# OpenShift Container Platform 3.11

## Scaling and Performance Guide

OpenShift Container Platform 3.11 Scaling and Performance Guide

Last Updated: 2022-05-02

# OpenShift Container Platform 3.11 Scaling and Performance Guide

OpenShift Container Platform 3.11 Scaling and Performance Guide

## Abstract

Scale up your cluster and tune performance in production environments

# Table of Contents

# CHAPTER 1. OVERVIEW

This guide provides procedures and examples for how to enhance your OpenShift Container Platform cluster performance and conduct scaling at different levels of an OpenShift Container Platform production stack. It includes recommended practices for building, scaling, and tuning OpenShift Container Platform clusters.

Tuning considerations can vary depending on your cluster setup, and be advised that any performance recommendations in this guide might come with trade-offs.

# CHAPTER 2. RECOMMENDED INSTALLATION PRACTICES

## 2.1. PRE-INSTALLING DEPENDENCIES

A node host will access the network to install any RPMs dependencies, such as **atomic-openshift-***, **iptables**, and **CRI-O** or **Docker**. Pre-installing these dependencies, creates a more efficient install, because the RPMs are only accessed when necessary, instead of a number of times per host during the install.

This is also useful for machines that cannot access the registry for security purposes.

## 2.2. ANSIBLE INSTALL OPTIMIZATION

The OpenShift Container Platform install method uses Ansible. Ansible is useful for running parallel operations, meaning a fast and efficient installation. However, these can be improved upon with additional tuning options. See the Configuring Ansible section for a list of available Ansible configuration options.

### IMPORTANT

Parallel behavior can overwhelm a content source, such as your image registry or Red Hat Satellite server. Preparing your server's infrastructure pods and operating system patches can help prevent this issue.

Run the installer from the lowest-possible latency control node (LAN speeds). Running over a wide area network (WAN) is not advised, neither is running the installation over a lossy network connection.

Ansible provides its own guidance for performance and scaling , including using RHEL 6.6 or later to ensure the version of OpenSSH supports **ControlPersist**, and running the installer from the same LAN as the cluster, but *not* running it from a machine in the cluster.

The following is an example Ansible configuration for large cluster installation and administration that incorporates the recommendations documented by Ansible:

```
# cat /etc/ansible/ansible.cfg
```

**Example Output**

```
# config file for ansible -- http://ansible.com/
# ==============================================
[defaults]
forks = 20 (1)
host_key_checking = False
remote_user = root
roles_path = roles/
gathering = smart
fact_caching = jsonfile
fact_caching_connection = $HOME/ansible/facts
fact_caching_timeout = 600
log_path = $HOME/ansible.log
nocows = 1
callback_whitelist = profile_tasks
```

```
[privilege_escalation]
become = False

[ssh_connection]
ssh_args = -o ControlMaster=auto -o ControlPersist=600s -o ServerAliveInterval=60
control_path = %(directory)s/%%h-%%r
pipelining = True (2)
timeout = 10
```

1. 20 forks is ideal, because larger forks can lead to installations failing.
2. Pipelining reduces the number of connections between control and target nodes, helping to improve installer performance.

## 2.3. NETWORKING CONSIDERATIONS

Network subnets can be changed post-install, but with difficulty. It is much easier to consider the network subnet size prior to installation, because underestimating the size can create problems with growing clusters.

See the Network Optimization topic for recommended network subnetting practices.

# CHAPTER 3. RECOMMENDED HOST PRACTICES

## 3.1. RECOMMENDED PRACTICES FOR OPENSHIFT CONTAINER PLATFORM MASTER HOSTS

In addition to pod traffic, the most-used data-path in an OpenShift Container Platform infrastructure is between the OpenShift Container Platform master hosts and etcd. The OpenShift Container Platform API server (part of the master binary) consults etcd for node status, network configuration, secrets, and more.

Optimize this traffic path by:

- Running etcd on master hosts. By default, etcd runs in a static pod on all master hosts.
- Ensuring an uncongested, low latency LAN communication link between master hosts.

The OpenShift Container Platform master caches deserialized versions of resources aggressively to ease CPU load. However, in smaller clusters of less than 1000 pods, this cache can waste a lot of memory for negligible CPU load reduction. The default cache size is 50,000 entries, which, depending on the size of your resources, can grow to occupy 1 to 2 GB of memory. This cache size can be reduced using the following setting the in ***/etc/origin/master/master-config.yaml***:

```
kubernetesMasterConfig:
  apiServerArguments:
    deserialization-cache-size:
    - "1000"
```

The number of client requests or API calls that are sent to the API server is determined by the Queries per second (QPS) value and the number of concurrent requests that can be processed by the API server is determined by the maxRequestsInFlight setting. The number of requests the client can make in excess of the QPS rate depends on the burst value, this is helpful for applications that are bursty in nature and can make irregular number of requests. The Response times for requests might have high latencies when there are large numbers of concurrent requests being handled by the API server especially for large and/or dense clusters. It is recommended to monitor the **apiserver_request_count** rate metric in Prometheus and adjust the **maxRequestsInFlight** and **QPS** accordingly.

There needs to be a good balance when changing the default values as the CPU and memory consumption of API server and etcd IOPS will increase when it is handling more requests in parallel. Also note that heavy non-watch requests might overload the API server as they get cancelled after a fixed 60 second timeout and the client starts retrying.

Provided sufficient CPU and memory resources are available on the API server system(s), the API server requests overloading issue can be safely alleviated by taking into account the factors mentioned above and bumping up the maxRequestsInFlight, API qps and burst values in the *_/etc/origin/master/master-config.yaml

```
masterClients:
  openshiftLoopbackClientConnectionOverrides:
    burst: 600
    qps: 300
servingInfo:
  maxRequestsInFlight: 500
```

NOTE

The maxRequestsInFlight, qps and burst values above are defaults for OpenShift Container Platform. The qps can be higher than maxRequestsInFlight value if the requests take less than a second. If `maxRequestsInFlight' is set to zero, there is no limit on the number of concurrent requests that can be processed by the server.

## 3.2. RECOMMENDED PRACTICES FOR OPENSHIFT CONTAINER PLATFORM NODE HOSTS

The OpenShift Container Platform node configuration file contains important options, such as the iptables synchronization period, the Maximum Transmission Unit (MTU) of the SDN network, and the proxy-mode. To configure your nodes, modify the appropriate node configuration map.

NOTE

Do not edit the **node-config.yaml** file directly.

The node configuration file allows you to pass arguments to the kubelet (node) process. You can view a list of possible options by running **kubelet --help**.

NOTE

Not all kubelet options are supported by OpenShift Container Platform, and are used in the upstream Kubernetes. This means certain options are in limited support.

NOTE

See the Cluster maximums page for the maximum supported limits for each version of OpenShift Container Platform.

In the ***/etc/origin/node/node-config.yaml*** file, one parameter controls the maximum number of pods that can be scheduled to a node: **max-pods**. When the **max-pods** option is in use, it limits the number of pods on a node. Exceeding this value can result in:

- Increased CPU utilization on both OpenShift Container Platform and Docker.
- Slow pod scheduling.
- Potential out-of-memory scenarios (depends on the amount of memory in the node).
- Exhausting the pool of IP addresses.
- Resource overcommitting, leading to poor user application performance.

NOTE

In Kubernetes, a pod that is holding a single container actually uses two containers. The second container is used to set up networking prior to the actual container starting. Therefore, a system running 10 pods will actually have 20 containers running.

**max-pods** sets the number of pods the node can run to a fixed value, regardless of the properties of the node. Cluster Limits documents maximum supported values for **max-pods**.

```
kubeletArguments:
  max-pods:
    - "250"
```

Using the above example, the default value for **max-pods** is **250**.

See the Sizing Considerations section in the installation documentation for the recommended limits for an OpenShift Container Platform cluster. The recommended sizing accounts for OpenShift Container Platform and container engine coordination for container status updates. This coordination puts CPU pressure on the master and container engine processes, which can include writing a large amount of log data.

The rate at which kubelet talks to API server depends on qps and burst values. The default values are good enough if there are limited pods running on each node. Provided there are enough CPU and memory resources on the node, the qps and burst values can be tweaked in the ***/etc/origin/node/node-config.yaml*** file:

```
kubeletArguments:
  kube-api-qps:
  - "20"
  kube-api-burst:
  - "40"
```

NOTE

The qps and burst values above are defaults for OpenShift Container Platform.

## 3.3. RECOMMENDED PRACTICES FOR OPENSHIFT CONTAINER PLATFORM ETCD HOSTS

etcd is a distributed key-value store that OpenShift Container Platform uses for configuration.

| OpenShift Container Platform Version | etcd version | storage schema version |
|---|---|---|
| 3.3 and earlier | 2.x | v2 |
| 3.4 and 3.5 | 3.x | v2 |
| 3.6 | 3.x | v2 (upgrades) |
| 3.6 | 3.x | v3 (new installations) |
| 3.7 and later | 3.x | v3 |

etcd 3.x introduces important scalability and performance improvements that reduce CPU, memory, network, and disk requirements for any size cluster. etcd 3.x also implements a backwards compatible storage API that facilitates a two-step migration of the on-disk etcd database. For migration purposes, the storage mode used by etcd 3.x in OpenShift Container Platform 3.5 remained in v2 mode. As of

OpenShift Container Platform 3.6, new installations use storage mode v3. Upgrades from previous versions of OpenShift Container Platform will *not* automatically migrate data from v2 to v3. You must use the supplied playbooks and follow the documented process to migrate the data.

