster capacity tool:

1. Create the cluster role:

   ```
   $ cat << EOF| oc create -f -
   ```

   **Example output**

   ```
   kind: ClusterRole
   apiVersion: rbac.authorization.k8s.io/v1
   metadata:
     name: cluster-capacity-role
   rules:
   - apiGroups: [""]
     resources: ["pods", "nodes", "persistentvolumeclaims", "persistentvolumes", "services",
   ```

```
  "replicationcontrollers"]
    verbs: ["get", "watch", "list"]
  - apiGroups: ["apps"]
    resources: ["replicasets", "statefulsets"]
    verbs: ["get", "watch", "list"]
  - apiGroups: ["policy"]
    resources: ["poddisruptionbudgets"]
    verbs: ["get", "watch", "list"]
  - apiGroups: ["storage.k8s.io"]
    resources: ["storageclasses"]
    verbs: ["get", "watch", "list"]
  EOF
```

2. Create the service account:

```
$ oc create sa cluster-capacity-sa
```

3. Add the role to the service account:

```
$ oc adm policy add-cluster-role-to-user cluster-capacity-role \
    system:serviceaccount:default:cluster-capacity-sa
```

4. Define and create the **Pod** spec:

```
apiVersion: v1
kind: Pod
metadata:
  name: small-pod
  labels:
    app: guestbook
    tier: frontend
spec:
  containers:
  - name: php-redis
    image: gcr.io/google-samples/gb-frontend:v4
    imagePullPolicy: Always
    resources:
      limits:
        cpu: 150m
        memory: 100Mi
      requests:
        cpu: 150m
        memory: 100Mi
```

5. The cluster capacity analysis is mounted in a volume using a **ConfigMap** object named **cluster-capacity-configmap** to mount input pod spec file **pod.yaml** into a volume **test-volume** at the path /**test-pod**.
   If you haven't created a **ConfigMap** object, create one before creating the job:

```
$ oc create configmap cluster-capacity-configmap \
    --from-file=pod.yaml=pod.yaml
```

6. Create the job using the below example of a job specification file:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: cluster-capacity-job
spec:
  parallelism: 1
  completions: 1
  template:
    metadata:
      name: cluster-capacity-pod
    spec:
        containers:
        - name: cluster-capacity
          image: openshift/origin-cluster-capacity
          imagePullPolicy: "Always"
          volumeMounts:
          - mountPath: /test-pod
            name: test-volume
          env:
          - name: CC_INCLUSTER ❶
            value: "true"
          command:
          - "/bin/sh"
          - "-ec"
          - |
            /bin/cluster-capacity --podspec=/test-pod/pod.yaml --verbose
        restartPolicy: "Never"
        serviceAccountName: cluster-capacity-sa
        volumes:
        - name: test-volume
          configMap:
            name: cluster-capacity-configmap
```

❶ A required environment variable letting the cluster capacity tool know that it is running inside a cluster as a pod.
The **pod.yaml** key of the **ConfigMap** object is the same as the **Pod** spec file name, though it is not required. By doing this, the input pod spec file can be accessed inside the pod as **/test-pod/pod.yaml**.

7. Run the cluster capacity image as a job in a pod:

   ```
   $ oc create -f cluster-capacity-job.yaml
   ```

8. Check the job logs to find the number of pods that can be scheduled in the cluster:

   ```
   $ oc logs jobs/cluster-capacity-job
   ```

**Example output**

```
small-pod pod requirements:
        - CPU: 150m
        - Memory: 100Mi
```

> The cluster can schedule 52 instance(s) of the pod small-pod.
>
> Termination reason: Unschedulable: No nodes are available that match all of the following predicates:: Insufficient cpu (2).
>
> Pod distribution among nodes:
> small-pod
>         - 192.168.124.214: 26 instance(s)
>         - 192.168.124.120: 26 instance(s)

## 7.3. RESTRICT RESOURCE CONSUMPTION WITH LIMIT RANGES

By default, containers run with unbounded compute resources on an OpenShift Container Platform cluster. With limit ranges, you can restrict resource consumption for specific objects in a project:

- pods and containers: You can set minimum and maximum requirements for CPU and memory for pods and their containers.

- Image streams: You can set limits on the number of images and tags in an **ImageStream** object.

- Images: You can limit the size of images that can be pushed to an internal registry.

- Persistent volume claims (PVC): You can restrict the size of the PVCs that can be requested.

If a pod does not meet the constraints imposed by the limit range, the pod cannot be created in the namespace.

### 7.3.1. About limit ranges

A limit range, defined by a **LimitRange** object, restricts resource consumption in a project. In the project you can set specific resource limits for a pod, container, image, image stream, or persistent volume claim (PVC).

All requests to create and modify resources are evaluated against each **LimitRange** object in the project. If the resource violates any of the enumerated constraints, the resource is rejected.

The following shows a limit range object for all components: pod, container, image, image stream, or PVC. You can configure limits for any or all of these components in the same object. You create a different limit range object for each project where you want to control resources.

**Sample limit range object for a container**

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits"
spec:
  limits:
    - type: "Container"
      max:
        cpu: "2"
        memory: "1Gi"
      min:
        cpu: "100m"
        memory: "4Mi"
```

```
    default:
      cpu: "300m"
      memory: "200Mi"
    defaultRequest:
      cpu: "200m"
      memory: "100Mi"
    maxLimitRequestRatio:
      cpu: "10"
```

### 7.3.1.1. About component limits

The following examples show limit range parameters for each component. The examples are broken out for clarity. You can create a single **LimitRange** object for any or all components as necessary.

#### 7.3.1.1.1. Container limits

A limit range allows you to specify the minimum and maximum CPU and memory that each container in a pod can request for a specific project. If a container is created in the project, the container CPU and memory requests in the **Pod** spec must comply with the values set in the **LimitRange** object. If not, the pod does not get created.

- The container CPU or memory request and limit must be greater than or equal to the **min** resource constraint for containers that are specified in the **LimitRange** object.

- The container CPU or memory request and limit must be less than or equal to the **max** resource constraint for containers that are specified in the **LimitRange** object.
  If the **LimitRange** object defines a **max** CPU, you do not need to define a CPU **request** value in the **Pod** spec. But you must specify a CPU **limit** value that satisfies the maximum CPU constraint specified in the limit range.

- The ratio of the container limits to requests must be less than or equal to the **maxLimitRequestRatio** value for containers that is specified in the **LimitRange** object.
  If the **LimitRange** object defines a **maxLimitRequestRatio** constraint, any new containers must have both a **request** and a **limit** value. OpenShift Container Platform calculates the limit-to-request ratio by dividing the **limit** by the **request**. This value should be a non-negative integer greater than 1.

  For example, if a container has **cpu: 500** in the **limit** value, and **cpu: 100** in the **request** value, the limit-to-request ratio for **cpu** is **5**. This ratio must be less than or equal to the **maxLimitRequestRatio**.

If the **Pod** spec does not specify a container resource memory or limit, the **default** or **defaultRequest** CPU and memory values for containers specified in the limit range object are assigned to the container.

**Container LimitRange object definition**

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits"  1
spec:
  limits:
    - type: "Container"
      max:
        cpu: "2"  2
```

```
      memory: "1Gi" 3
    min:
      cpu: "100m" 4
      memory: "4Mi" 5
    default:
      cpu: "300m" 6
      memory: "200Mi" 7
    defaultRequest:
      cpu: "200m" 8
      memory: "100Mi" 9
    maxLimitRequestRatio:
      cpu: "10" 10
```

**1**    The name of the LimitRange object.

**2**    The maximum amount of CPU that a single container in a pod can request.

**3**    The maximum amount of memory that a single container in a pod can request.

**4**    The minimum amount of CPU that a single container in a pod can request.

**5**    The minimum amount of memory that a single container in a pod can request.

**6**    The default amount of CPU that a container can use if not specified in the **Pod** spec.

**7**    The default amount of memory that a container can use if not specified in the **Pod** spec.

**8**    The default amount of CPU that a container can request if not specified in the **Pod** spec.

**9**    The default amount of memory that a container can request if not specified in the **Pod** spec.

**10**    The maximum limit-to-request ratio for a container.

### 7.3.1.1.2. Pod limits

A limit range allows you to specify the minimum and maximum CPU and memory limits for all containers across a pod in a given project. To create a container in the project, the container CPU and memory requests in the **Pod** spec must comply with the values set in the **LimitRange** object. If not, the pod does not get created.

If the **Pod** spec does not specify a container resource memory or limit, the **default** or **defaultRequest** CPU and memory values for containers specified in the limit range object are assigned to the container.

Across all containers in a pod, the following must hold true:

- The container CPU or memory request and limit must be greater than or equal to the **min** resource constraints for pods that are specified in the **LimitRange** object.

- The container CPU or memory request and limit must be less than or equal to the **max** resource constraints for pods that are specified in the **LimitRange** object.

- The ratio of the container limits to requests must be less than or equal to the **maxLimitRequestRatio** constraint specified in the **LimitRange** object.

Pod **LimitRange** object definition

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" 1
spec:
  limits:
    - type: "Pod"
      max:
        cpu: "2" 2
        memory: "1Gi" 3
      min:
        cpu: "200m" 4
        memory: "6Mi" 5
      maxLimitRequestRatio:
        cpu: "10" 6
```

**1**    The name of the limit range object.

**2**    The maximum amount of CPU that a pod can request across all containers.

**3**    The maximum amount of memory that a pod can request across all containers.

**4**    The minimum amount of CPU that a pod can request across all containers.

**5**    The minimum amount of memory that a pod can request across all containers.

**6**    The maximum limit-to-request ratio for a container.

### 7.3.1.1.3. Image limits

A **LimitRange** object allows you to specify the maximum size of an image that can be pushed to an internal registry.

