



OpenShift Container Platform 3.11

CRI-O Runtime

cri-o Runtime Guide

OpenShift Container Platform 3.11 CRI-O Runtime

cri-o Runtime Guide

Legal Notice

Copyright © 2021 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

Learn how to use cri-o

Table of Contents

CHAPTER 1. USING THE CRI-O CONTAINER ENGINE	3
1.1. UNDERSTANDING CRI-O	3
1.2. GETTING CRI-O	3
1.2.1. Installing CRI-O with a new OpenShift Container Platform cluster	4
1.2.2. Adding CRI-O nodes to an OpenShift Container Platform cluster	5
1.3. CONFIGURING CRI-O	6
1.3.1. Configuring CRI-O storage	7
1.3.2. Configuring CRI-O networking	8
1.4. TROUBLESHOOTING CRI-O	8
1.4.1. Checking CRI-O's general health	9
1.4.2. Inspecting CRI-O logs	9
1.4.2.1. Checking cri-o and origin-node logs	9
1.4.2.2. Turning on debugging for CRI-O	11
1.4.3. Troubleshooting CRI-O pods, and containers	11
1.4.3.1. Listing images, pods, and containers	12
1.4.3.2. Investigating images, pods, and containers	14
Additional resources	15

CHAPTER 1. USING THE CRI-O CONTAINER ENGINE

CRI-O is an open source, community-driven container engine. Its primary goal is to replace the Docker service as the container engine for Kubernetes implementations, such as OpenShift Container Platform.

If you want to start using CRI-O, this guide describes how to install CRI-O during OpenShift Container Platform installation as well as how to add a CRI-O node to an existing OpenShift Container Platform cluster. The guide also provides information on how to configure and troubleshoot your CRI-O engine.

1.1. UNDERSTANDING CRI-O

The [CRI-O](#) container engine provides a stable, more secure, and performant platform for running [Open Container Initiative](#) (OCI) compatible runtimes. You can use the CRI-O container engine to launch containers and pods by engaging OCI-compliant runtimes like `runc`, the default OCI runtime, or [Kata Containers](#). CRI-O's purpose is to be the container engine that implements the Kubernetes Container Runtime Interface (CRI) for OpenShift Container Platform and Kubernetes, replacing the Docker service.

CRI-O offers a streamlined container engine, while other container features are implemented as a separate set of innovative, independent commands. This approach allows container management features to develop at their own pace, without impeding CRI-O's primary goal of being a container engine for Kubernetes-based installations.

CRI-O's stability comes from the facts that it is developed, tested, and released in tandem with Kubernetes major and minor releases and that it follows OCI standards. For example, CRI-O 1.11 aligns with Kubernetes 1.11. The scope of CRI-O is tied to the [Container Runtime Interface](#) (CRI). CRI extracted and standardized exactly what a Kubernetes service (kubelet) needed from its container engine. The CRI team did this to help stabilize Kubernetes container engine requirements as multiple container engines began to be developed.

There is little need for direct command-line contact with CRI-O. However, to provide full access to CRI-O for testing and monitoring, and to provide features you expect with Docker that CRI-O does not offer, a set of container-related command-line tools are available. These tools replace and extend what is available with the **docker** command and service. Tools include:

- **crictl** - For troubleshooting and working directly with CRI-O container engines
- **runc** - For running container images
- **podman** - For managing pods and container images (run, stop, start, ps, attach, exec, etc.) outside of the container engine
- **buildah** - For building, pushing and signing container images
- **skopeo** - For copying, inspecting, deleting, and signing images

Some Docker features are included in other tools instead of in CRI-O. For example, **podman** offers exact command-line compatibility with many **docker** command features and extends those features to managing pods as well. No container engine is needed to run containers or pods with **podman**.

Features for building, pushing, and signing container images, which are also not required in a container engine, are available in the **buildah** command. For more information about these command alternatives to **docker**, see [Finding, Running and Building Containers without Docker](#).

1.2. GETTING CRI-O

CRI-O is not supported as a stand-alone container engine. You must use CRI-O as a container engine for a Kubernetes installation, such as OpenShift Container Platform. To run containers without Kubernetes or OpenShift Container Platform, use [podman](#).

