Container Runtimes

Note: Dockershim has been removed from the Kubernetes project as of release 1.24. Read the Dockershim Removal FAQ for further details.

You need to install a container runtime into each node in the cluster so that Pods can run there. This page outlines what is involved and describes related tasks for setting up nodes.

Kubernetes 1.31 requires that you use a runtime that conforms with the Container Runtime Interface (CRI).

See CRI version support for more information.

This page provides an outline of how to use several common container runtimes with Kubernetes.

- containerd
- CRI-O
- Docker Engine
- Mirantis Container Runtime

Note:
Kubernetes releases before v1.24 included a direct integration with Docker Engine, using a component named *dockershim*. That special direct integration is no longer part of Kubernetes (this removal was announced as part of the v1.20 release). You can read Check whether Dockershim removal affects you to understand how this removal might affect you. To learn about migrating from using dockershim, see Migrating from dockershim.

If you are running a version of Kubernetes other than v1.31, check the documentation for that version.

Install and configure prerequisites

Network configuration

By default, the Linux kernel does not allow IPv4 packets to be routed between interfaces. Most Kubernetes cluster networking implementations will change this setting (if needed), but some might expect the administrator to do it for them. (Some might also expect other sysctl parameters to be set, kernel modules to be loaded, etc; consult the documentation for your specific network implementation.)

Enable IPv4 packet forwarding

To manually enable IPv4 packet forwarding:

```
# sysctl params required by setup, params persist across reboots
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.ipv4.ip_forward = 1
EOF

# Apply sysctl params without reboot
sudo sysctl --system
```

Verify that `net.ipv4.ip_forward` is set to 1 with:

```
sysctl net.ipv4.ip_forward
```

cgroup drivers

On Linux, control groups are used to constrain resources that are allocated to processes.

Both the kubelet and the underlying container runtime need to interface with control groups to enforce resource management for pods and containers and set resources such as cpu/memory requests and limits. To interface with control groups, the kubelet and the container runtime need to use a *cgroup driver*. It's critical that the kubelet and the container runtime use the same cgroup driver and are configured the same.

There are two cgroup drivers available:

- `cgroupfs`
- `systemd`

cgroupfs driver

The `cgroupfs` driver is the default cgroup driver in the kubelet. When the `cgroupfs` driver is used, the kubelet and the container runtime directly interface with the cgroup filesystem to configure cgroups.

The `cgroupfs` driver is **not** recommended when systemd is the init system because systemd expects a single cgroup manager on the system. Additionally, if you use cgroup v2, use the `systemd` cgroup driver instead of `cgroupfs` .

systemd cgroup driver

When systemd is chosen as the init system for a Linux distribution, the init process generates and consumes a root control group (`cgroup`) and acts as a cgroup manager.

systemd has a tight integration with cgroups and allocates a cgroup per systemd unit. As a result, if you use `systemd` as the init system with the `cgroupfs` driver, the system gets two different cgroup managers.

Two cgroup managers result in two views of the available and in-use resources in the system. In some cases, nodes that are configured to use `cgroupfs` for the kubelet and container runtime, but use `systemd` for the rest of the processes become unstable under resource pressure.

The approach to mitigate this instability is to use `systemd` as the cgroup driver for the kubelet and the container runtime when systemd is the selected init system.

To set `systemd` as the cgroup driver, edit the `KubeletConfiguration` option of `cgroupDriver` and set it to `systemd` . For example:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
...
cgroupDriver: systemd
```

Note:

Starting with v1.22 and later, when creating a cluster with kubeadm, if the user does not set the `cgroupDriver` field under `KubeletConfiguration`, kubeadm defaults it to `systemd`.

If you configure `systemd` as the cgroup driver for the kubelet, you must also configure `systemd` as the cgroup driver for the container runtime. Refer to the documentation for your container runtime for instructions. For example:

- containerd
- CRI-O

In Kubernetes 1.31, with the `KubeletCgroupDriverFromCRI` feature gate enabled and a container runtime that supports the `RuntimeConfig` CRI RPC, the kubelet automatically detects the appropriate cgroup driver from the runtime, and ignores the `cgroupDriver` setting within the kubelet configuration.

Caution:

Changing the cgroup driver of a Node that has joined a cluster is a sensitive operation. If the kubelet has created Pods using the semantics of one cgroup driver, changing the container runtime to another cgroup driver can cause errors when trying to re-create the Pod sandbox for such existing Pods. Restarting the kubelet may not solve such errors.

If you have automation that makes it feasible, replace the node with another using the updated configuration, or reinstall it using automation.

Migrating to the `systemd` driver in kubeadm managed clusters

If you wish to migrate to the `systemd` cgroup driver in existing kubeadm managed clusters, follow configuring a cgroup driver.

CRI version support

Your container runtime must support at least v1alpha2 of the container runtime interface.

Kubernetes starting v1.26 *only works* with v1 of the CRI API. Earlier versions default to v1 version, however if a container runtime does not support the v1 API, the kubelet falls back to using the (deprecated) v1alpha2 API instead.

Container runtimes

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

containerd

This section outlines the necessary steps to use containerd as CRI runtime.

To install containerd on your system, follow the instructions on getting started with containerd. Return to this step once you've created a valid `config.toml` configuration file.

Linux | Windows

You can find this file under the path `/etc/containerd/config.toml` .

On Linux the default CRI socket for containerd is `/run/containerd/containerd.sock` . On Windows the default CRI endpoint is `npipe://./pipe/containerd-containerd` .

Configuring the `systemd` cgroup driver

To use the `systemd` cgroup driver in `/etc/containerd/config.toml` with `runc` , set