When pushing images to an internal registry, the following must hold true:

- The size of the image must be less than or equal to the **max** size for images that is specified in the **LimitRange** object.

## Image **LimitRange** object definition

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" 1
spec:
  limits:
    - type: openshift.io/Image
      max:
        storage: 1Gi 2
```

**1**    The name of the **LimitRange** object.

**2**    The maximum size of an image that can be pushed to an internal registry.

NOTE

To prevent blobs that exceed the limit from being uploaded to the registry, the registry must be configured to enforce quotas.

WARNING

The image size is not always available in the manifest of an uploaded image. This is especially the case for images built with Docker 1.10 or higher and pushed to a v2 registry. If such an image is pulled with an older Docker daemon, the image manifest is converted by the registry to schema v1 lacking all the size information. No storage limit set on images prevent it from being uploaded.

The issue is being addressed.

### 7.3.1.1.4. Image stream limits

A **LimitRange** object allows you to specify limits for image streams.

For each image stream, the following must hold true:

- The number of image tags in an **ImageStream** specification must be less than or equal to the **openshift.io/image-tags** constraint in the **LimitRange** object.

- The number of unique references to images in an **ImageStream** specification must be less than or equal to the **openshift.io/images** constraint in the limit range object.

Imagestream **LimitRange** object definition

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" 1
spec:
  limits:
    - type: openshift.io/ImageStream
      max:
        openshift.io/image-tags: 20 2
        openshift.io/images: 30 3
```

**1** The name of the **LimitRange** object.

**2** The maximum number of unique image tags in the **imagestream.spec.tags** parameter in imagestream spec.

**3** The maximum number of unique image references in the **imagestream.status.tags** parameter in the **imagestream** spec.

The **openshift.io/image-tags** resource represents unique image references. Possible references are an **ImageStreamTag**, an **ImageStreamImage** and a **DockerImage**. Tags can be created using the **oc tag**

and **oc import-image** commands. No distinction is made between internal and external references. However, each unique reference tagged in an **ImageStream** specification is counted just once. It does not restrict pushes to an internal container image registry in any way, but is useful for tag restriction.

The **openshift.io/images** resource represents unique image names recorded in image stream status. It allows for restriction of a number of images that can be pushed to the internal registry. Internal and external references are not distinguished.

### 7.3.1.1.5. Persistent volume claim limits

A **LimitRange** object allows you to restrict the storage requested in a persistent volume claim (PVC).

Across all persistent volume claims in a project, the following must hold true:

- The resource request in a persistent volume claim (PVC) must be greater than or equal the **min** constraint for PVCs that is specified in the **LimitRange** object.

- The resource request in a persistent volume claim (PVC) must be less than or equal the **max** constraint for PVCs that is specified in the **LimitRange** object.

PVC **LimitRange** object definition

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" 1
spec:
  limits:
    - type: "PersistentVolumeClaim"
      min:
        storage: "2Gi" 2
      max:
        storage: "50Gi" 3
```

**1** The name of the **LimitRange** object.

**2** The minimum amount of storage that can be requested in a persistent volume claim.

**3** The maximum amount of storage that can be requested in a persistent volume claim.

## 7.3.2. Creating a Limit Range

To apply a limit range to a project:

1. Create a **LimitRange** object with your required specifications:

   ```
   apiVersion: "v1"
   kind: "LimitRange"
   metadata:
     name: "resource-limits" 1
   spec:
     limits:
       - type: "Pod" 2
         max:
   ```

```
      cpu: "2"
      memory: "1Gi"
    min:
      cpu: "200m"
      memory: "6Mi"
  - type: "Container" 3
    max:
      cpu: "2"
      memory: "1Gi"
    min:
      cpu: "100m"
      memory: "4Mi"
    default: 4
      cpu: "300m"
      memory: "200Mi"
    defaultRequest: 5
      cpu: "200m"
      memory: "100Mi"
    maxLimitRequestRatio: 6
      cpu: "10"
  - type: openshift.io/Image 7
    max:
      storage: 1Gi
  - type: openshift.io/ImageStream 8
    max:
      openshift.io/image-tags: 20
      openshift.io/images: 30
  - type: "PersistentVolumeClaim" 9
    min:
      storage: "2Gi"
    max:
      storage: "50Gi"
```

**1** Specify a name for the **LimitRange** object.

**2** To set limits for a pod, specify the minimum and maximum CPU and memory requests as needed.

**3** To set limits for a container, specify the minimum and maximum CPU and memory requests as needed.

**4** Optional. For a container, specify the default amount of CPU or memory that a container can use, if not specified in the **Pod** spec.

**5** Optional. For a container, specify the default amount of CPU or memory that a container can request, if not specified in the **Pod** spec.

**6** Optional. For a container, specify the maximum limit-to-request ratio that can be specified in the **Pod** spec.

**7** To set limits for an Image object, set the maximum size of an image that can be pushed to an internal registry.

**8** To set limits for an image stream, set the maximum number of image tags and references that can be in the **ImageStream** object file, as needed.

**9**    To set limits for a persistent volume claim, set the minimum and maximum amount of storage that can be requested.

2. Create the object:

```
$ oc create -f <limit_range_file> -n <project> ❶
```

**1**    Specify the name of the YAML file you created and the project where you want the limits to apply.

### 7.3.3. Viewing a limit

You can view any limits defined in a project by navigating in the web console to the project's **Quota** page.

You can also use the CLI to view limit range details:

1. Get the list of **LimitRange** object defined in the project. For example, for a project called **demoproject**:

```
$ oc get limits -n demoproject
```

```
NAME            CREATED AT
resource-limits   2020-07-15T17:14:23Z
```

2. Describe the **LimitRange** object you are interested in, for example the **resource-limits** limit range:

```
$ oc describe limits resource-limits -n demoproject
```

```
Name:                    resource-limits
Namespace:               demoproject
Type                    Resource          Min   Max    Default Request Default Limit  Max Limit/Request Ratio
----                    --------          ---   ---    -------------- -------------  ----------------------
Pod                     cpu               200m  2      -              -              -
Pod                     memory            6Mi   1Gi    -              -              -
Container               cpu               100m  2      200m           300m           10
Container               memory            4Mi   1Gi    100Mi          200Mi          -
openshift.io/Image      storage           -     1Gi    -              -              -
openshift.io/ImageStream   openshift.io/image   -     12     -              -              -
openshift.io/ImageStream   openshift.io/image-tags -   10     -              -              -
PersistentVolumeClaim   storage           -     50Gi   -              -              -
```

### 7.3.4. Deleting a Limit Range

To remove any active **LimitRange** object to no longer enforce the limits in a project:

1. Run the following command:

```
$ oc delete limits <limit_name>
```

# 7.4. CONFIGURING CLUSTER MEMORY TO MEET CONTAINER MEMORY AND RISK REQUIREMENTS

As a cluster administrator, you can help your clusters operate efficiently through managing application memory by:

- Determining the memory and risk requirements of a containerized application component and configuring the container memory parameters to suit those requirements.

- Configuring containerized application runtimes (for example, OpenJDK) to adhere optimally to the configured container memory parameters.

- Diagnosing and resolving memory-related error conditions associated with running in a container.

## 7.4.1. Understanding managing application memory

It is recommended to fully read the overview of how OpenShift Container Platform manages Compute Resources before proceeding.

For each kind of resource (memory, CPU, storage), OpenShift Container Platform allows optional **request** and **limit** values to be placed on each container in a pod.

Note the following about memory requests and memory limits:

- **Memory request**

  - The memory request value, if specified, influences the OpenShift Container Platform scheduler. The scheduler considers the memory request when scheduling a container to a node, then fences off the requested memory on the chosen node for the use of the container.

  - If a node's memory is exhausted, OpenShift Container Platform prioritizes evicting its containers whose memory usage most exceeds their memory request. In serious cases of memory exhaustion, the node OOM killer may select and kill a process in a container based on a similar metric.

  - The cluster administrator can assign quota or assign default values for the memory request value.

  - The cluster administrator can override the memory request values that a developer specifies, to manage cluster overcommit.

- **Memory limit**

  - The memory limit value, if specified, provides a hard limit on the memory that can be allocated across all the processes in a container.

  - If the memory allocated by all of the processes in a container exceeds the memory limit, the node Out of Memory (OOM) killer will immediately select and kill a process in the container.

  - If both memory request and limit are specified, the memory limit value must be greater than or equal to the memory request.

  - The cluster administrator can assign quota or assign default values for the memory limit value.

- The minimum memory limit is 12 MB. If a container fails to start due to a **Cannot allocate memory** pod event, the memory limit is too low. Either increase or remove the memory limit. Removing the limit allows pods to consume unbounded node resources.

### 7.4.1.1. Managing application memory strategy

The steps for sizing application memory on OpenShift Container Platform are as follows:

1. **Determine expected container memory usage**
   Determine expected mean and peak container memory usage, empirically if necessary (for example, by separate load testing). Remember to consider all the processes that may potentially run in parallel in the container: for example, does the main application spawn any ancillary scripts?

2. **Determine risk appetite**
   Determine risk appetite for eviction. If the risk appetite is low, the container should request memory according to the expected peak usage plus a percentage safety margin. If the risk appetite is higher, it may be more appropriate to request memory according to the expected mean usage.

3. **Set container memory request**
   Set container memory request based on the above. The more accurately the request represents the application memory usage, the better. If the request is too high, cluster and quota usage will be inefficient. If the request is too low, the chances of application eviction increase.

4. **Set container memory limit, if required**
   Set container memory limit, if required. Setting a limit has the effect of immediately killing a container process if the combined memory usage of all processes in the container exceeds the limit, and is therefore a mixed blessing. On the one hand, it may make unanticipated excess memory usage obvious early ("fail fast"); on the other hand it also terminates processes abruptly.

   Note that some OpenShift Container Platform clusters may require a limit value to be set; some may override the request based on the limit; and some application images rely on a limit value being set as this is easier to detect than a request value.

   If the memory limit is set, it should not be set to less than the expected peak container memory usage plus a percentage safety margin.

5. **Ensure application is tuned**
   Ensure application is tuned with respect to configured request and limit values, if appropriate. This step is particularly relevant to applications which pool memory, such as the JVM. The rest of this page discusses this.

**Additional resources**

- Understanding compute resources and containers

## 7.4.2. Understanding OpenJDK settings for OpenShift Container Platform

The default OpenJDK settings do not work well with containerized environments. As a result, some additional Java memory settings must always be provided whenever running the OpenJDK in a container.

The JVM memory layout is complex, version dependent, and describing it in detail is beyond the scope of this documentation. However, as a starting point for running OpenJDK in a container, at least the following three memory-related tasks are key:

1. Overriding the JVM maximum heap size.

2. Encouraging the JVM to release unused memory to the operating system, if appropriate.

3. Ensuring all JVM processes within a container are appropriately configured.

Optimally tuning JVM workloads for running in a container is beyond the scope of this documentation, and may involve setting multiple additional JVM options.

### 7.4.2.1. Understanding how to override the JVM maximum heap size

For many Java workloads, the JVM heap is the largest single consumer of memory. Currently, the OpenJDK defaults to allowing up to 1/4 (1/**-XX:MaxRAMFraction**) of the compute node's memory to be used for the heap, regardless of whether the OpenJDK is running in a container or not. It is therefore **essential** to override this behavior, especially if a container memory limit is also set.

There are at least two ways the above can be achieved:

1. If the container memory limit is set and the experimental options are supported by the JVM, set **-XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap**.

   > **NOTE**
   >
   > The **UseCGroupMemoryLimitForHeap** option has been removed in JDK 11. Use **-XX:+UseContainerSupport** instead.

   This sets **-XX:MaxRAM** to the container memory limit, and the maximum heap size ( **-XX:MaxHeapSize** / **-Xmx**) to 1/ **-XX:MaxRAMFraction** (1/4 by default).

2. Directly override one of **-XX:MaxRAM**, **-XX:MaxHeapSize** or **-Xmx**.
   This option involves hard-coding a value, but has the advantage of allowing a safety margin to be calculated.

### 7.4.2.2. Understanding how to encourage the JVM to release unused memory to the operating system

By default, the OpenJDK does not aggressively return unused memory to the operating system. This may be appropriate for many containerized Java workloads, but notable exceptions include workloads where additional active processes co-exist with a JVM within a container, whether those additional processes are native, additional JVMs, or a combination of the two.

The OpenShift Container Platform Jenkins maven slave image uses the following JVM arguments to encourage the JVM to release unused memory to the operating system:

```
-XX:+UseParallelGC
-XX:MinHeapFreeRatio=5 -XX:MaxHeapFreeRatio=10 -XX:GCTimeRatio=4
-XX:AdaptiveSizePolicyWeight=90.
```

These arguments are intended to return heap memory to the operating system whenever allocated memory exceeds 110% of in-use memory (**-XX:MaxHeapFreeRatio**), spending up to 20% of CPU time in the garbage collector (**-XX:GCTimeRatio**). At no time will the application heap allocation be less than

the initial heap allocation (overridden by **-XX:InitialHeapSize** / **-Xms**). Detailed additional information is available Tuning Java's footprint in OpenShift (Part 1) , Tuning Java's footprint in OpenShift (Part 2) , and at OpenJDK and Containers.

### 7.4.2.3. Understanding how to ensure all JVM processes within a container are appropriately configured

In the case that multiple JVMs run in the same container, it is essential to ensure that they are all configured appropriately. For many workloads it will be necessary to grant each JVM a percentage memory budget, leaving a perhaps substantial additional safety margin.

Many Java tools use different environment variables (**JAVA_OPTS**, **GRADLE_OPTS**, **MAVEN_OPTS**, and so on) to configure their JVMs and it can be challenging to ensure that the right settings are being passed to the right JVM.

The **JAVA_TOOL_OPTIONS** environment variable is always respected by the OpenJDK, and values specified in **JAVA_TOOL_OPTIONS** will be overridden by other options specified on the JVM command line. By default, to ensure that these options are used by default for all JVM workloads run in the slave image, the OpenShift Container Platform Jenkins maven slave image sets:

```
JAVA_TOOL_OPTIONS="-XX:+UnlockExperimentalVMOptions
-XX:+UseCGroupMemoryLimitForHeap -Dsun.zip.disableMemoryMapping=true"
```

> **NOTE**
>
> The **UseCGroupMemoryLimitForHeap** option has been removed in JDK 11. Use **-XX:+UseContainerSupport** instead.

This does not guarantee that additional options are not required, but is intended to be a helpful starting point.

### 7.4.3. Finding the memory request and limit from within a pod

An application wishing to dynamically discover its memory request and limit from within a pod should use the Downward API.

**Procedure**

1. Configure the pod to add the **MEMORY_REQUEST** and **MEMORY_LIMIT** stanzas:

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: test
   spec:
     containers:
     - name: test
       image: fedora:latest
       command:
       - sleep
       - "3600"
       env:
       - name: MEMORY_REQUEST ❶
         valueFrom:
   ```

```
      resourceFieldRef:
        containerName: test
        resource: requests.memory
  - name: MEMORY_LIMIT 2
    valueFrom:
      resourceFieldRef:
        containerName: test
        resource: limits.memory
  resources:
    requests:
      memory: 384Mi
    limits:
      memory: 512Mi
```

**1**    Add this stanza to discover the application memory request value.

**2**    Add this stanza to discover the application memory limit value.

2. Create the pod:

   ```
   $ oc create -f <file-name>.yaml
   ```

3. Access the pod using a remote shell:

   ```
   $ oc rsh test
   ```

4. Check that the requested values were applied:

   ```
   $ env | grep MEMORY | sort
   ```

   **Example output**

   ```
   MEMORY_LIMIT=536870912
   MEMORY_REQUEST=402653184
   ```
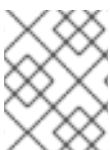
> **NOTE**
>
> The memory limit value can also be read from inside the container by the
> **/sys/fs/cgroup/memory/memory.limit_in_bytes** file.

## 7.4.4. Understanding OOM kill policy

OpenShift Container Platform can kill a process in a container if the total memory usage of all the processes in the container exceeds the memory limit, or in serious cases of node memory exhaustion.

When a process is Out of Memory (OOM) killed, this might result in the container exiting immediately. If the container PID 1 process receives the **SIGKILL**, the container will exit immediately. Otherwise, the container behavior is dependent on the behavior of the other processes.

For example, a container process exited with code 137, indicating it received a SIGKILL signal.

If the container does not exit immediately, an OOM kill is detectable as follows:

1. Access the pod using a remote shell:

   ```
   # oc rsh test
   ```

2. Run the following command to see the current OOM kill count in
   **/sys/fs/cgroup/memory/memory.oom_control**:

   ```
   $ grep '^oom_kill ' /sys/fs/cgroup/memory/memory.oom_control
   oom_kill 0
   ```

3. Run the following command to provoke an OOM kill:

   ```
   $ sed -e '' </dev/zero
   ```

   **Example output**

   ```
   Killed
   ```

4. Run the following command to view the exit status of the **sed** command:

   ```
   $ echo $?
   ```

   **Example output**

   ```
   137
   ```

   The **137** code indicates the container process exited with code 137, indicating it received a
   SIGKILL signal.

5. Run the following command to see that the OOM kill counter in
   **/sys/fs/cgroup/memory/memory.oom_control** incremented:

   ```
   $ grep '^oom_kill ' /sys/fs/cgroup/memory/memory.oom_control
   oom_kill 1
   ```

   If one or more processes in a pod are OOM killed, when the pod subsequently exits, whether
   immediately or not, it will have phase **Failed** and reason **OOMKilled**. An OOM-killed pod might
   be restarted depending on the value of **restartPolicy**. If not restarted, controllers such as the
   replication controller will notice the pod's failed status and create a new pod to replace the old
   one.