To set up a CRI-O container engine to use with an OpenShift Container Platform cluster, you can:

- Install CRI-O along with a new OpenShift Container Platform cluster or
- Add a node to an existing cluster and identify CRI-O as the container engine for that node. Both CRI-O and Docker nodes can exist on the same cluster.

The following section describes how to install CRI-O with a new OpenShift Container Platform cluster

1.2.1. Installing CRI-O with a new OpenShift Container Platform cluster

You can choose CRI-O as the container engine for your OpenShift Container Platform nodes on a per-node basis at install time. Here are a few things you should know about enabling the CRI-O container engine when you install OpenShift Container Platform:

- Previously, using CRI-O on your nodes required that the Docker container engine be available as well. As of OpenShift Container Platform 3.10 and later, the Docker container engine is no longer required in all cases. Now you can now have CRI-O-only nodes in your OpenShift Container Platform cluster. However, nodes that do build and push operations still need to have the Docker container engine installed along with CRI-O.
- Enabling CRI-O using a CRI-O container is no longer supported. An rpm-based installation of CRI-O is required.

The following procedure assumes you are installing OpenShift Container Platform using Ansible inventory files, such as those described in [Configuring Your Inventory File](#).



NOTE

Do not set **`/var/lib/docker`** as a separate mount point for an OpenShift Container Platform node using CRI-O as its container engine. When deploying a CRI-O node, the installer tries to make **`/var/lib/docker`** a symbolic link to **`/var/lib/containers`**. That action will fail because it won't be able to remove the existing **`/var/lib/docker`** to create the symbolic link.

1. With the OpenShift Container Platform Ansible playbooks installed, edit the appropriate inventory file to enable CRI-O.
2. Locate CRI-O setting in your selected inventory file. To have the CRI-O container engine installed on your nodes during OpenShift Container Platform installation, locate the `[OSEv3:vars]` section of an Ansible inventory file. A section of CRI-O settings might include the following:

```
[OSEv3:vars]
...
# Install and run cri-o.
#openshift_use_crio=False
#openshift_use_crio_only=False
# The following two variables are used when openshift_use_crio is True
# and cleans up after builds that pass through docker. When openshift_use_crio is True
# these variables are set to the defaults shown. You may override them here.
```



```
# NOTE: You will still need to tag cri-o nodes with your given label(s)!
# Enable docker garbage collection when using cri-o
#openshift_crio_enable_docker_gc=True
# Node Selectors to run the garbage collection
#openshift_crio_docker_gc_node_selector={'runtime': 'cri-o'}
```

3. Enable CRI-O settings. You can decide to either enable CRI-O alone or CRI-O alongside Docker. The following settings allow CRI-O and Docker as your node container engines and enables Docker garbage collection on nodes with overlay2 storage:



NOTE

To be able to build containers on CRI-O nodes, you must have the Docker container engine installed. If you want to have CRI-O-only nodes, you can do that and simply designate other nodes to do container builds.

```
[OSEv3:vars]
...
openshift_use_crio=True
openshift_use_crio_only=False
openshift_crio_enable_docker_gc=True
```

4. Set the `openshift_node_group_name` for each node to a configmap that configures the kubelet for the CRI-O runtime. There's a corresponding CRI-O configmap for all the default node groups. [Defining Node Groups and Host Mappings](#) covers node groups and mappings in detail.

```
[nodes]
ocp-crio01 openshift_node_group_name='node-config-all-in-one-crio'
ocp-docker01 openshift_node_group_name='node-config-all-in-one'
```

This will automatically install the necessary CRI-O packages.

The resulting OpenShift Container Platform configuration will be running the CRI-O container engine on the nodes of your OpenShift Container Platform installation. Use the `oc` command to check the status of the nodes and identify the nodes running CRI-O:

```
$ oc get nodes -o wide
NAME      STATUS ROLES      AGE ... CONTAINER-RUNTIME
ocp-crio01 Ready  compute,infra,  16d ... cri-o://1.11.5
ocp-docker01 Ready  compute,infra,  16d ... docker://1.13.1
```

1.2.2. Adding CRI-O nodes to an OpenShift Container Platform cluster

OpenShift Container Platform does not support the direct upgrading of nodes from using the docker container engine to using CRI-O. To upgrade an existing OpenShift Container Platform cluster to use CRI-O, do the following:

- Scale up a node that is configured to use the CRI-O container engine
- Check that the CRI-O node performs as expected
- Add more CRI-O nodes as needed

- Scale down Docker nodes as the cluster stabilizes

To see what actions are taken when you create a node with the CRI-O container engine, refer to [Upgrading to CRI-O with Ansible](#).