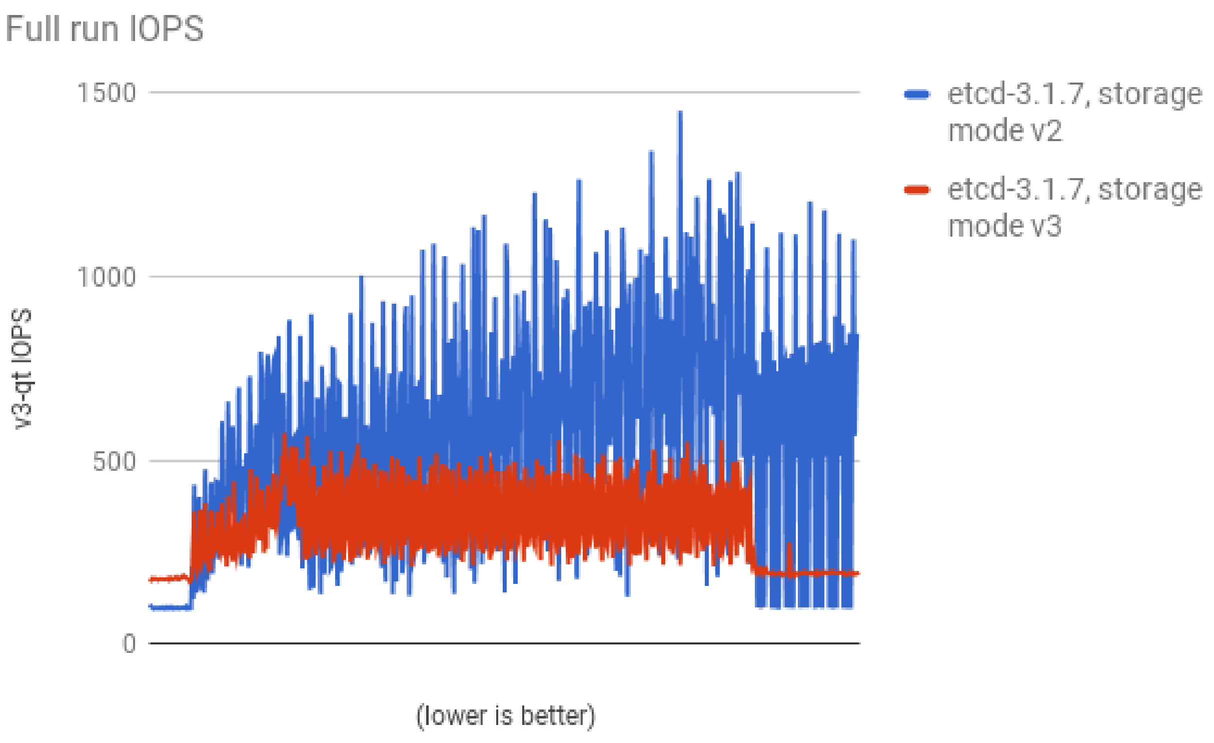
Version 3 of etcd implements a backwards compatible storage API that facilitates a two-step migration of the on-disk etcd database. For migration purposes, the storage mode used by etcd 3.x in OpenShift Container Platform 3.5 remained in v2 mode. As of OpenShift Container Platform 3.6, new installations use storage mode v3. As part of the process to upgrade to OpenShift Container Platform 3.7, you upgrade your etcd storage API to v3, if required. In versions 3.7 and later, you must use the v3 API.

In addition to changing the storage mode for new installs to v3, OpenShift Container Platform 3.6 also begins enforcing *quorum reads* for all OpenShift Container Platform types. This is done to ensure that queries against etcd do not return stale data. In single-node etcd clusters, stale data is not a concern. In highly available etcd deployments typically found in production clusters, quorum reads ensure valid query results. A quorum read is *linearizable* in database terms - every client sees the latest updated state of the cluster, and all clients see the same sequence of reads and writes. See the etcd 3.1 announcement for more information on performance improvements.

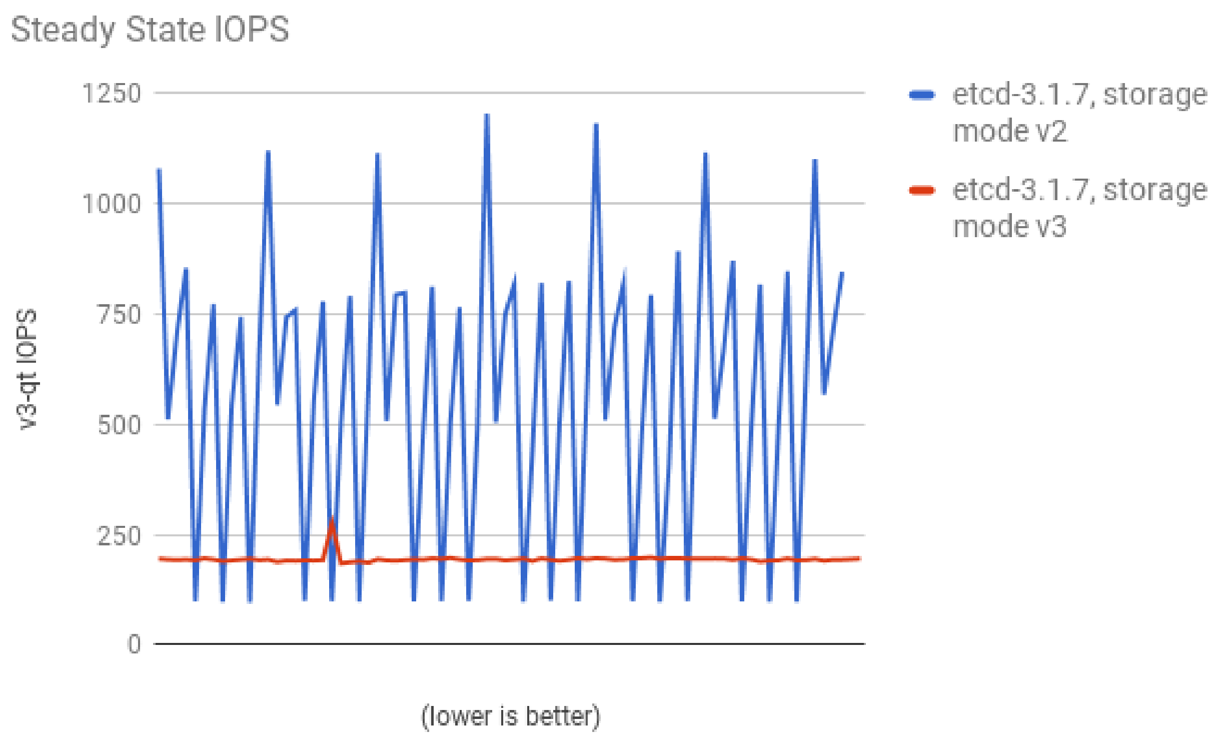
It is important to note that OpenShift Container Platform uses etcd for storing additional information beyond what Kubernetes itself requires. For example, OpenShift Container Platform stores information about images, builds, and other components in etcd, as is required by features that OpenShift Container Platform adds on top of Kubernetes. Ultimately, this means that guidance around performance and sizing for etcd hosts will differ from Kubernetes and other recommendations in salient ways. Red Hat tests etcd scalability and performance with the OpenShift Container Platform use-case and parameters in mind to generate the most accurate recommendations.

Performance improvements were quantified using a 300-node OpenShift Container Platform 3.6 cluster using the cluster-loader utility. Comparing etcd 3.x (storage mode v2) versus etcd 3.x (storage mode v3), clear improvements are identified in the charts below.