   Use the follwing command to get the pod status:

   ```
   $ oc get pod test
   ```

   **Example output**

   ```
   NAME    READY   STATUS     RESTARTS   AGE
   test    0/1     OOMKilled  0          1m
   ```

   - If the pod has not restarted, run the following command to view the pod:

```
$ oc get pod test -o yaml
```

**Example output**

```
...
status:
  containerStatuses:
  - name: test
    ready: false
    restartCount: 0
    state:
      terminated:
        exitCode: 137
        reason: OOMKilled
  phase: Failed
```

- If restarted, run the following command to view the pod:

```
$ oc get pod test -o yaml
```

**Example output**

```
...
status:
  containerStatuses:
  - name: test
    ready: true
    restartCount: 1
    lastState:
      terminated:
        exitCode: 137
        reason: OOMKilled
    state:
      running:
  phase: Running
```

### 7.4.5. Understanding pod eviction

OpenShift Container Platform may evict a pod from its node when the node's memory is exhausted. Depending on the extent of memory exhaustion, the eviction may or may not be graceful. Graceful eviction implies the main process (PID 1) of each container receiving a SIGTERM signal, then some time later a SIGKILL signal if the process has not exited already. Non-graceful eviction implies the main process of each container immediately receiving a SIGKILL signal.

An evicted pod has phase **Failed** and reason **Evicted**. It will not be restarted, regardless of the value of **restartPolicy**. However, controllers such as the replication controller will notice the pod's failed status and create a new pod to replace the old one.

```
$ oc get pod test
```

**Example output**

```
NAME    READY    STATUS   RESTARTS   AGE
test    0/1      Evicted  0          1m
```

```
$ oc get pod test -o yaml
```

**Example output**

```
...
status:
  message: 'Pod The node was low on resource: [MemoryPressure].'
  phase: Failed
  reason: Evicted
```

## 7.5. CONFIGURING YOUR CLUSTER TO PLACE PODS ON OVERCOMMITTED NODES

In an *overcommitted* state, the sum of the container compute resource requests and limits exceeds the resources available on the system. For example, you might want to use overcommitment in development environments where a trade-off of guaranteed performance for capacity is acceptable.

Containers can specify compute resource requests and limits. Requests are used for scheduling your container and provide a minimum service guarantee. Limits constrain the amount of compute resource that can be consumed on your node.

The scheduler attempts to optimize the compute resource use across all nodes in your cluster. It places pods onto specific nodes, taking the pods' compute resource requests and nodes' available capacity into consideration.

OpenShift Container Platform administrators can control the level of overcommit and manage container density on nodes. You can configure cluster-level overcommit using the ClusterResourceOverride Operator to override the ratio between requests and limits set on developer containers. In conjunction with node overcommit and project memory and CPU limits and defaults , you can adjust the resource limit and request to achieve the desired level of overcommit.
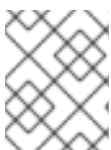
> **NOTE**
>
> In OpenShift Container Platform, you must enable cluster-level overcommit. Node overcommitment is enabled by default. See Disabling overcommitment for a node .

### 7.5.1. Resource requests and overcommitment

For each compute resource, a container may specify a resource request and limit. Scheduling decisions are made based on the request to ensure that a node has enough capacity available to meet the requested value. If a container specifies limits, but omits requests, the requests are defaulted to the limits. A container is not able to exceed the specified limit on the node.

The enforcement of limits is dependent upon the compute resource type. If a container makes no request or limit, the container is scheduled to a node with no resource guarantees. In practice, the container is able to consume as much of the specified resource as is available with the lowest local priority. In low resource situations, containers that specify no resource requests are given the lowest quality of service.

Scheduling is based on resources requested, while quota and hard limits refer to resource limits, which

can be set higher than requested resources. The difference between request and limit determines the level of overcommit; for instance, if a container is given a memory request of 1Gi and a memory limit of 2Gi, it is scheduled based on the 1Gi request being available on the node, but could use up to 2Gi; so it is 200% overcommitted.

## 7.5.2. Cluster-level overcommit using the Cluster Resource Override Operator

The Cluster Resource Override Operator is an admission webhook that allows you to control the level of overcommit and manage container density across all the nodes in your cluster. The Operator controls how nodes in specific projects can exceed defined memory and CPU limits.

You must install the Cluster Resource Override Operator using the OpenShift Container Platform console or CLI as shown in the following sections. During the installation, you create a **ClusterResourceOverride** custom resource (CR), where you set the level of overcommit, as shown in the following example:

```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
    name: cluster 1
spec:
  podResourceOverride:
    spec:
      memoryRequestToLimitPercent: 50 2
      cpuRequestToLimitPercent: 25 3
      limitCPUToMemoryPercent: 200 4
```

**1** The name must be **cluster**.

**2** Optional. If a container memory limit has been specified or defaulted, the memory request is overridden to this percentage of the limit, between 1-100. The default is 50.

**3** Optional. If a container CPU limit has been specified or defaulted, the CPU request is overridden to this percentage of the limit, between 1-100. The default is 25.

**4** Optional. If a container memory limit has been specified or defaulted, the CPU limit is overridden to a percentage of the memory limit, if specified. Scaling 1Gi of RAM at 100 percent is equal to 1 CPU core. This is processed prior to overriding the CPU request (if configured). The default is 200.

> **NOTE**
>
> The Cluster Resource Override Operator overrides have no effect if limits have not been set on containers. Create a **LimitRange** object with default limits per individual project or configure limits in **Pod** specs for the overrides to apply.

When configured, overrides can be enabled per-project by applying the following label to the Namespace object for each project:

```
apiVersion: v1
kind: Namespace
metadata:

....
```

```
labels:
  clusterresourceoverrides.admission.autoscaling.openshift.io/enabled: "true"

....
```

The Operator watches for the **ClusterResourceOverride** CR and ensures that the **ClusterResourceOverride** admission webhook is installed into the same namespace as the operator.

### 7.5.2.1. Installing the Cluster Resource Override Operator using the web console

You can use the OpenShift Container Platform web console to install the Cluster Resource Override Operator to help control overcommit in your cluster.

#### Prerequisites

- The Cluster Resource Override Operator has no effect if limits have not been set on containers. You must specify default limits for a project using a **LimitRange** object or configure limits in **Pod** specs for the overrides to apply.

#### Procedure

To install the Cluster Resource Override Operator using the OpenShift Container Platform web console:

1. In the OpenShift Container Platform web console, navigate to **Home → Projects**

   a. Click **Create Project**.

   b. Specify **clusterresourceoverride-operator** as the name of the project.

   c. Click **Create**.

2. Navigate to **Operators → OperatorHub**.

   a. Choose **ClusterResourceOverride Operator** from the list of available Operators and click **Install**.

   b. On the **Install Operator** page, make sure **A specific Namespace on the cluster** is selected for **Installation Mode**.

   c. Make sure **clusterresourceoverride-operator** is selected for **Installed Namespace**.

   d. Select an **Update Channel** and **Approval Strategy**.

   e. Click **Install**.

3. On the **Installed Operators** page, click **ClusterResourceOverride**.

   a. On the **ClusterResourceOverride Operator** details page, click **Create Instance**.

   b. On the **Create ClusterResourceOverride** page, edit the YAML template to set the overcommit values as needed:

      ```
      apiVersion: operator.autoscaling.openshift.io/v1
      kind: ClusterResourceOverride
      metadata:
        name: cluster ❶
      ```

```
spec:
  podResourceOverride:
    spec:
      memoryRequestToLimitPercent: 50  2
      cpuRequestToLimitPercent: 25  3
      limitCPUToMemoryPercent: 200  4
```

**1**   The name must be **cluster**.

**2**   Optional. Specify the percentage to override the container memory limit, if used, between 1-100. The default is 50.

**3**   Optional. Specify the percentage to override the container CPU limit, if used, between 1-100. The default is 25.

**4**   Optional. Specify the percentage to override the container memory limit, if used. Scaling 1Gi of RAM at 100 percent is equal to 1 CPU core. This is processed prior to overriding the CPU request, if configured. The default is 200.

   c.   Click **Create**.

4.   Check the current state of the admission webhook by checking the status of the cluster custom resource:

   a.   On the **ClusterResourceOverride Operator** page, click **cluster**.

   b.   On the **ClusterResourceOverride Details** page, click **YAML**. The **mutatingWebhookConfigurationRef** section appears when the webhook is called.

```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"operator.autoscaling.openshift.io/v1","kind":"ClusterResourceOverride","met
adata":{"annotations":{},"name":"cluster"},"spec":{"podResourceOverride":{"spec":
{"cpuRequestToLimitPercent":25,"limitCPUToMemoryPercent":200,"memoryRequestToLi
mitPercent":50}}}}
  creationTimestamp: "2019-12-18T22:35:02Z"
  generation: 1
  name: cluster
  resourceVersion: "127622"
  selfLink: /apis/operator.autoscaling.openshift.io/v1/clusterresourceoverrides/cluster
  uid: 978fc959-1717-4bd1-97d0-ae00ee111e8d
spec:
  podResourceOverride:
    spec:
      cpuRequestToLimitPercent: 25
      limitCPUToMemoryPercent: 200
      memoryRequestToLimitPercent: 50
status:

....
```

```
    mutatingWebhookConfigurationRef: 1
      apiVersion: admissionregistration.k8s.io/v1beta1
      kind: MutatingWebhookConfiguration
      name: clusterresourceoverrides.admission.autoscaling.openshift.io
      resourceVersion: "127621"
      uid: 98b3b8ae-d5ce-462b-8ab5-a729ea8f38f3

....
```

**1**   Reference to the **ClusterResourceOverride** admission webhook.

### 7.5.2.2. Installing the Cluster Resource Override Operator using the CLI

You can use the OpenShift Container Platform CLI to install the Cluster Resource Override Operator to help control overcommit in your cluster.