NOTE

If you are upgrading your entire OpenShift Container Platform cluster to OpenShift Container Platform 3.10 or later, and a containerized version of CRI-O is running on a node, the CRI-O container will be removed from that node and the CRI-O rpm will be installed. The CRI-O service will be run as a systemd service from then on. See [BZ#1618425](#) for details.

1.3. CONFIGURING CRI-O

Because CRI-O is intended to be deployed, upgraded and managed by OpenShift Container Platform, you should only change CRI-O configuration files through OpenShift Container Platform or for the purposes of testing or troubleshooting CRI-O. On a running OpenShift Container Platform node, most CRI-O configuration settings are kept in the `/etc/crio/crio.conf` file.

Settings in a `crio.conf` file define how storage, the listening socket, runtime features, and networking are configured for CRI-O. Here's an example of the default `crio.conf` file (look in the file itself to see comments describing these settings):

```
[crio]
root = "/var/lib/containers/storage"
runroot = "/var/run/containers/storage"
storage_driver = "overlay"
storage_option = [
    "overlay.override_kernel_check=1",
]

[crio.api]
listen = "/var/run/crio/crio.sock"
stream_address = ""
stream_port = "10010"
file_locking = true

[crio.runtime]
runtime = "/usr/bin/runc"
runtime_untrusted_workload = ""
default_workload_trust = "trusted"
no_pivot = false
common = "/usr/libexec/crio/common"
common_env = [
    "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
]
selinux = true
seccomp_profile = "/etc/crio/seccomp.json"
apparmor_profile = "crio-default"
cgroup_manager = "systemd"
hooks_dir_path = "/usr/share/containers/oci/hooks.d"
default_mounts = [
    "/usr/share/rhel/secrets:/run/secrets",
]
```

```

pids_limit = 1024
enable_shared_pid_namespace = false
log_size_max = 52428800

[crio.image]
default_transport = "docker://"
pause_image = "docker.io/openshift/origin-pod:v3.11"
pause_command = "/usr/bin/pod"
signature_policy = ""
image_volumes = "mkdir"
insecure_registries = [
""
]
registries = [
"docker.io"
]

[crio.network]
network_dir = "/etc/cni/net.d/"
plugin_dir = "/opt/cni/bin"

```

The following sections describe how different CRI-O configurations might be used in the **crio.conf** file.

1.3.1. Configuring CRI-O storage

OverlayFS2 is the recommended (and default) storage driver for OpenShift Container Platform, whether you use CRI-O or Docker as your container engine. See [Choosing a graph driver](#) for details on available storage devices.



NOTE

Although devicemapper is a supported storage facility for CRI-O, the CRI-O garbage collection feature does not yet work with devicemapper and so is not recommended for production use. Also, see [BZ1625394](#) and [BZ1623944](#) for other devicemapper issues that apply to how both CRI-O and **podman** use container storage.

Things you should know about CRI-O storage include the facts that CRI-O storage:

- Holds images by storing the root filesystem of each container, along with any layers that go with it.
- Incorporates the same storage layer that is used with the Docker service.
- Uses **container-storage-setup** to manage the container storage area.
- Uses configuration information from the **/etc/containers/storage.conf** and **/etc/crio/crio.conf** files.
- Stores data in **/var/lib/containers** by default. That directory is used by both CRI-O and tools for running containers (such as **podman**).



NOTE

Although they use the same storage directory, the container engine and the container tools manage their containers separately.

- Can store both Docker version 1 and version 2 schemas.

For information on using **container-storage-setup** to configure storage for CRI-O, see [Using container-storage-setup](#).

1.3.2. Configuring CRI-O networking

CRI-O supports networking facilities that are compatible with the [Container Network Interface](#) (CNI). Supported networking features include loopback, flannel, and openshift-sdn, which are implemented as network plugins.