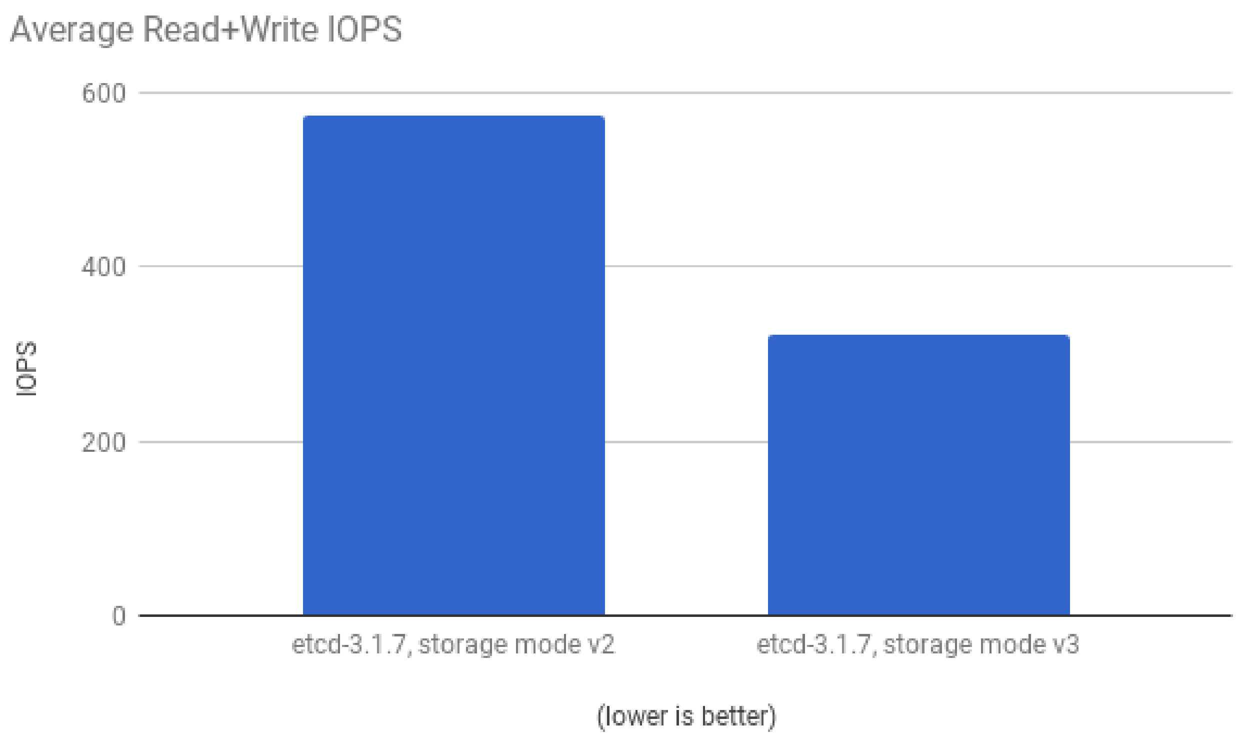
Storage IOPS under load is significantly reduced:



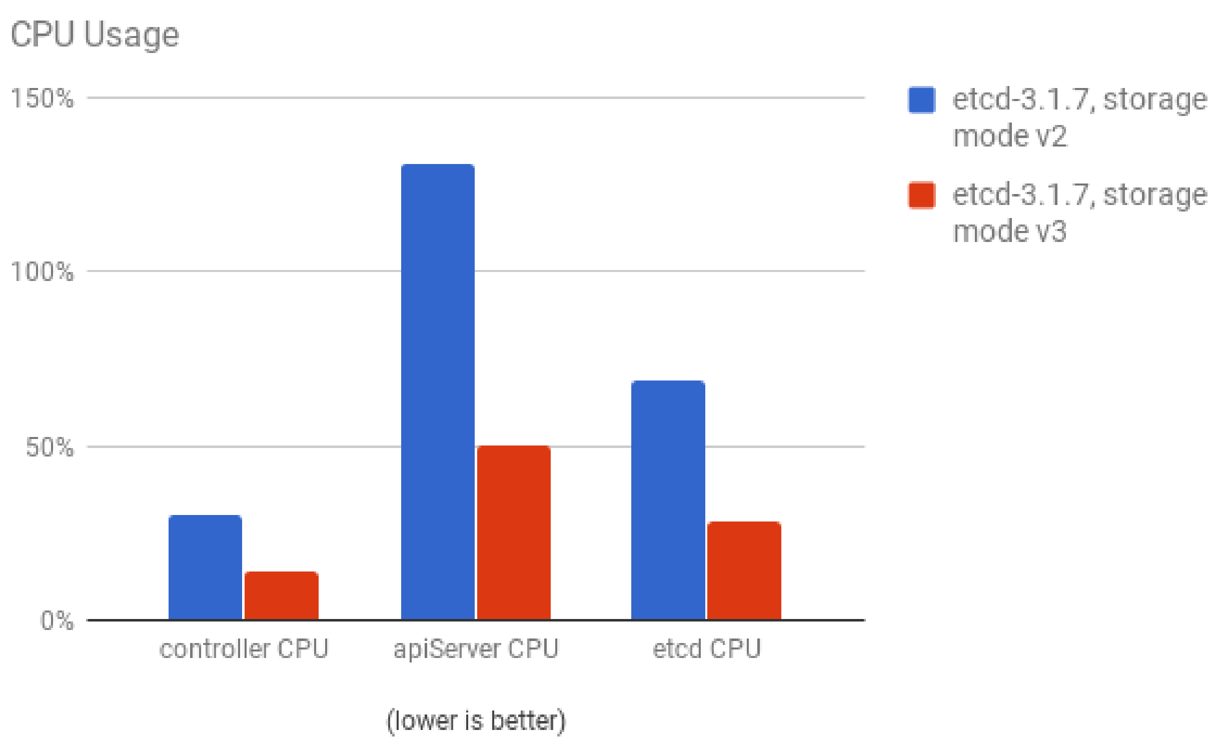
Storage IOPS in steady state is also significantly reduced:



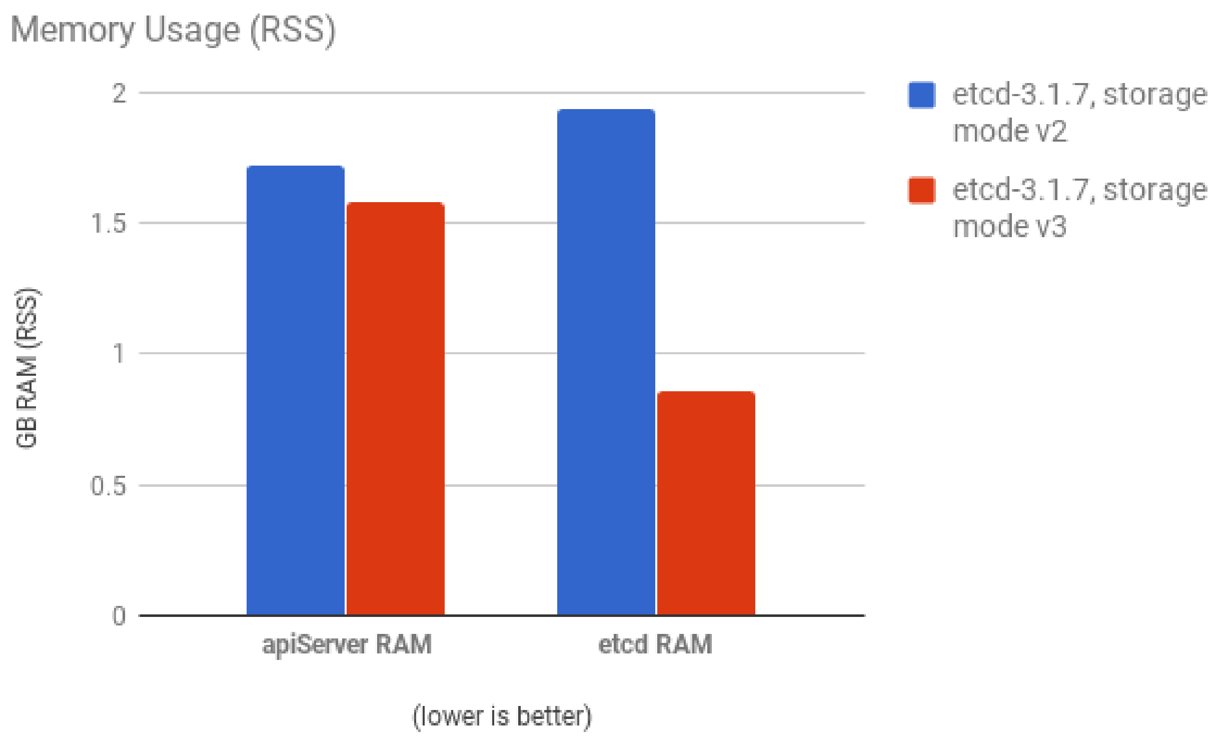
Viewing the same I/O data, plotting the average IOPS in both modes:



CPU utilization by both the API server (master) and etcd processes is reduced:



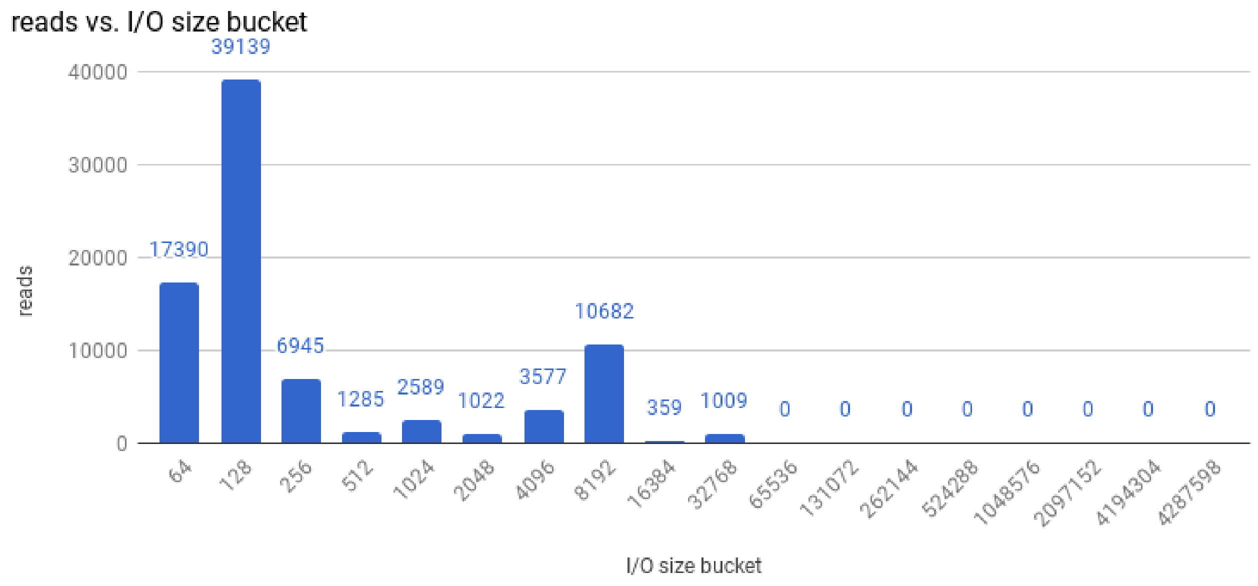
Memory utilization by both the API server (master) and etcd processes is also reduced:



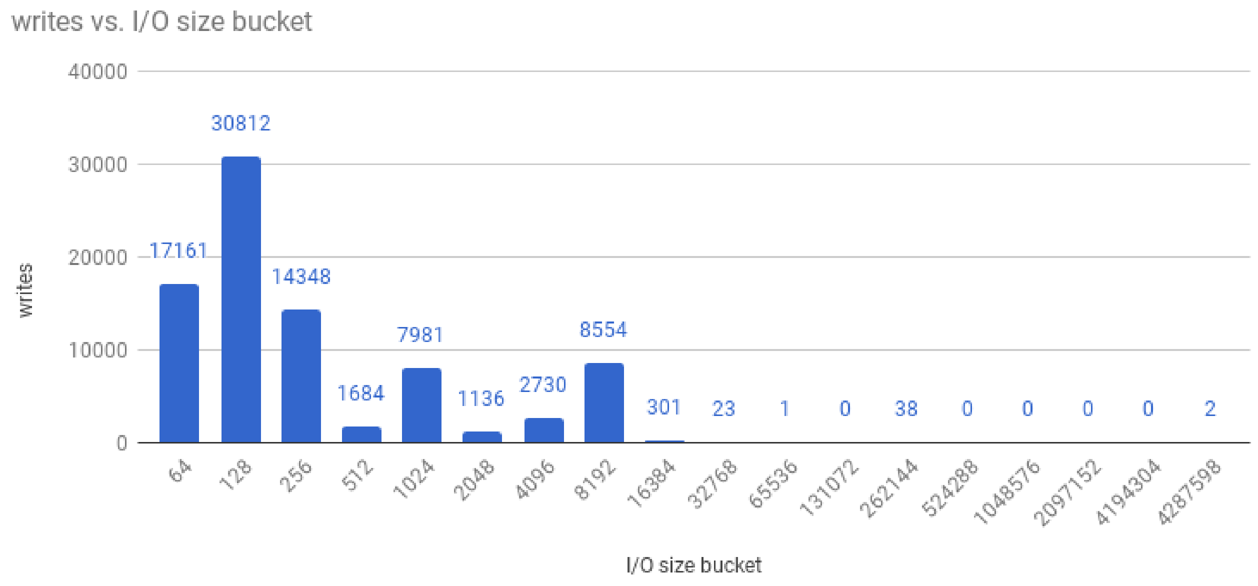
**IMPORTANT**

After profiling etcd under OpenShift Container Platform, etcd frequently performs small amounts of storage input and output. Using etcd with storage that handles small read/write operations quickly, such as SSD, is highly recommended.

Looking at the size I/O operations done by a 3-node cluster of etcd 3.1 (using storage v3 mode and with quorum reads enforced), read sizes are as follows:



And writes:



**NOTE**

etcd processes are typically memory intensive. Master / API server processes are CPU intensive. This makes them a reasonable co-location pair within a single machine or virtual machine (VM). Optimize communication between etcd and master hosts either by co-locating them on the same host, or providing a dedicated network.

### 3.3.1. Providing Storage to an etcd Node Using PCI Passthrough with OpenStack

To provide fast storage to an etcd node so that etcd is stable at large scale, use PCI passthrough to pass a non-volatile memory express (NVMe) device directly to the etcd node. To set this up with Red Hat OpenStack 11 or later, complete the following on the OpenStack compute nodes where the PCI device exists.

1. Ensure Intel Vt-x is enabled in BIOS.

2. Enable the input–output memory management unit (IOMMU). In the ***/etc/sysconfig/grub*** file, add **intel_iommu=on iommu=pt** to the end of the **GRUB_CMDLINX_LINUX** line, within the quotation marks.

3. Regenerate ***/etc/grub2.cfg*** by running:

   ```
   $ grub2-mkconfig -o /etc/grub2.cfg
   ```

4. Reboot the system.

5. On controllers in ***/etc/nova.conf***:

   ```
   [filter_scheduler]

   enabled_filters=RetryFilter,AvailabilityZoneFilter,RamFilter,DiskFilter,ComputeFilter,ComputeC
   apabilitiesFilter,ImagePropertiesFilter,ServerGroupAntiAffinityFilter,ServerGroupAffinityFilter,Pci
   PassthroughFilter

   available_filters=nova.scheduler.filters.all_filters

   [pci]

   alias = { "vendor_id":"144d", "product_id":"a820",
   "device_type":"type-PCI", "name":"nvme" }
   ```

6. Restart **nova-api** and **nova-scheduler** on the controllers.

7. On compute nodes in ***/etc/nova/nova.conf***:

   ```
   [pci]

   passthrough_whitelist = { "address": "0000:06:00.0" }

   alias = { "vendor_id":"144d", "product_id":"a820",
   "device_type":"type-PCI", "name":"nvme" }
   ```

   To retrieve the required **address**, **vendor_id**, and **product_id** values of the NVMe device you want to passthrough, run:

   ```
   # lspci -nn | grep devicename
   ```

8. Restart **nova-compute** on the compute nodes.

9. Configure the OpenStack version you are running to use the NVMe and launch the etcd node.

## 3.4. SCALING HOSTS USING THE TUNED PROFILE

**Tuned** is a tuning profile delivery mechanism enabled by default in Red Hat Enterprise Linux (RHEL) and other Red Hat products. **Tuned** customizes Linux settings, such as sysctls, power management, and kernel command line options, to optimize the operating system for different workload performance and scalability requirements.

OpenShift Container Platform leverages the **tuned** daemon and includes **Tuned** profiles called **openshift**, **openshift-node** and **openshift-control-plane**. These profiles safely increase some of the commonly encountered vertical scaling limits present in the kernel, and are automatically applied to your system during installation.

The **Tuned** profiles support inheritance between profiles. They also support an auto-parent functionality which selects a parent profile based on whether the profile is used in a virtual environment. The **openshift** profile uses both of these features and is a parent of **openshift-node** and **openshift-control-plane** profiles. It contains tuning relevant to both OpenShift Container Platform application nodes and control plane nodes respectively. The **openshift-node** and **openshift-control-plane** profiles are set on application and control plane nodes respectively.

The profile hierarchy with the **openshift** profile as a parent ensures the tuning delivered to the OpenShift Container Platform system is a union of **throughput-performance** (the default for RHEL) for bare metal hosts and **virtual-guest** for RHEL and **atomic-guest** for RHEL Atomic Host nodes.

To see which **Tuned** profile is enabled on your system, run:

```
# tuned-adm active
```

### Example Output

```
Current active profile: openshift-node
```

See the Red Hat Enterprise Linux Performance Tuning Guide for more information about **Tuned**.

# CHAPTER 4. OPTIMIZING COMPUTE RESOURCES

## 4.1. OVERCOMMITTING

You can use overcommit procedures so that resources such as CPU and memory are more accessible to the parts of your cluster that need them.

IMPORTANT

To avoid erratic cluster behavior due to scheduling collisions between the hypervisor and Kubernetes, do not overcommit at the hypervisor level.

Note that when you overcommit, there is a risk that another application may not have access to the resources it requires when it needs them, which will result in reduced performance. However, this may be an acceptable trade-off in favor of increased density and reduced costs. For example, development, quality assurance (QA), or test environments may be overcommitted, whereas production might not be.

OpenShift Container Platform implements resource management through the compute resource model and quota system. See the documentation for more information about the OpenShift resource model.

For more information and strategies for overcommitting, see the Overcommitting documentation in the Cluster Administration Guide.

## 4.2. IMAGE CONSIDERATIONS

### 4.2.1. Using a Pre-deployed Image to Improve Efficiency

You can create a base OpenShift Container Platform image with a number of tasks built-in to improve efficiency, maintain configuration consistency on all node hosts, and reduce repetitive tasks. This is known as a pre-deployed image.

For example, because every node requires the **ose-pod** image in order to run pods, each node has to periodically connect to the container image registry in order to pull the latest image. This can become problematic when you have 100 nodes attempting this at the same time, and can lead to resource contention on the image registry, waste of network bandwidth, and increased pod launch times.

To build a pre-deployed image:

- Create an instance of the type and size required.
- Ensure a dedicated storage device is available for CRI-O or Docker local image or container storage, separate from any persistent volumes for containers.
- Fully update the system, and ensure CRI-O or Docker is installed.
- Ensure the host has access to all yum repositories.
- Set up thin-provisioned LVM storage.
- Pre-seed your commonly used images (such as the **rhel7** base image), as well as OpenShift Container Platform infrastructure container images (**ose-pod**, **ose-deployer**, etc.) into your pre-deployed image.