#### Prerequisites

- The Cluster Resource Override Operator has no effect if limits have not been set on containers. You must specify default limits for a project using a **LimitRange** object or configure limits in **Pod** specs for the overrides to apply.

#### Procedure

To install the Cluster Resource Override Operator using the CLI:

1. Create a namespace for the Cluster Resource Override Operator:

   a. Create a **Namespace** object YAML file (for example, **cro-namespace.yaml**) for the Cluster Resource Override Operator:

      ```
      apiVersion: v1
      kind: Namespace
      metadata:
        name: clusterresourceoverride-operator
      ```

   b. Create the namespace:

      ```
      $ oc create -f <file-name>.yaml
      ```

      For example:

      ```
      $ oc create -f cro-namespace.yaml
      ```

2. Create an Operator group:

   a. Create an **OperatorGroup** object YAML file (for example, cro-og.yaml) for the Cluster Resource Override Operator:

      ```
      apiVersion: operators.coreos.com/v1
      kind: OperatorGroup
      metadata:
        name: clusterresourceoverride-operator
        namespace: clusterresourceoverride-operator
      ```

```
spec:
  targetNamespaces:
    - clusterresourceoverride-operator
```

b. Create the Operator Group:

```
$ oc create -f <file-name>.yaml
```

For example:

```
$ oc create -f cro-og.yaml
```

3. Create a subscription:

a. Create a **Subscription** object YAML file (for example, cro-sub.yaml) for the Cluster Resource Override Operator:

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: clusterresourceoverride
  namespace: clusterresourceoverride-operator
spec:
  channel: "4.8"
  name: clusterresourceoverride
  source: redhat-operators
  sourceNamespace: openshift-marketplace
```

b. Create the subscription:

```
$ oc create -f <file-name>.yaml
```

For example:

```
$ oc create -f cro-sub.yaml
```

4. Create a **ClusterResourceOverride** custom resource (CR) object in the **clusterresourceoverride-operator** namespace:

a. Change to the **clusterresourceoverride-operator** namespace.

```
$ oc project clusterresourceoverride-operator
```

b. Create a **ClusterResourceOverride** object YAML file (for example, cro-cr.yaml) for the Cluster Resource Override Operator:

```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
  name: cluster 1
spec:
  podResourceOverride:
    spec:
```

```
        memoryRequestToLimitPercent: 50 2
        cpuRequestToLimitPercent: 25 3
        limitCPUToMemoryPercent: 200 4
```

**1** The name must be **cluster**.

**2** Optional. Specify the percentage to override the container memory limit, if used, between 1–100. The default is 50.

**3** Optional. Specify the percentage to override the container CPU limit, if used, between 1–100. The default is 25.

**4** Optional. Specify the percentage to override the container memory limit, if used. Scaling 1Gi of RAM at 100 percent is equal to 1 CPU core. This is processed prior to overriding the CPU request, if configured. The default is 200.

c. Create the **ClusterResourceOverride** object:

```
$ oc create -f <file-name>.yaml
```

For example:

```
$ oc create -f cro-cr.yaml
```

5. Verify the current state of the admission webhook by checking the status of the cluster custom resource.

```
$ oc get clusterresourceoverride cluster -n clusterresourceoverride-operator -o yaml
```

The **mutatingWebhookConfigurationRef** section appears when the webhook is called.

**Example output**

```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"operator.autoscaling.openshift.io/v1","kind":"ClusterResourceOverride","metadata":{"annotations":{},"name":"cluster"},"spec":{"podResourceOverride":{"spec":{"cpuRequestToLimitPercent":25,"limitCPUToMemoryPercent":200,"memoryRequestToLimitPercent":50}}}}
  creationTimestamp: "2019-12-18T22:35:02Z"
  generation: 1
  name: cluster
  resourceVersion: "127622"
  selfLink: /apis/operator.autoscaling.openshift.io/v1/clusterresourceoverrides/cluster
  uid: 978fc959-1717-4bd1-97d0-ae00ee111e8d
spec:
  podResourceOverride:
    spec:
      cpuRequestToLimitPercent: 25
      limitCPUToMemoryPercent: 200
```

```
      memoryRequestToLimitPercent: 50
status:

....

  mutatingWebhookConfigurationRef: ❶
    apiVersion: admissionregistration.k8s.io/v1beta1
    kind: MutatingWebhookConfiguration
    name: clusterresourceoverrides.admission.autoscaling.openshift.io
    resourceVersion: "127621"
    uid: 98b3b8ae-d5ce-462b-8ab5-a729ea8f38f3

....
```

❶ Reference to the **ClusterResourceOverride** admission webhook.

### 7.5.2.3. Configuring cluster-level overcommit

The Cluster Resource Override Operator requires a **ClusterResourceOverride** custom resource (CR) and a label for each project where you want the Operator to control overcommit.

#### Prerequisites

- The Cluster Resource Override Operator has no effect if limits have not been set on containers. You must specify default limits for a project using a **LimitRange** object or configure limits in **Pod** specs for the overrides to apply.

#### Procedure

To modify cluster-level overcommit:

1. Edit the **ClusterResourceOverride** CR:

   ```
   apiVersion: operator.autoscaling.openshift.io/v1
   kind: ClusterResourceOverride
   metadata:
       name: cluster
   spec:
     podResourceOverride:
      spec:
        memoryRequestToLimitPercent: 50 ❶
        cpuRequestToLimitPercent: 25 ❷
        limitCPUToMemoryPercent: 200 ❸
   ```

   ❶ Optional. Specify the percentage to override the container memory limit, if used, between 1-100. The default is 50.

   ❷ Optional. Specify the percentage to override the container CPU limit, if used, between 1-100. The default is 25.

   ❸ Optional. Specify the percentage to override the container memory limit, if used. Scaling 1Gi of RAM at 100 percent is equal to 1 CPU core. This is processed prior to overriding the CPU request, if configured. The default is 200.

2. Ensure the following label has been added to the Namespace object for each project where you want the Cluster Resource Override Operator to control overcommit:

```
apiVersion: v1
kind: Namespace
metadata:

...

 labels:
   clusterresourceoverrides.admission.autoscaling.openshift.io/enabled: "true" 1