By default, OpenShift Container Platform uses openshift-sdn networking. The following settings in the **crio.conf** file define where CNI network configuration files are stored (`/etc/cni/net.d/`) and where CNI plugin binaries are stored (`/opt/cni/bin/`)

```
[crio.network]
network_dir = "/etc/cni/net.d/"
plugin_dir = "/opt/cni/bin/"
```

To understand the networking features needed by CRI-O in OpenShift Container Platform, refer to both [Kubernetes](#) and [OpenShift Container Platform](#) networking requirements.

1.4. TROUBLESHOOTING CRI-O

To check the health of your CRI-O container engine and troubleshoot problems, you can use the **crictl** command, along with some well-known Linux and OpenShift Container Platform commands. As with any OpenShift Container Platform container engine, you can use commands such as **oc** and **kubectl** to investigate the pods in CRI-O as well.

For example, to list pods, run the following:

```
$ sudo oc get pods -o wide
NAME                READY STATUS RESTARTS AGE IP          NODE   NOMINATED NODE
docker-registry-1-fb2g8 1/1   Running 1      5d 10.128.0.4  hostA <none>
registry-console-1-vk1l6 1/1   Running 0      5d 10.128.0.6  hostA <none>
router-1-hjfm7        1/1   Running 0      5d 192.168.122.188 hostA <none>
```

To ensure that a pod is running in CRI-O, use the **describe** option and **grep** for **cri-o**:

```
$ sudo oc describe pods registry-console-1-vk1l6 | grep cri-o
Container ID: cri-o://9a9209dc0608ce80f62bb4d7f7df61bcf8dd2abd77ef53075dee0542548238b7
```

To query and debug a CRI-O container runtime, run the **crictl** command to communicate directly with CRI-O. The CRI-O instance that **crictl** uses is identified in the **crictl.yaml** file.

```
# cat /etc/crictl.yaml
runtime-endpoint: /var/run/crio/crio.sock
```

By default, the **crictl.yaml** file causes crictl to point to the CRI-O socket on the local system. To see options available with **crictl**, run **crictl** with no arguments. To get help with a particular option, add **--help**. For example:

```
$ sudo crictl ps --help
```

```

NAME:
  crictl ps - List containers

USAGE:
  crictl ps [command options] [arguments...]

OPTIONS:
  --all, -a          Show all containers
  --id value         Filter by container id
  --label value     Filter by key=value label
  ...

```

1.4.1. Checking CRI-O's general health

Log in to a node in your OpenShift Container Platform cluster that is running CRI-O and run the following commands to check the general health of the CRI-O container engine:

Check that the CRI-O related packages are installed. That includes the `crio` (CRI-O daemon and config files) and `cri-tools` (`crictl` command) packages:

```

# rpm -qa | grep ^cri-
cri-o-1.11.6-1.rhaos3.11.git2d0f8c7.el7.x86_64
cri-tools-1.11.1-1.rhaos3.11.gitedabfb5.el7_5.x86_64

```

Check that the `crio` service is running:

```

# systemctl status -l crio
● crio.service - Open Container Initiative Daemon
  Loaded: loaded (/usr/lib/systemd/system/crio.service; enabled; vendor preset: disabled)
  Active: active (running) since Tue 2018-10-16 15:15:49 UTC; 3h 30min ago
    Docs: https://github.com/kubernetes-sigs/cri-o
  Main PID: 889 (crio)
    Tasks: 14
   Memory: 2.3G
   CGroup: /system.slice/crio.service
           └─889 /usr/bin/crio

Oct 16 15:15:48 hostA systemd[1]: Starting Open Container Initiative Daemon...
Oct 16 15:15:49 hostA systemd[1]: Started Open Container Initiative Daemon.
Oct 16 18:30:55 hostA crio[889]: time="2018-10-16 18:30:55.128074704Z" level=error

```

1.4.2. Inspecting CRI-O logs

Because the CRI-O container engine is implemented as a `systemd` service, you can use the standard `journalctl` command to inspect log messages for CRI-O.

1.4.2.1. Checking `crio` and `origin-node` logs

To check the journal for information from the `crio` service, use the `-u` option. In this example, you can see that the service is running, but a pod failed to start:

```

$ sudo journalctl -u crio
-- Logs begin at Tue 2018-10-16 15:01:31 UTC, end at Tue 2018-10-16 19:10:52 UTC. --
Oct 16 15:05:42 hostA systemd[1]: Starting Open Container Initiative Daemon...
Oct 16 15:05:42 hostA systemd[1]: Started Open Container Initiative Daemon.