```
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
  ...
  [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]
    SystemdCgroup = true
```

The `systemd` cgroup driver is recommended if you use cgroup v2.

Note:
If you installed containerd from a package (for example, RPM or `.deb`), you may find that the CRI integration plugin is disabled by default.

You need CRI support enabled to use containerd with Kubernetes. Make sure that `cri` is not included in the `disabled_plugins` list within `/etc/containerd/config.toml`; if you made changes to that file, also restart `containerd`.

If you experience container crash loops after the initial cluster installation or after installing a CNI, the containerd configuration provided with the package might contain incompatible configuration parameters. Consider resetting the containerd configuration with `containerd config default > /etc/containerd/config.toml` as specified in getting-started.md and then set the configuration parameters specified above accordingly.

If you apply this change, make sure to restart containerd:

```
sudo systemctl restart containerd
```

When using kubeadm, manually configure the cgroup driver for kubelet.

In Kubernetes v1.28, you can enable automatic detection of the cgroup driver as an alpha feature. See systemd cgroup driver for more details.

Overriding the sandbox (pause) image

In your containerd config you can overwrite the sandbox image by setting the following config:

```
[plugins."io.containerd.grpc.v1.cri"]
  sandbox_image = "registry.k8s.io/pause:3.2"
```

You might need to restart `containerd` as well once you've updated the config file: `systemctl restart containerd`.

CRI-O

This section contains the necessary steps to install CRI-O as a container runtime.

To install CRI-O, follow CRI-O Install Instructions.

cgroup driver

CRI-O uses the systemd cgroup driver per default, which is likely to work fine for you. To switch to the `cgroupfs` cgroup driver, either edit `/etc/crio/crio.conf` or place a drop-in configuration in `/etc/crio/crio.conf.d/02-cgroup-manager.conf`, for example:

```
[crio.runtime]
conmon_cgroup = "pod"
cgroup_manager = "cgroupfs"
```

You should also note the changed `conmon_cgroup`, which has to be set to the value `pod` when using CRI-O with `cgroupfs`. It is generally necessary to keep the cgroup driver configuration of the kubelet (usually done via kubeadm) and CRI-O in sync.

In Kubernetes v1.28, you can enable automatic detection of the cgroup driver as an alpha feature. See systemd cgroup driver for more details.

For CRI-O, the CRI socket is `/var/run/crio/crio.sock` by default.

Overriding the sandbox (pause) image

In your CRI-O config you can set the following config value:

```
[crio.image]
pause_image="registry.k8s.io/pause:3.6"
```

This config option supports live configuration reload to apply this change: `systemctl reload crio` or by sending `SIGHUP` to the `crio` process.

Docker Engine

Note:
These instructions assume that you are using the `cri-dockerd` adapter to integrate Docker Engine with Kubernetes.

1. On each of your nodes, install Docker for your Linux distribution as per Install Docker Engine.
2. Install `cri-dockerd`, following the directions in the install section of the documentation.

For `cri-dockerd`, the CRI socket is `/run/cri-dockerd.sock` by default.

Mirantis Container Runtime

Mirantis Container Runtime (MCR) is a commercially available container runtime that was formerly known as Docker Enterprise Edition.

You can use Mirantis Container Runtime with Kubernetes using the open source `cri-dockerd` component, included with MCR.

To learn more about how to install Mirantis Container Runtime, visit MCR Deployment Guide.

Check the systemd unit named `cri-docker.socket` to find out the path to the CRI socket.

Overriding the sandbox (pause) image

The `cri-dockerd` adapter accepts a command line argument for specifying which container image to use as the Pod infrastructure container (“pause image”). The command line argument to use is `--pod-infra-container-image`.

What's next

As well as a container runtime, your cluster will need a working network plugin.

Items on this page refer to third party products or projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for those third-party products or projects. See the CNCF website guidelines for more details.

You should read the content guide before proposing a change that adds an extra third-party link.

Feedback

Was this page helpful?

Yes No

Last modified August 30, 2024 at 12:05 AM PST: Remove reference of closed containerd issue (0909e130d6)