Ensure that pre-deployed images are configured for any appropriate cluster configurations, such as being able to run on OpenStack, or AWS, as well as any other cluster configurations.

### 4.2.2. Pre-pulling Images

To efficiently produce images, you can pre-pull any necessary container images to all node hosts. This means the image does not have to be initially pulled, which saves time and performance over slow connections, especially for images, such as S2I, metrics, and logging, which can be large.

This is also useful for machines that cannot access the registry for security purposes.

Alternatively, you can use a local image instead of the default of a specified registry. To do this:

1. Pull from local images by setting the **imagePullPolicy** parameter of a pod configuration to **IfNotPresent** or **Never**.
2. Ensure that all nodes in the cluster have the same images saved locally.

NOTE

Pulling from a local registry is suitable if you can control node configuration. However, it will not work reliably on cloud providers that do not replace nodes automatically, such as GCE. If you are running on Google Container Engine (GKE), there will already be a ***.dockercfg*** file on each node with Google Container Registry credentials.

## 4.3. DEBUGGING USING THE RHEL TOOLS CONTAINER IMAGE

Red Hat distributes a **rhel-tools** container image, packaging tools that aid in debugging scaling or performance problems. This container image:

- Allows users to deploy minimal footprint container hosts by moving packages out of the base distribution and into this support container.
- Provides debugging capabilities for Red Hat Enterprise Linux 7 Atomic Host, which has an immutable package tree. **rhel-tools** includes utilities such as tcpdump, sosreport, git, gdb, perf, and many more common system administration utilities.

Use the **rhel-tools** container with the following:

```
# atomic run rhel7/rhel-tools
```

See the RHEL Tools Container documentation for more information.

## 4.4. DEBUGGING USING ANSIBLE-BASED HEALTH CHECKS

Additional diagnostic health checks are available through the Ansible-based tooling used to install and manage OpenShift Container Platform clusters. They can report common deployment problems for the current OpenShift Container Platform installation.

These checks can be run either using the **ansible-playbook** command (the same method used during cluster installations) or as a containerized version of **openshift-ansible**. For the **ansible-playbook** method, the checks are provided by the **openshift-ansible** RPM package. For the containerized method, the **openshift3/ose-ansible** container image is distributed via the Red Hat Container Registry .

See Ansible-based Health Checks in the Cluster Administration guide for information on the available health checks and example usage.

# CHAPTER 5. OPTIMIZING PERSISTENT STORAGE

## 5.1. OVERVIEW

Optimizing storage helps to minimize storage use across all resources. By optimizing storage, administrators help ensure that existing storage resources are working in an efficient manner.

**NOTE**

This guide primarily focuses on optimizing persistent storage. Local ephemeral storage for data utilized during the lifetime of pods has fewer options. Ephemeral storage is only available if you enabled the ephemeral storage technology preview. This feature is disabled by default. See configuring for ephemeral storage for more information.

## 5.2. GENERAL STORAGE GUIDELINES

The following table lists the available persistent storage technologies for OpenShift Container Platform.

**Table 5.1. Available storage options**

| Storage type | Description | Examples |
|---|---|---|
| Block | • Presented to the operating system (OS) as a block device<br>• Suitable for applications that need full control of storage and operate at a low level on files bypassing the file system<br>• Also referred to as a Storage Area Network (SAN)<br>• Non-shareable, which means that only one client at a time can mount an endpoint of this type | converged mode/independent mode GlusterFS [1], iSCSI, Fibre Channel, Ceph RBD, OpenStack Cinder, AWS EBS [1], Dell/EMC Scale.IO, VMware vSphere Volume, GCE Persistent Disk [1], Azure Disk |
| File | • Presented to the OS as a file system export to be mounted<br>• Also referred to as Network Attached Storage (NAS)<br>• Concurrency, latency, file locking mechanisms, and other capabilities vary widely between protocols, implementations, vendors, and scales. | converged mode/independent mode GlusterFS [1], RHEL NFS, NetApp NFS [2], Azure File, Vendor NFS, Vendor GlusterFS [3], Azure File, AWS EFS |

| Storage type | Description | Examples |
|---|---|---|
| Object | • Accessible through a REST API endpoint<br>• Configurable for use in the OpenShift Container Platform Registry<br>• Applications must build their drivers into the application and/or container. | converged mode/independent mode GlusterFS [1], Ceph Object Storage (RADOS Gateway), OpenStack Swift, Aliyun OSS, AWS S3, Google Cloud Storage, Azure Blob Storage, Vendor S3 [3], Vendor Swift [3] |

1. converged mode/independent mode GlusterFS, Ceph RBD, OpenStack Cinder, AWS EBS, Azure Disk, GCE persistent disk, and VMware vSphere support dynamic persistent volume (PV) provisioning natively in OpenShift Container Platform.
2. NetApp NFS supports dynamic PV provisioning when using the Trident plug-in.
3. Vendor GlusterFS, Vendor S3, and Vendor Swift supportability and configurability might vary.

You can use converged mode GlusterFS (a hyperconverged or cluster-hosted storage solution) or independent mode GlusterFS (an externally hosted storage solution) for block, file, and object storage for OpenShift Container Platform registry, logging, and monitoring.

## 5.3. STORAGE RECOMMENDATIONS

The following table summarizes the recommended and configurable storage technologies for the given OpenShift Container Platform cluster application.

**Table 5.2. Recommended and configurable storage technology**

| Storage type | RWO [1] | ROX [2] | RWX [3] | Registry | Scaled registry | Monitoring | Logging | Apps |
|---|---|---|---|---|---|---|---|---|
| Block | Yes | Yes [4] | No | Configurable | Not configurable | Recommended | Recommended | Recommended |
| File | Yes | Yes [4] | Yes | Configurable | Configurable | Configurable [5] | Configurable [6] | Recommended |
| Object | Yes | Yes | Yes | Recommended | Recommended | Not configurable | Not configurable | Not configurable [7] |

1. ReadWriteOnce
2. ReadOnlyMany
3. ReadWriteMany

4. This does not apply to physical disk, VM physical disk, VMDK, loopback over NFS, AWS EBS, Azure Disk and Cinder (the latter for block).

5. For monitoring components, using file storage with the ReadWriteMany (RWX) access mode is unreliable. If you use file storage, do not configure the RWX access mode on any persistent volume claims (PVCs) that are configured for use with monitoring.

6. For logging, using any shared storage would be an anti-pattern. One volume per logging-es is required.

7. Object storage is not consumed through OpenShift Container Platform’s PVs or PVCs. Apps must integrate with the object storage REST API.

NOTE

A scaled registry is an OpenShift Container Platform registry where three or more pod replicas are running.

### 5.3.1. Specific application storage recommendations

IMPORTANT

Testing shows issues with using the RHEL NFS server as storage backend for the container image registry. This includes the OpenShift Container Registry and Quay, Prometheus for metrics storage, and ElasticSearch for logging storage. Therefore, using the RHEL NFS server to back PVs used by core services is not recommended.

Other NFS implementations on the marketplace might not have these issues. Contact the individual NFS implementation vendor for more information on any testing that was possibly completed against these OpenShift core components.

#### 5.3.1.1. Registry

In a non-scaled/high-availability (HA) OpenShift Container Platform registry cluster deployment:

- The preferred storage technology is object storage followed by block storage. The storage technology does not need to support RWX access mode.

- The storage technology must ensure read-after-write consistency. All NAS storage (excluding converged mode/independent mode GlusterFS as it uses an object storage interface) are not recommended for OpenShift Container Platform Registry cluster deployment with production workloads.

- While **hostPath** volumes are configurable for a non-scaled/HA OpenShift Container Platform Registry, they are not recommended for cluster deployment.

#### 5.3.1.2. Scaled registry

In a scaled/HA OpenShift Container Platform registry cluster deployment:

- The preferred storage technology is object storage. The storage technology must support RWX access mode and must ensure read-after-write consistency.

- File storage and block storage are not recommended for a scaled/HA OpenShift Container Platform registry cluster deployment with production workloads.

- All NAS storage (excluding converged mode/independent mode GlusterFS as it uses an object storage interface) are not recommended for OpenShift Container Platform Registry cluster deployment with production workloads.

#### 5.3.1.3. Monitoring

In an OpenShift Container Platform hosted monitoring cluster deployment:

- The preferred storage technology is block storage.

- If you decide to configure file storage, make sure that it follows POSIX standards.

**IMPORTANT**

Testing shows significant unrecoverable corruptions using NFS and, therefore, is not recommended for use.

Other NFS implementations on the marketplace might not have these issues. Contact the individual NFS implementation vendor for more information on any testing that was possibly completed against these OpenShift core components.

#### 5.3.1.4. Logging

In an OpenShift Container Platform hosted logging cluster deployment:

- The preferred storage technology is block storage.

- It is not recommended to use NAS storage (excluding converged mode/independent mode GlusterFS as it uses a block storage interface from iSCSI) for a hosted metrics cluster deployment with production workloads.