```

```

Oct 16 15:06:35 hostA systemd[1]: Stopping Open Container Initiative Daemon...
Oct 16 15:06:35 hostA cri-o[4863]: time="2018-10-16 15:06:35.018523314Z" level=error msg="Failed
to start streaming server: http: Server closed"
Oct 16 15:06:35 hostA systemd[1]: Starting Open Container Initiative Daemon...
Oct 16 15:06:35 hostA systemd[1]: Started Open Container Initiative Daemon.
Oct 16 15:10:27 hostA cri-o[6874]: time="2018-10-16 15:10:26.900411457Z" level=error msg="Failed
to start streaming server: http: Server closed"
Oct 16 15:10:26 hostA systemd[1]: Stopping Open Container Initiative Daemon...
Oct 16 15:10:27 hostA systemd[1]: Stopped Open Container Initiative Daemon.
-- Reboot --
Oct 16 15:15:48 hostA systemd[1]: Starting Open Container Initiative Daemon...
Oct 16 15:15:49 hostA systemd[1]: Started Open Container Initiative Daemon.
Oct 16 18:30:55 hostA cri-o[889]: time="2018-10-16 18:30:55.128074704Z" level=error msg="Error
adding network: CNI request failed with status 400: 'pods "

```

You can also check the origin-node service for CRI-O related messages. For example:

```

$ sudo journalctl -u origin-node | grep -i cri-o

Oct 16 15:26:30 hostA origin-node[10624]: I1016 15:26:30.120889 10624
  kuberuntime_manager.go:186] Container runtime cri-o initialized,
  version: 1.11.6, apiVersion: v1alpha1
Oct 16 15:26:30 hostA origin-node[10624]: I1016 15:26:30.177213 10624
  factory.go:157] Registering CRI-O factory
Oct 16 15:27:27 hostA origin-node[11107]: I1016 15:27:27.449197 11107
  kuberuntime_manager.go:186] Container runtime cri-o initialized,
  version: 1.11.6, apiVersion: v1alpha1
Oct 16 15:27:27 hostA origin-node[11107]: I1016 15:27:27.507030 11107
  factory.go:157] Registering CRI-O factory
Oct 16 19:27:56 hostA origin-node[8326]: I1016 19:27:56.224770 8326
  kuberuntime_manager.go:186] Container runtime cri-o initialized,
  version: 1.11.6, apiVersion: v1alpha1
Oct 16 19:27:56 hostA origin-node[8326]: I1016 19:27:56.282138 8326
  factory.go:157] Registering CRI-O factory
Oct 16 19:27:57 hostA origin-node[8326]: I1016 19:27:57.783304 8326
  status_manager.go:375] Status Manager: adding pod:
  "db1f45e3-d157-11e8-8645-42010a8e0002", with status: ('\x01', {Running ...
  docker.io/openshift/origin-node:v3.11 docker.io/openshift/origin-node@sha256:6f9b0fbdd...
  cri-o://c94cc6
  2c27d021d61e8b7c1a82703d51db5847e74f5e57c667432f90c07013e4}} Burstable}} to
  podStatusChannel

```

If you wanted to further investigate what was happening with one of the pods listed, (such as the last one shown as cri-o//c94cc6), you can use the **crictl logs** command:

```

$ sudo crictl logs c94cc6
/etc/openvswitch/conf.db does not exist ... (warning).
Creating empty database /etc/openvswitch/conf.db [ OK ]
Starting ovsdb-server [ OK ]
Configuring Open vSwitch system IDs [ OK ]
Inserting openvswitch module [ OK ]
Starting ovs-vswitchd [ OK ]
Enabling remote OVSDDB managers [ OK ]

```

1.4.2.2. Turning on debugging for CRI-O

To get more details from the logging facility for CRI-O, you can temporarily set the loglevel to debug as follows:

1. Edit the `/usr/lib/systemd/system/crio.service` file and add `--loglevel=debug` to the `ExecStart=` line so it appears as follows:

```
ExecStart=/usr/bin/crio --log-level=debug \
    $CRIO_STORAGE_OPTIONS \
    $CRIO_NETWORK_OPTIONS
```

2. Reload the configuration file and restart the service as follows:

```
# systemctl daemon-reload
# systemctl restart crio
```