...
```

**1** Add this label to each project.

## 7.5.3. Node-level overcommit

You can use various ways to control overcommit on specific nodes, such as quality of service (QOS) guarantees, CPU limits, or reserve resources. You can also disable overcommit for specific nodes and specific projects.

### 7.5.3.1. Understanding compute resources and containers

The node-enforced behavior for compute resources is specific to the resource type.

#### 7.5.3.1.1. Understanding container CPU requests

A container is guaranteed the amount of CPU it requests and is additionally able to consume excess CPU available on the node, up to any limit specified by the container. If multiple containers are attempting to use excess CPU, CPU time is distributed based on the amount of CPU requested by each container.

For example, if one container requested 500m of CPU time and another container requested 250m of CPU time, then any extra CPU time available on the node is distributed among the containers in a 2:1 ratio. If a container specified a limit, it will be throttled not to use more CPU than the specified limit. CPU requests are enforced using the CFS shares support in the Linux kernel. By default, CPU limits are enforced using the CFS quota support in the Linux kernel over a 100ms measuring interval, though this can be disabled.

#### 7.5.3.1.2. Understanding container memory requests

A container is guaranteed the amount of memory it requests. A container can use more memory than requested, but once it exceeds its requested amount, it could be terminated in a low memory situation on the node. If a container uses less memory than requested, it will not be terminated unless system tasks or daemons need more memory than was accounted for in the node's resource reservation. If a container specifies a limit on memory, it is immediately terminated if it exceeds the limit amount.

### 7.5.3.2. Understanding overcomitment and quality of service classes

A node is *overcommitted* when it has a pod scheduled that makes no request, or when the sum of limits across all pods on that node exceeds available machine capacity.

In an overcommitted environment, it is possible that the pods on the node will attempt to use more

compute resource than is available at any given point in time. When this occurs, the node must give priority to one pod over another. The facility used to make this decision is referred to as a Quality of Service (QoS) Class.

A pod is designated as one of three QoS classes with decreasing order of priority:

Table 7.19. Quality of Service Classes

| Priority | Class Name | Description |
| --- | --- | --- |
| 1 (highest) | Guaranteed | If limits and optionally requests are set (not equal to 0) for all resources and they are equal, then the pod is classified as **Guaranteed**. |
| 2 | Burstable | If requests and optionally limits are set (not equal to 0) for all resources, and they are not equal, then the pod is classified as **Burstable**. |
| 3 (lowest) | BestEffort | If requests and limits are not set for any of the resources, then the pod is classified as **BestEffort**. |

Memory is an incompressible resource, so in low memory situations, containers that have the lowest priority are terminated first:

- **Guaranteed** containers are considered top priority, and are guaranteed to only be terminated if they exceed their limits, or if the system is under memory pressure and there are no lower priority containers that can be evicted.

- **Burstable** containers under system memory pressure are more likely to be terminated once they exceed their requests and no other **BestEffort** containers exist.

- **BestEffort** containers are treated with the lowest priority. Processes in these containers are first to be terminated if the system runs out of memory.

### 7.5.3.2.1. Understanding how to reserve memory across quality of service tiers

You can use the **qos-reserved** parameter to specify a percentage of memory to be reserved by a pod in a particular QoS level. This feature attempts to reserve requested resources to exclude pods from lower OoS classes from using resources requested by pods in higher QoS classes.

OpenShift Container Platform uses the **qos-reserved** parameter as follows:

- A value of **qos-reserved=memory=100%** will prevent the **Burstable** and **BestEffort** QoS classes from consuming memory that was requested by a higher QoS class. This increases the risk of inducing OOM on **BestEffort** and **Burstable** workloads in favor of increasing memory resource guarantees for **Guaranteed** and **Burstable** workloads.

- A value of **qos-reserved=memory=50%** will allow the **Burstable** and **BestEffort** QoS classes to consume half of the memory requested by a higher QoS class.

- A value of **qos-reserved=memory=0%** will allow a **Burstable** and **BestEffort** QoS classes to consume up to the full node allocatable amount if available, but increases the risk that a **Guaranteed** workload will not have access to requested memory. This condition effectively disables this feature.

### 7.5.3.3. Understanding swap memory and QOS

You can disable swap by default on your nodes to preserve quality of service (QOS) guarantees. Otherwise, physical resources on a node can oversubscribe, affecting the resource guarantees the Kubernetes scheduler makes during pod placement.

For example, if two guaranteed pods have reached their memory limit, each container could start using swap memory. Eventually, if there is not enough swap space, processes in the pods can be terminated due to the system being oversubscribed.

Failing to disable swap results in nodes not recognizing that they are experiencing **MemoryPressure**, resulting in pods not receiving the memory they made in their scheduling request. As a result, additional pods are placed on the node to further increase memory pressure, ultimately increasing your risk of experiencing a system out of memory (OOM) event.

> **IMPORTANT**
>
> If swap is enabled, any out-of-resource handling eviction thresholds for available memory will not work as expected. Take advantage of out-of-resource handling to allow pods to be evicted from a node when it is under memory pressure, and rescheduled on an alternative node that has no such pressure.

### 7.5.3.4. Understanding nodes overcommitment

In an overcommitted environment, it is important to properly configure your node to provide best system behavior.

When the node starts, it ensures that the kernel tunable flags for memory management are set properly. The kernel should never fail memory allocations unless it runs out of physical memory.

To ensure this behavior, OpenShift Container Platform configures the kernel to always overcommit memory by setting the **vm.overcommit_memory** parameter to **1**, overriding the default operating system setting.

OpenShift Container Platform also configures the kernel not to panic when it runs out of memory by setting the **vm.panic_on_oom** parameter to **0**. A setting of 0 instructs the kernel to call oom_killer in an Out of Memory (OOM) condition, which kills processes based on priority

You can view the current setting by running the following commands on your nodes:

```
$ sysctl -a |grep commit
```

**Example output**

```
vm.overcommit_memory = 1
```

```
$ sysctl -a |grep panic
```

**Example output**

```
vm.panic_on_oom = 0
```

> **NOTE**
>
> The above flags should already be set on nodes, and no further action is required.

You can also perform the following configurations for each node:

- Disable or enforce CPU limits using CPU CFS quotas

- Reserve resources for system processes

- Reserve memory across quality of service tiers

### 7.5.3.5. Disabling or enforcing CPU limits using CPU CFS quotas

Nodes by default enforce specified CPU limits using the Completely Fair Scheduler (CFS) quota support in the Linux kernel.

If you disable CPU limit enforcement, it is important to understand the impact on your node:

- If a container has a CPU request, the request continues to be enforced by CFS shares in the Linux kernel.

- If a container does not have a CPU request, but does have a CPU limit, the CPU request defaults to the specified CPU limit, and is enforced by CFS shares in the Linux kernel.

- If a container has both a CPU request and limit, the CPU request is enforced by CFS shares in the Linux kernel, and the CPU limit has no impact on the node.

**Prerequisites**

1. Obtain the label associated with the static **MachineConfigPool** CRD for the type of node you want to configure by entering the following command:

   ```
   $ oc edit machineconfigpool <name>
   ```

   For example:

   ```
   $ oc edit machineconfigpool worker
   ```

   **Example output**

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: MachineConfigPool
   metadata:
     creationTimestamp: "2022-11-16T15:34:25Z"
     generation: 4
     labels:
       pools.operator.machineconfiguration.openshift.io/worker: ""    ❶
     name: worker
   ```

   ❶     The label appears under Labels.

   **TIP**

   If the label is not present, add a key/value pair such as:

   ```
   $ oc label machineconfigpool worker custom-kubelet=small-pods
   ```

**Procedure**

1. Create a custom resource (CR) for your configuration change.

   **Sample configuration for a disabling CPU limits**

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: KubeletConfig
   metadata:
     name: disable-cpu-units ❶
   spec:
     machineConfigPoolSelector:
       matchLabels:
         pools.operator.machineconfiguration.openshift.io/worker: "" ❷
     kubeletConfig:
       cpuCfsQuota: ❸
         - "false"
   ```

   ❶    Assign a name to CR.

   ❷    Specify the label from the machine config pool.

   ❸    Set the **cpuCfsQuota** parameter to **false**.

2. Run the following command to create the CR:

   ```
   $ oc create -f <file_name>.yaml
   ```

### 7.5.3.6. Reserving resources for system processes

To provide more reliable scheduling and minimize node resource overcommitment, each node can reserve a portion of its resources for use by system daemons that are required to run on your node for your cluster to function. In particular, it is recommended that you reserve resources for incompressible resources such as memory.

**Procedure**

To explicitly reserve resources for non-pod processes, allocate node resources by specifying resources available for scheduling. For more details, see Allocating Resources for Nodes.

### 7.5.3.7. Disabling overcommitment for a node

When enabled, overcommitment can be disabled on each node.

**Procedure**

To disable overcommitment in a node run the following command on that node:

```
$ sysctl -w vm.overcommit_memory=0
```

### 7.5.4. Project-level limits

To help control overcommit, you can set per-project resource limit ranges, specifying memory and CPU limits and defaults for a project that overcommit cannot exceed.

For information on project-level resource limits, see Additional resources.

Alternatively, you can disable overcommitment for specific projects.

### 7.5.4.1. Disabling overcommitment for a project

When enabled, overcommitment can be disabled per-project. For example, you can allow infrastructure components to be configured independently of overcommitment.

#### Procedure

To disable overcommitment in a project:

1. Edit the project object file

2. Add the following annotation:

   ```
   quota.openshift.io/cluster-resource-override-enabled: "false"
   ```

3. Create the project object:

   ```
   $ oc create -f <file-name>.yaml
   ```

### 7.5.5. Additional resources

- [Setting deployment resources](#).

- [Allocating resources for nodes](#).

## 7.6. ENABLING FEATURES USING FEATURE GATES

As an administrator, you can use feature gates to enable features that are not part of the default set of features.

### 7.6.1. Understanding feature gates

You can use the **FeatureGate** custom resource (CR) to enable specific feature sets in your cluster. A feature set is a collection of OpenShift Container Platform features that are not enabled by default.

You can activate any of the following feature sets by using the **FeatureGate** CR:

- **TechPreviewNoUpgrade**. This feature set is a subset of the current Technology Preview features. This feature set allows you to enable these tech preview features on test clusters, where you can fully test them, while leaving the features disabled on production clusters. Enabling this feature set cannot be undone and prevents minor version updates. This feature set is not recommended on production clusters.

> **WARNING**
>
> Enabling the **TechPreviewNoUpgrade** feature set on your cluster cannot be undone and prevents minor version updates. You should not enable this feature set on production clusters.

The following Technology Preview features are enabled by this feature set:

- Azure Disk CSI Driver Operator. Enables the provisioning of persistent volumes (PVs) by using the Container Storage Interface (CSI) driver for Microsoft Azure Disk Storage.

- VMware vSphere CSI Driver Operator. Enables the provisioning of persistent volumes (PVs) by using the Container Storage Interface (CSI) VMware vSphere driver for Virtual Machine Disk (VMDK) volumes.

- CSI automatic migration. Enables the automatic migration of supported in-tree volume plugins to their equivalent Container Storage Interface (CSI) drivers. Available as a Technology Preview for:

  - Amazon Web Services (AWS) Elastic Block Storage (EBS)

  - OpenStack Cinder

**Additional resources**

- For more information about the features activated by the **TechPreviewNoUpgrade** feature gate, see the following topics:

  - [Azure Disk CSI Driver Operator](#)

  - [VMware vSphere CSI Driver Operator](#)

  - [CSI automatic migration](#)

## 7.6.2. Enabling feature sets using the web console

You can use the OpenShift Container Platform web console to enable feature sets for all of the nodes in a cluster by editing the **FeatureGate** custom resource (CR).

**Procedure**

To enable feature sets:

1. In the OpenShift Container Platform web console, switch to the **Administration → Custom Resource Definitions** page.

2. On the **Custom Resource Definitions** page, click **FeatureGate**.

3. On the **Custom Resource Definition Details** page, click the **Instances** tab.

4. Click the **cluster** feature gate, then click the **YAML** tab.

5. Edit the **cluster** instance to add specific feature sets:

> **WARNING**
>
> Enabling the **TechPreviewNoUpgrade** feature set on your cluster cannot be undone and prevents minor version updates. You should not enable this feature set on production clusters.

**Sample Feature Gate custom resource**