**IMPORTANT**

Testing shows issues with using the NFS server on RHEL as storage backend for the container image registry. This includes ElasticSearch for logging storage. Therefore, using NFS to back PVs used by core services is not recommended.

Other NFS implementations on the marketplace might not have these issues. Contact the individual NFS implementation vendor for more information on any testing that was possibly completed against these OpenShift core components.

#### 5.3.1.5. Applications

Application use cases vary from application to application, as described in the following examples:

- Storage technologies that support dynamic PV provisioning have low mount time latencies, and are not tied to nodes to support a healthy cluster.

- Application developers are responsible for knowing and understanding the storage requirements for their application, and how it works with the provided storage to ensure that issues do not occur when an application scales or interacts with the storage layer.

### 5.3.2. Other specific application storage recommendations

- OpenShift Container Platform Internal **etcd**: For the best etcd reliability, the lowest consistent latency storage technology is preferable.
- Databases: Databases (RDBMSs, NoSQL DBs, etc.) tend to perform best with dedicated block storage.

## 5.4. CHOOSING A GRAPH DRIVER

Container runtimes store images and containers in a graph driver (a pluggable storage technology), such as DeviceMapper and OverlayFS. Each has advantages and disadvantages.

For more information about OverlayFS, including supportability and usage caveats, see the Red Hat Enterprise Linux (RHEL) 7 Release Notes for your version.

**Table 5.3. Graph driver comparisons**

| Name | Description | Benefits | Limitations |
|---|---|---|---|
| OverlayFS<br>• overlay<br>• overlay2 | Combines a lower (parent) and upper (child) filesystem and a working directory (on the same filesystem as the child). The lower filesystem is the base image, and when you create new containers, a new upper filesystem is created containing the deltas. | • Faster than Device Mapper at starting and stopping containers. The startup time difference between Device Mapper and Overlay is generally less than one second.<br>• Allows for page cache sharing. | Not POSIX compliant. |
| Device Mapper Thin Provisioning | Uses LVM, Device Mapper, and the dm-thinp kernel module. It differs by removing the loopback device, talking straight to a raw partition (no filesystem). | • There are measurable performance advantages at moderate load and high density.<br>• It gives you per-container limits for capacity (10G by default). | • You have to have a dedicated partition for it.<br>• It is not set up by default in Red Hat Enterprise Linux (RHEL).<br>• All containers and images share the same pool of capacity. It cannot be resized without destroying and re-creating the pool. |

| Name | Description | Benefits | Limitations |
|---|---|---|---|
| Device Mapper loop-lvm | Uses the Device Mapper thin provisioning module (dm-thin-pool) to implement copy-on-write (CoW) snapshots. For each device mapper graph location, thin pool is created based on two block devices, one for data and one for metadata. By default, these block devices are created automatically by using loopback mounts of automatically created sparse files. | It works out of the box, so it is useful for prototyping and development purposes. | <ul><li>Not all Portable Operating System Interface for Unix (POSIX) features work (for example, **O_DIRECT**). Most importantly, this mode is unsupported for production workloads.</li><li>All containers and images share the same pool of capacity. It cannot be resized without destroying and re-creating the pool.</li></ul> |

For better performance, Red Hat strongly recommends using the overlayFS storage driver over Device Mapper. However, if you are already using Device Mapper in a production environment, Red Hat strongly recommends using thin provisioning for container images and container root file systems. Otherwise, always use overlayfs2 for Docker engine or overlayFS for CRI-O.

Using a loop device can affect performance. While you can still continue to use it, the following warning message is logged:

```
devmapper: Usage of loopback devices is strongly discouraged for production use.
Please use `--storage-opt dm.thinpooldev` or use `man docker` to refer to
dm.thinpooldev section.
```

To ease storage configuration, use the **docker-storage-setup** utility, which automates much of the configuration details:

**For Overlay**

1. Edit the ***/etc/sysconfig/docker-storage-setup*** file to specify the device driver:

   ```
   STORAGE_DRIVER=overlay2
   ```

   **NOTE**

   If using CRI-O, specify **STORAGE_DRIVER=overlay**.

   With CRI-O, the default **overlay** storage driver uses the **overlay2** optimizations.

With OverlayFS, if you want to have **imagefs** on a different logical volume, then you must set **CONTAINER_ROOT_LV_NAME** and **CONTAINER _ROOT_LV_MOUNT_PATH**. Setting **CONTAINER_ROOT_LV_MOUNT_PATH** requires **CONTAINER_ROOT_LV_NAME** to be set. For example, **CONTAINER_ROOT_LV_NAME="container-root-lv"**. See Using the Overlay Graph Driver for more information.

2. If you had a separate disk drive dedicated to docker storage (for example, ***/dev/xvdb***), add the following to the ***/etc/sysconfig/docker-storage-setup*** file:

```
DEVS=/dev/xvdb
VG=docker_vg
```

3. Restart the **docker-storage-setup** service:

```
# systemctl restart docker-storage-setup
```

4. To verify that docker is using overlay2, and to monitor disk space use, run the **docker info** command:

```
# docker info | egrep -i 'storage|pool|space|filesystem'
```

**Example Output**

```
Storage Driver: overlay2 1
 Backing Filesystem: extfs
```

1 The **docker info** output when using **overlay2**.

OverlayFS is also supported for container runtimes use cases as of Red Hat Enterprise Linux 7.2, and provides faster start up time and page cache sharing, which can potentially improve density by reducing overall memory utilization.

## For Thinpool

1. Edit the ***/etc/sysconfig/docker-storage-setup*** file to specify the device driver:

```
STORAGE_DRIVER=devicemapper
```

2. If you had a separate disk drive dedicated to docker storage (for example, ***/dev/xvdb***), add the following to the ***/etc/sysconfig/docker-storage-setup*** file:

```
DEVS=/dev/xvdb
VG=docker_vg
```

3. Restart the **docker-storage-setup** service:

```
# systemctl restart docker-storage-setup
```

After the restart, **docker-storage-setup** sets up a volume group named **docker_vg** and creates a thin-pool logical volume. Documentation for thin provisioning on RHEL is available in the LVM Administrator Guide. View the newly created volumes with the **lsblk** command:

```
# lsblk /dev/xvdb
```

**Example Output**

```
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
xvdb 202:16 0 20G 0 disk
└─xvdb1 202:17 0 10G 0 part
  ├─docker_vg-docker--pool_tmeta 253:0 0 12M 0 lvm
  │ └─docker_vg-docker--pool 253:2 0 6.9G 0 lvm
  └─docker_vg-docker--pool_tdata 253:1 0 6.9G 0 lvm
  └─docker_vg-docker--pool 253:2 0 6.9G 0 lvm
```

**NOTE**

Thin-provisioned volumes are not mounted and have no file system (individual containers do have an XFS file system), thus they do not show up in **df** output.

4. To verify that docker is using an LVM thinpool, and to monitor disk space use, run the **docker info** command:

```
# docker info | egrep -i 'storage|pool|space|filesystem'
```

**Example Output**

```
Storage Driver: devicemapper (1)
 Pool Name: docker_vg-docker--pool (2)
 Pool Blocksize: 524.3 kB
 Backing Filesystem: xfs
 Data Space Used: 62.39 MB
 Data Space Total: 6.434 GB
 Data Space Available: 6.372 GB
 Metadata Space Used: 40.96 kB
 Metadata Space Total: 16.78 MB
 Metadata Space Available: 16.74 MB
```

(1) The **docker info** output when using **devicemapper**.

(2) Corresponds to the **VG** you specified in */etc/sysconfig/docker-storage-setup*.

By default, a thin pool is configured to use 40% of the underlying block device. As you use the storage, LVM automatically extends the thin pool up to 100%. This is why the **Data Space Total** value does not match the full size of the underlying LVM device.

In development, docker in Red Hat distributions defaults to a loopback mounted sparse file. To see if your system is using the loopback mode:

```
# docker info|grep loop0
```

**Example Output**

```
Data file: /dev/loop0
```

### 5.4.1. Benefits of using OverlayFS or DeviceMapper with SELinux

The main advantage of the OverlayFS graph is Linux page cache sharing among containers that share an image on the same node. This attribute of OverlayFS leads to reduced input/output (I/O) during container startup (and, thus, faster container startup time by several hundred milliseconds), as well as reduced memory usage when similar images are running on a node. Both of these results are beneficial in many environments, especially those with the goal of optimizing for density and have high container churn rate (such as a build farm), or those that have significant overlap in image content.

Page cache sharing is not possible with DeviceMapper because thin-provisioned devices are allocated on a per-container basis.

NOTE

OverlayFS is the default Docker storage driver for Red Hat Enterprise Linux (RHEL) 7.5 and is supported in 7.3 and later. Set OverlayFS to the default Docker storage configuration on RHEL to improve performance. See the instructions for configuring OverlayFS for use with the Docker container runtime.

### 5.4.2. Comparing the Overlay and Overlay2 graph drivers

OverlayFS is a type of union file system. It allows you to overlay one file system on top of another. Changes are recorded in the upper file system, while the lower file system remains unmodified. This allows multiple users to share a file-system image, such as a container or a DVD-ROM, where the base image is on read-only media.

OverlayFS layers two directories on a single Linux host and presents them as a single directory. These directories are called layers, and the unification process is referred to as a union mount.