3. Run the `journalctl` command again. You should begin to see lots of debug messages, representing the processing going on with your CRI-O service:

```
# journalctl -u crio
Oct 18 08:41:31 mynode01-crio crio[21998]:
    time="2018-10-18 08:41:31.839702058-04:00" level=debug
    msg="ListContainersRequest
    &ListContainersRequest{Filter:&ContainerFilter{Id:,State:nil,PodSandboxId:
    ,LabelSelector:map[string]string{,},}"
Oct 18 08:41:31 mynode01-crio crio[21998]: time="2018-10-18
    08:41:31.839928476-04:00" level=debug msg="no filters were applied,
    returning full container list"
Oct 18 08:41:31 mynode01-crio crio[21998]: time="2018-10-18 08:41:31.841814536-04:00"
    level=debug msg="ListContainersResponse: &ListContainersResponse{Containers:
    [&Container{Id:e1934cc46696ff821bc35154f281764e80ac1122563ffd95aa92d01477225603,
    PodSandboxId:d904d45e6e46110a044758f20047805d8832b6859e10dc903c104cf757894e8d,
    Metadata:&ContainerMetadata{Name:c,Attempt:0,},Image:&ImageSpec{
    Image:e72de76ca8d5410497ae3171b6b059e7c7d11e4d1f3225df8d05812f29e205b7,},
    ImageRef:docker.io/openshift/origin-template-service-broker@sha256:fd539 ...
```

4. Remove the `--loglevel=debug` option when you are done investigating, to reduce the amount of messages generated. Then rerun the two `systemctl` commands:

```
# systemctl daemon-reload
# systemctl restart crio
```

1.4.3. Troubleshooting CRI-O pods, and containers

With the `crictl` command, you interface directly with the CRI-O container engine to check on and manipulate the containers, images, and pods associated with that container engine. The `runc` container runtime is another way to interact with CRI-O. If you want to run containers outside of the CRI-O container engine, for example to run support-tools on a node, you can use the `podman` command.

See [Crictl vs. Podman](#) for descriptions of those two commands and how they differ.

To begin, you can check the general status of the CRI-O service using the **crictl info** and **crictl version** commands:

```
$ sudo crictl info
  {
    "status": {
      "conditions": [
        {
          "type": "RuntimeReady",
          "status": true,
          "reason": "",
          "message": ""
        },
        {
          "type": "NetworkReady",
          "status": true,
          "reason": "",
          "message": ""
        }
      ]
    }
  }
$ sudo crictl version
Version: 0.1.0
RuntimeName: cri-o
RuntimeVersion: 1.11.6
RuntimeApiVersion: v1alpha1
```

1.4.3.1. Listing images, pods, and containers

The **crictl** command provides options for investigating the components in your CRI-O environment. Here are examples of some of the uses of **crictl** for listing information about images, pods, and containers.

To see the images that have been pulled to the local CRI-O node, run the **crictl images** command:

```
$ sudo crictl images
IMAGE                                TAG    IMAGE ID    SIZE
docker.io/openshift/oauth-proxy      v1.1.0 90c45954eb03e 242MB
docker.io/openshift/origin-haproxy-router v3.11 13f40ad4d2e21 410MB
docker.io/openshift/origin-node      v3.11 93d2aeddcd6db 1.17GB
docker.io/openshift/origin-pod       v3.11 89ceff8fb1907 263MB
docker.io/openshift/prometheus-alertmanager v0.15.2 68bbd00063784 242MB
docker.io/openshift/prometheus-node-exporter v0.16.0 f9f775bf6d0ef 225MB
quay.io/coreos/cluster-monitoring-operator v0.1.1 4488a207a5bca 531MB
quay.io/coreos/configmap-reload       v0.0.1 3129a2ca29d75 4.79MB
quay.io/coreos/kube-rbac-proxy       v0.3.1 992ac1a5e7c79 40.4MB
quay.io/coreos/kube-state-metrics    v1.3.1 a9c8f313b7aad 22.2MB
```