```
apiVersion: config.openshift.io/v1
kind: FeatureGate
metadata:
  name: cluster 1
....

spec:
  featureSet: TechPreviewNoUpgrade 2
```

**1**    The name of the **FeatureGate** CR must be **cluster**.

**2**    Add the feature set that you want to enable:

- **TechPreviewNoUpgrade** enables specific Technology Preview features.

After you save the changes, new machine configs are created, the machine config pools are updated, and scheduling on each node is disabled while the change is being applied.

**Verification**

You can verify that the feature gates are enabled by looking at the **kubelet.conf** file on a node after the nodes return to the ready state.

1. From the **Administrator** perspective in the web console, navigate to **Compute → Nodes**.

2. Select a node.

3. In the **Node details** page, click **Terminal**.

4. In the terminal window, change your root directory to **/host**:
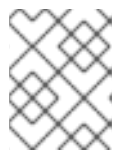
```
sh-4.2# chroot /host
```

5. View the **kubelet.conf** file:

```
sh-4.2# cat /etc/kubernetes/kubelet.conf
```

**Sample output**

```
...
featureGates:
  InsightsOperatorPullingSCA: true,
  LegacyNodeRoleBehavior: false
...
```

The features that are listed as **true** are enabled on your cluster.

> **NOTE**
>
> The features listed vary depending upon the OpenShift Container Platform version.

### 7.6.3. Enabling feature sets using the CLI

You can use the OpenShift CLI (**oc**) to enable feature sets for all of the nodes in a cluster by editing the **FeatureGate** custom resource (CR).

**Prerequisites**

- You have installed the OpenShift CLI (**oc**).

**Procedure**

To enable feature sets:

1. Edit the **FeatureGate** CR named **cluster**:

   ```
   $ oc edit featuregate cluster
   ```

   > **WARNING**
   >
   > Enabling the **TechPreviewNoUpgrade** feature set on your cluster cannot be undone and prevents minor version updates. You should not enable this feature set on production clusters.

   **Sample FeatureGate custom resource**

   ```
   apiVersion: config.openshift.io/v1
   kind: FeatureGate
   metadata:
     name: cluster ❶
   spec:
     featureSet: TechPreviewNoUpgrade ❷
   ```

   ❶ The name of the **FeatureGate** CR must be **cluster**.

   ❷ Add the feature set that you want to enable:

- **TechPreviewNoUpgrade** enables specific Technology Preview features.

After you save the changes, new machine configs are created, the machine config pools are updated, and scheduling on each node is disabled while the change is being applied.

## Verification

You can verify that the feature gates are enabled by looking at the **kubelet.conf** file on a node after the nodes return to the ready state.

1. From the **Administrator** perspective in the web console, navigate to  **Compute → Nodes**.

2. Select a node.

3. In the **Node details** page, click **Terminal**.

4. In the terminal window, change your root directory to **/host**:

   ```
   sh-4.2# chroot /host
   ```
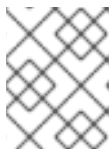
5. View the **kubelet.conf** file:

   ```
   sh-4.2# cat /etc/kubernetes/kubelet.conf
   ```

   **Sample output**

   ```
   ...
   featureGates:
     InsightsOperatorPullingSCA: true,
     LegacyNodeRoleBehavior: false
   ...
   ```

   The features that are listed as **true** are enabled on your cluster.

   > **NOTE**
   >
   > The features listed vary depending upon the OpenShift Container Platform version.

# CHAPTER 8. REMOTE WORKER NODES ON THE NETWORK EDGE

## 8.1. USING REMOTE WORKER NODES AT THE NETWORK EDGE

You can configure OpenShift Container Platform clusters with nodes located at your network edge. In this topic, they are called *remote worker nodes*. A typical cluster with remote worker nodes combines on-premise master and worker nodes with worker nodes in other locations that connect to the cluster. This topic is intended to provide guidance on best practices for using remote worker nodes and does not contain specific configuration details.

There are multiple use cases across different industries, such as telecommunications, retail, manufacturing, and government, for using a deployment pattern with remote worker nodes. For example, you can separate and isolate your projects and workloads by combining the remote worker nodes into Kubernetes zones.

However, having remote worker nodes can introduce higher latency, intermittent loss of network connectivity, and other issues. Among the challenges in a cluster with remote worker node are:

- **Network separation**: The OpenShift Container Platform control plane and the remote worker nodes must be able communicate with each other. Because of the distance between the control plane and the remote worker nodes, network issues could prevent this communication. See Network separation with remote worker nodes for information on how OpenShift Container Platform responds to network separation and for methods to diminish the impact to your cluster.

- **Power outage**: Because the control plane and remote worker nodes are in separate locations, a power outage at the remote location or at any point between the two can negatively impact your cluster. See Power loss on remote worker nodes for information on how OpenShift Container Platform responds to a node losing power and for methods to diminish the impact to your cluster.

- **Latency spikes or temporary reduction in throughput** As with any network, any changes in network conditions between your cluster and the remote worker nodes can negatively impact your cluster. These types of situations are beyond the scope of this documentation.

Note the following limitations when planning a cluster with remote worker nodes:

- Remote worker nodes are supported on only bare metal clusters with user-provisioned infrastructure.

- OpenShift Container Platform does not support remote worker nodes that use a different cloud provider than the on-premise cluster uses.

- Moving workloads from one Kubernetes zone to a different Kubernetes zone can be problematic due to system and environment issues, such as a specific type of memory not being available in a different zone.

- Proxies and firewalls can present additional limitations that are beyond the scope of this document. See the relevant OpenShift Container Platform documentation for how to address such limitations, such as Configuring your firewall.

- You are responsible for configuring and maintaining L2/L3-level network connectivity between the control plane and the network-edge nodes.

## 8.1.1. Network separation with remote worker nodes

All nodes send heartbeats to the Kubernetes Controller Manager Operator (kube controller) in the OpenShift Container Platform cluster every 10 seconds. If the cluster does not receive heartbeats from a node, OpenShift Container Platform responds using several default mechanisms.

OpenShift Container Platform is designed to be resilient to network partitions and other disruptions. You can mitigate some of the more common disruptions, such as interruptions from software upgrades, network splits, and routing issues. Mitigation strategies include ensuring that pods on remote worker nodes request the correct amount of CPU and memory resources, configuring an appropriate replication policy, using redundancy across zones, and using Pod Disruption Budgets on workloads.

If the kube controller loses contact with a node after a configured period, the node controller on the control plane updates the node health to **Unhealthy** and marks the node **Ready** condition as **Unknown**. In response, the scheduler stops scheduling pods to that node. The on-premise node controller adds a **node.kubernetes.io/unreachable** taint with a **NoExecute** effect to the node and schedules pods on the node for eviction after five minutes, by default.

If a workload controller, such as a **Deployment** object or **StatefulSet** object, is directing traffic to pods on the unhealthy node and other nodes can reach the cluster, OpenShift Container Platform routes the traffic away from the pods on the node. Nodes that cannot reach the cluster do not get updated with the new traffic routing. As a result, the workloads on those nodes might continue to attempt to reach the unhealthy node.

You can mitigate the effects of connection loss by:

- using daemon sets to create pods that tolerate the taints

- using static pods that automatically restart if a node goes down

- using Kubernetes zones to control pod eviction

- configuring pod tolerations to delay or avoid pod eviction

- configuring the kubelet to control the timing of when it marks nodes as unhealthy.

For more information on using these objects in a cluster with remote worker nodes, see About remote worker node strategies.

## 8.1.2. Power loss on remote worker nodes

If a remote worker node loses power or restarts ungracefully, OpenShift Container Platform responds using several default mechanisms.

If the Kubernetes Controller Manager Operator (kube controller) loses contact with a node after a configured period, the control plane updates the node health to **Unhealthy** and marks the node **Ready** condition as **Unknown**. In response, the scheduler stops scheduling pods to that node. The on-premise node controller adds a **node.kubernetes.io/unreachable** taint with a **NoExecute** effect to the node and schedules pods on the node for eviction after five minutes, by default.

On the node, the pods must be restarted when the node recovers power and reconnects with the control plane.

> **NOTE**
>
> If you want the pods to restart immediately upon restart, use static pods.

After the node restarts, the kubelet also restarts and attempts to restart the pods that were scheduled on the node. If the connection to the control plane takes longer than the default five minutes, the control plane cannot update the node health and remove the **node.kubernetes.io/unreachable** taint. On the node, the kubelet terminates any running pods. When these conditions are cleared, the scheduler can start scheduling pods to that node.

You can mitigate the effects of power loss by:

- using daemon sets to create pods that tolerate the taints

- using static pods that automatically restart with a node

- configuring pods tolerations to delay or avoid pod eviction

- configuring the kubelet to control the timing of when the node controller marks nodes as unhealthy.

For more information on using these objects in a cluster with remote worker nodes, see About remote worker node strategies.

### 8.1.3. Remote worker node strategies

If you use remote worker nodes, consider which objects to use to run your applications.

It is recommend to use daemon sets or static pods based on the behavior you want in the event of network issues or power loss. In addition, you can use Kubernetes zones and tolerations to control or avoid pod evictions if the control plane cannot reach remote worker nodes.

**Daemon sets**

Daemon sets are the best approach to managing pods on remote worker nodes for the following reasons:

- Daemon sets do not typically need rescheduling behavior. If a node disconnects from the cluster, pods on the node can continue to run. OpenShift Container Platform does not change the state of daemon set pods, and leaves the pods in the state they last reported. For example, if a daemon set pod is in the **Running** state, when a node stops communicating, the pod keeps running and is assumed to be running by OpenShift Container Platform.

- Daemon set pods, by default, are created with **NoExecute** tolerations for the **node.kubernetes.io/unreachable** and **node.kubernetes.io/not-ready** taints with no **tolerationSeconds** value. These default values ensure that daemon set pods are never evicted if the control plane cannot reach a node. For example:

  **Tolerations added to daemon set pods by default**