OverlayFS uses one of two graph drivers, **overlay** or **overlay2**. As of Red Hat Enterprise Linux 7.2, **overlay**became a supported graph driver . As of Red Hat Enterprise Linux 7.4, **overlay2** became supported. SELinux on the docker daemon became supported in Red Hat Enterprise Linux 7.4. See the Red Hat Enterprise Linux release notes for information on using OverlayFS with your version of RHEL, including supportability and usage caveats.

The **overlay2** driver natively supports up to 128 lower OverlayFS layers but, the **overlay** driver works only with a single lower OverlayFS layer. Because of this capability, the **overlay2** driver provides better performance for layer-related Docker commands, such as **docker build**, and consumes fewer inodes on the backing filesystem.

Because the **overlay** driver works with a single lower OverlayFS layer, you cannot implement multi-layered images as multiple OverlayFS layers. Instead, each image layer is implemented as its own directory under ***/var/lib/docker/overlay***. Hard links are then used as a space-efficient way to reference data shared with lower layers.

Docker recommends using the **overlay2** driver with OverlayFS rather than the **overlay** driver, because it is more efficient in terms of inode utilization.

# CHAPTER 6. OPTIMIZING EPHEMERAL STORAGE

## 6.1. OVERVIEW

**NOTE**

This topic applies only if you enabled the ephemeral storage technology preview. This feature is disabled by default. To enable this feature, see configuring for ephemeral storage.

**NOTE**

Technology Preview releases are not supported with Red Hat production service-level agreements (SLAs) and might not be functionally complete, and Red Hat does not recommend using them for 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 see Red Hat Technology Preview Features Support Scope.

Pods use ephemeral storage for their internal operation such as saving temporary files. The lifetime of this ephemeral storage does not extend beyond the life of the individual pod, and this ephemeral storage cannot be shared across pods.

Prior to OpenShift Container Platform 3.10, ephemeral local storage was exposed to pods through the container’s writable layer, logs directory, and EmptyDir volumes. Issues related to the lack of local storage accounting and isolation include the following:

- Pods do not know how much local storage is available to them.
- Pods cannot request guaranteed local storage.
- Local storage is a best effort resource.
- Pods can get evicted due to other pods filling the local storage, after which, new pods are not admitted until sufficient storage has been reclaimed.

Ephemeral storage is still exposed to pods in the same way, but there are new methods for implementing requests and limits on pods' consumption of ephemeral storage.

**NOTE**

Management of container logs applies only if using CRI-O as the container runtime and file-based logging for logging.

It is important to understand that ephemeral storage is shared among all pods in the system, and that OpenShift Container Platform does not provide any mechanism for guaranteeing any level of service beyond the requests and limits established by the administrator and users. For example, ephemeral storage does not provide any guarantees of throughput, I/O operations per second, or any other measure of storage performance.

## 6.2. GENERAL STORAGE GUIDELINES

A node’s local storage can be broken into primary and secondary partitions. Primary partitions are the only ones you can use for ephemeral local storage. There are two supported primary partitions, root and runtime.

- Root
  Root partitions hold the kubelet’s root directory, **/var/lib/kubelet/** by default, and **/var/log/** directory. You can share this partition among pods, the operating system, and OpenShift Container Platform system daemons. Pods can access this partition by using EmptyDir volumes, container logs, image layers, and container writable layers. OpenShift Container Platform manages shared access and isolation of this partition.

- Runtime
  Runtime partitions are optional partitions you can use for overlay file systems. OpenShift Container Platform attempts to identify and provide shared access along with isolation to this partition. This partition contains container image layers and writable layers. If the runtime partition exists, the **root** partition does not hold any image layer or writable layers.

# CHAPTER 7. NETWORK OPTIMIZATION

## 7.1. OPTIMIZING NETWORK PERFORMANCE

The OpenShift SDN uses OpenvSwitch, virtual extensible LAN (VXLAN) tunnels, OpenFlow rules, and iptables. This network can be tuned by using jumbo frames, network interface cards (NIC) offloads, multi-queue, and ethtool settings.

VXLAN provides benefits over VLANs, such as an increase in networks from 4096 to over 16 million, and layer 2 connectivity across physical networks. This allows for all pods behind a service to communicate with each other, even if they are running on different systems.

VXLAN encapsulates all tunneled traffic in user datagram protocol (UDP) packets. However, this leads to increased CPU utilization. Both these outer- and inner-packets are subject to normal checksumming rules to guarantee data has not been corrupted during transit. Depending on CPU performance, this additional processing overhead can cause a reduction in throughput and increased latency when compared to traditional, non-overlay networks.

Cloud, VM, and bare metal CPU performance can be capable of handling much more than one Gbps network throughput. When using higher bandwidth links such as 10 or 40 Gbps, reduced performance can occur. This is a known issue in VXLAN-based environments and is not specific to containers or OpenShift Container Platform. Any network that relies on VXLAN tunnels will perform similarly because of the VXLAN implementation.

If you are looking to push beyond one Gbps, you can:

- Use Native Container Routing. This option has important operational caveats that do not exist when using OpenShift SDN, such as updating routing tables on a router.
- Evaluate network plug-ins that implement different routing techniques, such as border gateway protocol (BGP).
- Use VXLAN-offload capable network adapters. VXLAN-offload moves the packet checksum calculation and associated CPU overhead off of the system CPU and onto dedicated hardware on the network adapter. This frees up CPU cycles for use by pods and applications, and allows users to utilize the full bandwidth of their network infrastructure.

VXLAN-offload does not reduce latency. However, CPU utilization is reduced even in latency tests.

### 7.1.1. Optimizing the MTU for Your Network

There are two important maximum transmission units (MTUs): the network interface card (NIC) MTU and the SDN overlay's MTU.

The NIC MTU must be less than or equal to the maximum supported value of the NIC of your network. If you are optimizing for throughput, pick the largest possible value. If you are optimizing for lowest latency, pick a lower value.

The SDN overlay's MTU must be less than the NIC MTU by 50 bytes at a minimum. This accounts for the SDN overlay header. So, on a normal ethernet network, set this to 1450. On a jumbo frame ethernet network, set this to 8950.

NOTE

This 50 byte overlay header is relevant to the OpenShift SDN. Other SDN solutions might require the value to be more or less.

To configure the MTU, edit the appropriate node configuration map and modify the following section:

```
networkConfig:
  mtu: 1450 (1)
  networkPluginName: "redhat/openshift-ovs-subnet" (2)
```

1. Maximum transmission unit (MTU) for the pod overlay network.
2. Set to **redhat/openshift-ovs-subnet** for the **ovs-subnet** plug-in, **redhat/openshift-ovs-multitenant** for the **ovs-multitenant** plug-in, or **redhat/openshift-ovs-networkpolicy** for the **ovs-networkpolicy** plug-in. This can also be set to any other CNI-compatible plug-in as well.

NOTE

You must change the MTU size on all masters and nodes that are part of the OpenShift Container Platform SDN. Also, the MTU size of the tun0 interface must be the same across all nodes that are part of the cluster.

## 7.2. CONFIGURING NETWORK SUBNETS

OpenShift Container Platform provides IP address management for both pods and services. The default values allow for:

- A maximum cluster size of 1024 nodes
- Each of the 1024 nodes having a /23 allocated to it (510 usable IPs for pods)
- Around 65,536 IP addresses for services

Under most circumstances, these networks cannot be changed after deployment. So, planning ahead for growth is important.

Restrictions for resizing networks are document in the Configuring SDN documentation.

To plan for a larger environment, the following are suggested values to consider adding to the **[OSE3:vars]** section in your Ansible inventory file:

```
[OSE3:vars]
osm_cluster_network_cidr=10.128.0.0/10
```

This will allow for 8192 nodes, each with 510 usable IP addresses.

See the supportability limits in the OpenShift Container Platform documentation for node/pod limits for the version of software you are installing.

## 7.3. OPTIMIZING IPSEC

Because encrypting and decrypting node hosts uses CPU power, performance is affected both in throughput and CPU usage on the nodes when encryption is enabled, regardless of the IP security system being used.

IPSec encrypts traffic at the IP payload level, before it hits the NIC, protecting fields that would otherwise be used for NIC offloading. This means that some NIC acceleration features may not be usable when IPSec is enabled and will lead to decreased throughput and increased CPU usage.

# CHAPTER 8. ROUTING OPTIMIZATION

## 8.1. SCALING OPENSHIFT CONTAINER PLATFORM HAPROXY ROUTER

### 8.1.1. Baseline Performance

The OpenShift Container Platform router is the ingress point for all external traffic destined for OpenShift Container Platform services.

When evaluating a single HAProxy router performance in terms of HTTP requests handled per second, the performance varies depending on many factors. In particular:

- HTTP keep-alive/close mode,
- route type
- TLS session resumption client support
- number of concurrent connections per target route
- number of target routes
- backend server page size
- underlying infrastructure (network/SDN solution, CPU, and so on)

While performance in your specific environment will vary, our lab tests on a public cloud instance of size 4 vCPU/16GB RAM, a single HAProxy router handling 100 routes terminated by backends serving 1kB static pages is able to handle the following number of transactions per second.