To see the pods that are currently active in the CRI-O environment, run **crictl pods**:

```
$ sudo crictl pods

POD ID    CREATED    STATE    NAME                                NAMESPACE    ATTEMPT
09997515d7729 5 hours ago Ready  kube-state-metrics-...  openshift-monitoring  0
```



```

958b0789e0552 5 hours ago Ready node-exporter-rkbzp openshift-monitoring 0
4ec0498dacec8 5 hours ago Ready alertmanager-main-0 openshift-monitoring 0
2873b697df1d2 5 hours ago Ready cluster-monitoring-... openshift-monitoring 0
b9e221481fb7e 5 hours ago Ready router-1-968t4 default 0
f02ce4a4b4186 5 hours ago Ready sdn-c45cm openshift-sdn 0
bdf5b1dcc0a08 5 hours ago Ready ovs-kdvzs openshift-sdn 0
49dbc57455c8f 5 hours ago Ready sync-hgfvb openshift-node 0

```

To see containers that are currently running, run the **crictl ps** command:

```

$ sudo crictl ps
CONTAINER ID IMAGE CREATED STATE NAME ATTEMPT
376eb13e3cb37 quay.io/coreos/kube-state-metrics... 4 hours ago Running kube-state-metrics 0
72d61c3d393b5 992ac1a5e7c79d627321dc7877f741a00... 4 hours ago Running kube-rbac-proxy-
self 0
5fa8c93484055 992ac1a5e7c79d627321dc7877f741a00... 4 hours ago Running kube-rbac-proxy-
main 0
a2d35508fc0ee quay.io/coreos/kube-rbac-proxy... 4 hours ago Running kube-rbac-proxy 0
9adda43f3595f docker.io/openshift/prometheus-no... 4 hours ago Running node-exporter 0
7f4ce5b25cfdb docker.io/openshift/oauth-proxy... 4 hours ago Running alertmanager-proxy 0
85418badbf6ae quay.io/coreos/configmap-reload... 4 hours ago Running config-reloader 0
756f20138381c docker.io/openshift/prometheus-al... 4 hours ago Running alertmanager 0
5e6d8ff4852ba quay.io/coreos/cluster-monitoring... 4 hours ago Running cluster-monitoring- 0
1c96cfcfa10a7 docker.io/openshift/origin-haprox... 5 hours ago Running route 0
8f90bb4cded60 docker.io/openshift/origin-node... 5 hours ago Running sdn 0
59e5fb8514262 docker.io/openshift/origin-node... 5 hours ago Running openvswitch 0
73323a2c26abe docker.io/openshift/origin-node... 5 hours ago Running sync 0

```

To see both running containers as well as containers that are stopped or exited, run **crictl ps -a**:

```
$ sudo crictl ps -a
```

If your CRI-O service is stopped or malfunctioning, you can list the containers that were run in CRI-O using the **runc** command. This example searches for the existence of a container with CRI-O running and not running. It then shows that you can investigate that container with **runc**, even when CRI-O is stopped:

```

$ crictl ps | grep d36a99a9a40ec
d36a99a9a40ec 062cd20609d3895658e54e5f367b9d70f42db4f86ca14bae7309512c7e0777fd
11 hours ago CONTAINER_RUNNING sync 2
$ sudo systemctl stop crio
$ sudo crictl ps | grep d36a99a9a40ec
2018/10/25 11:22:16 grpc: addrConn.resetTransport failed to create client transport:
connection error: desc = "transport: dial unix /var/run/crio/crio.sock: connect:
no such file or directory"; Reconnecting to {/var/run/crio/crio.sock <nil>}
FATA[0000] listing containers failed: rpc error: code = Unavailable desc = grpc:
the connection is unavailable
$ sudo runc list | grep d36a99a9a40ec
d36a99a9a40ecc4c830f10ed2d5bb3ce1c6deadc1a4879ff342e315051a71ed 19477 running
/run/containers/storage/overlay-
containers/d36a99a9a40ecc4c830f10ed2d5bb3ce1c6deadc1a4879ff342e315051a71ed/userdata
2018-10-25T04:44:29.47950187Z root
$ ls /run/containers/storage/overlay-containers/d36*/userdata/
attach config.json ctl pidfile run
$ less /run/containers/storage/overlay-containers/d36*/userdata/config.json

```

```
{
  "ociVersion": "1.0.0",
  "process": {
    "user": {
      "uid": 0,
      "gid": 0
    },
    "args": [
      "/bin/bash",
      "-c",
      "#!/bin/bash\nset -euo pipefail\n\n# set by the node
      image\nunset KUBECONFIG\n\ntrap 'kill $(jobs -p);
      exit 0' TERM\n\n# track the current state of the ...
      $ sudo systemctl start cri-o
    ]
  }
}
```

As you can see, even with the CRI-O service off, **runc** shows the existence of the container and its location in the file system, in case you want to look into it further.