  ```
  tolerations:
    - key: node.kubernetes.io/not-ready
      operator: Exists
      effect: NoExecute
    - key: node.kubernetes.io/unreachable
      operator: Exists
      effect: NoExecute
    - key: node.kubernetes.io/disk-pressure
      operator: Exists
      effect: NoSchedule
    - key: node.kubernetes.io/memory-pressure
  ```

```
      operator: Exists
      effect: NoSchedule
    - key: node.kubernetes.io/pid-pressure
      operator: Exists
      effect: NoSchedule
    - key: node.kubernetes.io/unschedulable
      operator: Exists
      effect: NoSchedule
```

- Daemon sets can use labels to ensure that a workload runs on a matching worker node.

- You can use an OpenShift Container Platform service endpoint to load balance daemon set pods.

> **NOTE**
>
> Daemon sets do not schedule pods after a reboot of the node if OpenShift Container Platform cannot reach the node.

**Static pods**

If you want pods restart if a node reboots, after a power loss for example, consider static pods. The kubelet on a node automatically restarts static pods as node restarts.

> **NOTE**
>
> Static pods cannot use secrets and config maps.

**Kubernetes zones**

Kubernetes zones can slow down the rate or, in some cases, completely stop pod evictions.

When the control plane cannot reach a node, the node controller, by default, applies **node.kubernetes.io/unreachable** taints and evicts pods at a rate of 0.1 nodes per second. However, in a cluster that uses Kubernetes zones, pod eviction behavior is altered.

If a zone is fully disrupted, where all nodes in the zone have a **Ready** condition that is **False** or **Unknown**, the control plane does not apply the **node.kubernetes.io/unreachable** taint to the nodes in that zone.

For partially disrupted zones, where more than 55% of the nodes have a **False** or **Unknown** condition, the pod eviction rate is reduced to 0.01 nodes per second. Nodes in smaller clusters, with fewer than 50 nodes, are not tainted. Your cluster must have more than three zones for these behavior to take effect.

You assign a node to a specific zone by applying the **topology.kubernetes.io/region** label in the node specification.

**Sample node labels for Kubernetes zones**

```
kind: Node
apiVersion: v1
metadata:
  labels:
    topology.kubernetes.io/region=east
```

**KubeletConfig** objects

You can adjust the amount of time that the kubelet checks the state of each node.

To set the interval that affects the timing of when the on-premise node controller marks nodes with the **Unhealthy** or **Unreachable** condition, create a **KubeletConfig** object that contains the **node-status-update-frequency** and **node-status-report-frequency** parameters.

The kubelet on each node determines the node status as defined by the **node-status-update-frequency** setting and reports that status to the cluster based on the **node-status-report-frequency** setting. By default, the kubelet determines the pod status every 10 seconds and reports the status every minute. However, if the node state changes, the kubelet reports the change to the cluster immediately. OpenShift Container Platform uses the **node-status-report-frequency** setting only when the Node Lease feature gate is enabled, which is the default state in OpenShift Container Platform clusters. If the Node Lease feature gate is disabled, the node reports its status based on the **node-status-update-frequency** setting.

**Example kubelet config**

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: disable-cpu-units
spec:
  machineConfigPoolSelector:
    matchLabels:
      machineconfiguration.openshift.io/role: worker 1
  kubeletConfig:
    node-status-update-frequency: 2
      - "10s"
    node-status-report-frequency: 3
      - "1m"
```

**[1]** Specify the type of node type to which this **KubeletConfig** object applies using the label from the **MachineConfig** object.

**[2]** Specify the frequency that the kubelet checks the status of a node associated with this **MachineConfig** object. The default value is **10s**. If you change this default, the **node-status-report-frequency** value is changed to the same value.

**[3]** Specify the frequency that the kubelet reports the status of a node associated with this **MachineConfig** object. The default value is **1m**.

The **node-status-update-frequency** parameter works with the **node-monitor-grace-period** and **pod-eviction-timeout** parameters.

- The **node-monitor-grace-period** parameter specifies how long OpenShift Container Platform waits after a node associated with a **MachineConfig** object is marked **Unhealthy** if the controller manager does not receive the node heartbeat. Workloads on the node continue to run after this time. If the remote worker node rejoins the cluster after **node-monitor-grace-period** expires, pods continue to run. New pods can be scheduled to that node. The **node-monitor-grace-period** interval is **40s**. The **node-status-update-frequency** value must be lower than the **node-monitor-grace-period** value.

- The **pod-eviction-timeout** parameter specifies the amount of time OpenShift Container Platform waits after marking a node that is associated with a **MachineConfig** object as **Unreachable** to start marking pods for eviction. Evicted pods are rescheduled on other nodes.

If the remote worker node rejoins the cluster after **pod-eviction-timeout** expires, the pods running on the remote worker node are terminated because the node controller has evicted the pods on-premise. Pods can then be rescheduled to that node. The **pod-eviction-timeout** interval is **5m0s**.

> **NOTE**
>
> Modifying the **node-monitor-grace-period** and **pod-eviction-timeout** parameters is not supported.

**Tolerations**

You can use pod tolerations to mitigate the effects if the on-premise node controller adds a **node.kubernetes.io/unreachable** taint with a **NoExecute** effect to a node it cannot reach.

A taint with the **NoExecute** effect affects pods that are running on the node in the following ways:

- Pods that do not tolerate the taint are queued for eviction.

- Pods that tolerate the taint without specifying a **tolerationSeconds** value in their toleration specification remain bound forever.

- Pods that tolerate the taint with a specified **tolerationSeconds** value remain bound for the specified amount of time. After the time elapses, the pods are queued for eviction.

You can delay or avoid pod eviction by configuring pods tolerations with the **NoExecute** effect for the **node.kubernetes.io/unreachable** and **node.kubernetes.io/not-ready** taints.

**Example toleration in a pod spec**

```
...
tolerations:
- key: "node.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute" 1
- key: "node.kubernetes.io/not-ready"
  operator: "Exists"
  effect: "NoExecute" 2
  tolerationSeconds: 600
...
```

**1** The **NoExecute** effect without **tolerationSeconds** lets pods remain forever if the control plane cannot reach the node.

**2** The **NoExecute** effect with **tolerationSeconds**: 600 lets pods remain for 10 minutes if the control plane marks the node as **Unhealthy**.

OpenShift Container Platform uses the **tolerationSeconds** value after the **pod-eviction-timeout** value elapses.

**Other types of OpenShift Container Platform objects**

You can use replica sets, deployments, and replication controllers. The scheduler can reschedule these pods onto other nodes after the node is disconnected for five minutes. Rescheduling onto other nodes can be beneficial for some workloads, such as REST APIs, where an administrator can guarantee a specific number of pods are running and accessible.

> **NOTE**
>
> When working with remote worker nodes, rescheduling pods on different nodes might not be acceptable if remote worker nodes are intended to be reserved for specific functions.

stateful sets do not get restarted when there is an outage. The pods remain in the **terminating** state until the control plane can acknowledge that the pods are terminated.

To avoid scheduling a to a node that does not have access to the same type of persistent storage, OpenShift Container Platform cannot migrate pods that require persistent volumes to other zones in the case of network separation.

### Additional resources

- For more information on Daemonesets, see DaemonSets.

- For more information on taints and tolerations, see Controlling pod placement using node taints.

- For more information on configuring **KubeletConfig** objects, see Creating a KubeletConfig CRD.

- For more information on replica sets, see ReplicaSets.

- For more information on deployments, see Deployments.

- For more information on replication controllers, see Replication controllers.

- For more information on the controller manager, see Kubernetes Controller Manager Operator.