1.4.3.2. Investigating images, pods, and containers

To find out details about what is happening inside of images, pods or containers for your CRI-O environment, there are several **crictl** options you can use.

With a container ID in hand (from the output of **crictl ps**), you can exec a command inside that container. For example, to see the name and release of the operating system inside of a container, run:

```
$ crictl exec 756f20138381c cat /etc/redhat-release
CentOS Linux release 7.5.1804 (Core)
```

To see a list of processes running inside of a container, run:

```
$ crictl exec -t e47b3a837aa30 ps -ef
UID      PID PPID C STIME TTY      TIME CMD
1000130+  1   0  0 Oct17 ?        00:38:14 /usr/bin/origin-web-console --au
1000130+ 15894 0  0 15:38 pts/0    00:00:00 ps -ef
1000130+ 17518 1  0 Oct23 ?        00:00:00 [curl] <defunct>
```

As an alternative, you can "exec" into a container using the **runc** command:

```
$ sudo runc exec -t e47b3a837aa3023c748c4c31a090266f014afba641a8ab9cfca31b065b4f2ddd ps
-ef
UID      PID PPID C STIME TTY      TIME CMD
1000130+  1   0  0 Oct17 ?        00:38:16 /usr/bin/origin-web-console --audit-log-path=- -v=0 --
config=/var/webconsole-config/webc
1000130+ 16541 0  0 15:48 pts/0    00:00:00 ps -ef
1000130+ 17518 1  0 Oct23 ?        00:00:00 [curl] <defunct>
```

If there is no **ps** command inside the container, **runc** has the **ps** option, which has the same effect of showing the processes running in the container:

```
$ sudo runc ps e47b3a837aa3023c748c4c31a090266f014afba641a8ab9cfca31b065b4f2ddd
```

Note that **runc** requires the full container ID, while **crictl** only needs a few unique characters from the beginning.

With a pod sandbox ID in hand (output from **crictl pods**), run **crictl inspectp** to display information about that pod sandbox:

```
$ sudo crictl pods | grep 5a60ac777aaa0
5a60ac777aaa0 8 days ago SANDBOX_READY registry-console-1-vk1l6 default 0
$ sudo crictl inspectp 5a60ac777aaa0
{
  "status": {
    "id": "5a60ac777aaa055f14b998a9f2ced3e146b3cddb270154abb75decd583bf879",
    "metadata": {
      "attempt": 0,
      "name": "registry-console-1-vk1l6",
      "namespace": "default",
      "uid": "6af860cc-d20b-11e8-b094-525400535ba1"
    },
    "state": "SANDBOX_READY",
    "createdAt": "2018-10-17T08:53:22.828511516-04:00",
    "network": {
      "ip": "10.128.0.6"
    }
  }
}
```

To see status information about an image that is available to CRI-O on the local system, run **crictl inspecti**:

```
$ sudo crictl inspecti ff5dd2137a4ff
{
  "status": {
    "id": "ff5dd2137a4ffd5ccb9837d5a0aa0a5d10729f9c186df02e54e58748a32d08b0",
    "repoTags": [
      "quay.io/coreos/etcd:v3.2.22"
    ],
    "repoDigests": [
      "quay.io/coreos/etcd@sha256:43fbc8a457aa0cb887da63d74a48659e13947cb74b96a53ba8f47abb6172a948"
    ],
    "size": "37547599",
    "username": ""
  }
}
```

Additional resources

- [CRI-O - OCI-based implementation of Kubernetes Container Runtime Interface](#)
- [CRI-O Lightweight Container Runtime for Kubernetes](#)
- [CRI-O Command Line Interface: crictl](#)
- [Finding, Running, and Building Containers without Docker](#)
- [Container Commandos Coloring Book](#)
- [CRI-O now running production workloads in OpenShift Online](#)
- [CRI-O How Standards Power a Container Runtime](#)

- [A Practical Introduction to Container Terminology](#)