In HTTP keep-alive mode scenarios:

| Encryption | ROUTER_THREADS unset | ROUTER_THREADS=4 |
|---|---|---|
| none | 23681 | 24327 |
| edge | 14981 | 22768 |
| passthrough | 34358 | 34331 |
| re-encrypt | 13288 | 24605 |

In HTTP close (no keep-alive) scenarios:

| Encryption | ROUTER_THREADS unset | ROUTER_THREADS=4 |
|---|---|---|
| none | 3245 | 4527 |
| edge | 1910 | 3043 |

| Encryption | ROUTER_THREADS unset | ROUTER_THREADS=4 |
|---|---|---|
| passthrough | 3408 | 3922 |
| re-encrypt | 1333 | 2239 |

TLS session resumption was used for encrypted routes. With HTTP keep-alive, a single HAProxy router is capable of saturating 1 Gbit NIC at page sizes as small as 8 kB.

When running on bare metal with modern processors, you can expect roughly twice the performance of the public cloud instance above. This overhead is introduced by the virtualization layer in place on public clouds and holds mostly true for private cloud-based virtualization as well. The following table is a guide on how many applications to use behind the router:

| Number of applications | Application type |
|---|---|
| 5-10 | static file/web server or caching proxy |
| 100-1000 | applications generating dynamic content |

In general, HAProxy can support routes for 5 to 1000 applications, depending on the technology in use. Router performance might be limited by the capabilities and performance of the applications behind it, such as language or static versus dynamic content.

Router sharding should be used to serve more routes towards applications and help horizontally scale the routing tier.

### 8.1.2. Performance Optimizations

#### 8.1.2.1. Setting the Maximum Number of Connections

One of the most important tunable parameters for HAProxy scalability is the **maxconn** parameter, which sets the maximum per-process number of concurrent connections to a given number. Adjust this parameter by editing the **ROUTER_MAX_CONNECTIONS** environment variable in the OpenShift Container Platform HAProxy router's deployment configuration file.

NOTE

A connection includes the frontend and internal backend. This counts as two connections. Be sure to set **ROUTER_MAX_CONNECTIONS** to double than the number of connections you intend to create.

#### 8.1.2.2. CPU and Interrupt Affinity

In OpenShift Container Platform, the HAProxy router runs as a single process. The OpenShift Container Platform HAProxy router typically performs better on a system with fewer but high frequency cores, rather than on an symmetric multiprocessing (SMP) system with a high number of lower frequency cores.

Pinning the HAProxy process to one CPU core and the network interrupts to another CPU core tends to increase network performance. Having processes and interrupts on the same non-uniform memory access (NUMA) node helps avoid memory accesses by ensuring a shared L3 cache. However, this level of control is generally not possible on a public cloud environment. On bare metal hosts, **irqbalance** automatically handles peripheral component interconnect (PCI) locality and NUMA affinity for interrupt request lines (IRQs). On a cloud environment, this level of information is generally not provided to the operating system.

CPU pinning is performed either by **taskset** or by using HAProxy's **cpu-map** parameter. This directive takes two arguments: the process ID and the CPU core ID. For example, to pin HAProxy process **1** onto CPU core **0**, add the following line to the global section of HAProxy's configuration file:

```
cpu-map 1 0
```

To modify the HAProxy configuration file, refer to Deploying a Customized HAProxy Router .

### 8.1.2.3. Increasing the Number of Threads

The HAProxy router comes with support for multithreading in OpenShift Container Platform. On a multiple CPU core system, increasing the number of threads can help the performance, especially when terminating SSL on the router.

To specify the number of threads for the HAProxy router, refer to Enable HAProxy Threading and Router Environment Variables.

### 8.1.2.4. Impacts of Buffer Increases

The OpenShift Container Platform HAProxy router request buffer configuration limits the size of headers in incoming requests and responses from applications. The HAProxy parameter **tune.bufsize** can be increased to allow processing of larger headers and to allow applications with very large cookies to work, such as those accepted by load balancers provided by many public cloud providers. However, this affects the total memory use, especially when large numbers of connections are open. With very large numbers of open connections, the memory usage will be nearly proportionate to the increase of this tunable parameter.

### 8.1.2.5. Optimizations for HAProxy Reloads

Long-lasting connections, such as WebSocket connections, combined with long client/server HAProxy timeouts and short HAProxy reload intervals, can cause instantiation of many HAProxy processes. These processes must handle old connections, which were started before the HAProxy configuration reload. A large number of these processes is undesirable, as it will exert unnecessary load on the system and can lead to issues, such as out of memory conditions.

Router environment variables affecting this behavior are **ROUTER_DEFAULT_TUNNEL_TIMEOUT**, **ROUTER_DEFAULT_CLIENT_TIMEOUT**, **ROUTER_DEFAULT_SERVER_TIMEOUT**, and **RELOAD_INTERVAL** in particular.

# CHAPTER 9. SCALING CLUSTER METRICS

## 9.1. OVERVIEW

OpenShift Container Platform exposes metrics that can be collected and stored in back-ends by Heapster. As an OpenShift Container Platform administrator, you can view containers and components metrics in one user interface. These metrics are also used by horizontal pod autoscalers in order to determine when and how to scale.

This topic provides information for scaling the metrics components.

**NOTE**

Autoscaling the metrics components, such as Hawkular and Heapster, is not supported by OpenShift Container Platform.

## 9.2. RECOMMENDATIONS FOR OPENSHIFT CONTAINER PLATFORM

- Run metrics pods on dedicated OpenShift Container Platform infrastructure nodes.
- Use persistent storage when configuring metrics. Set **USE_PERSISTENT_STORAGE=true**.
- Keep the **METRICS_RESOLUTION=30** parameter in OpenShift Container Platform metrics deployments. Using a value lower than the default value of **30** for **METRICS_RESOLUTION** is not recommended. When using the Ansible metrics installation procedure, this is the **openshift_metrics_resolution** parameter.
- Closely monitor OpenShift Container Platform nodes with host metrics pods to detect early capacity shortages (CPU and memory) on the host system. These capacity shortages can cause problems for metrics pods.
- In OpenShift Container Platform version 3.7 testing, test cases up to 25,000 pods were monitored in a OpenShift Container Platform cluster.

## 9.3. CAPACITY PLANNING FOR CLUSTER METRICS

In tests performed with 210 and 990 OpenShift Container Platform nodes, where 10500 pods and 11000 pods were monitored respectively, the Cassandra database grew at the speed shown in the table below:

**Table 9.1. Cassandra Database storage requirements based on number of nodes/pods in the cluster**

| Number of Nodes | Number of Pods | Cassandra Storage growth speed | Cassandra storage growth per day | Cassandra storage growth per week |
|---|---|---|---|---|
| 210 | 10500 | 500 MB per hour | 15 GB | 75 GB |
| 990 | 11000 | 1 GB per hour | 30 GB | 210 GB |

In the above calculation, approximately 20 percent of the expected size was added as overhead to ensure that the storage requirements do not exceed calculated value.

If the **METRICS_DURATION** and **METRICS_RESOLUTION** values are kept at the default ( **7** days and **15** seconds respectively), it is safe to plan Cassandra storage size requirements for week, as in the values above.

**WARNING**

Because OpenShift Container Platform metrics uses the Cassandra database as a datastore for metrics data, if **USE_PERSISTENT_STORAGE=true** is set during the metrics set up process, **PV** will be on top in the network storage, with NFS as the default. However, using network storage in combination with Cassandra is not recommended.

If you use a Cassandra database as a datastore for metrics data, see the Cassandra documentation for their recommendations.

## 9.4. SCALING OPENSHIFT CONTAINER PLATFORM METRICS PODS

One set of metrics pods (Cassandra/Hawkular/Heapster) is able to monitor at least 25,000 pods.

**CAUTION**

Pay attention to system load on nodes where OpenShift Container Platform metrics pods run. Use that information to determine if it is necessary to scale out a number of OpenShift Container Platform metrics pods and spread the load across multiple OpenShift Container Platform nodes. Scaling OpenShift Container Platform metrics heapster pods is not recommended.

### 9.4.1. Prerequisites

If persistent storage was used to deploy OpenShift Container Platform metrics, then you must create a persistent volume (PV) for the new Cassandra pod to use before you can scale out the number of OpenShift Container Platform metrics Cassandra pods. However, if Cassandra was deployed with dynamically provisioned PVs, then this step is not necessary.

### 9.4.2. Scaling the Cassandra Components

Cassandra nodes use persistent storage. Therefore, scaling up or down is not possible with replication controllers.

Scaling a Cassandra cluster requires modifying the **openshift_metrics_cassandra_replicas** variable and re-running the deployment. By default, the Cassandra cluster is a single-node cluster.

To scale up the number of OpenShift Container Platform metrics hawkular pods to two replicas, run:

```
# oc scale -n openshift-infra --replicas=2 rc hawkular-metrics
```

Alternatively, update your inventory file and re-run the deployment.

NOTE

If you add a new node to or remove an existing node from a Cassandra cluster, the data stored in the cluster rebalances across the cluster.

To scale down:

1. If remotely accessing the container, run the following for the Cassandra node you want to remove:

   ```
   $ oc exec -it <hawkular-cassandra-pod> nodetool decommission
   ```

   If locally accessing the container, run the following instead:

   ```
   $ oc rsh <hawkular-cassandra-pod> nodetool decommission
   ```

   This command can take a while to run since it copies data across the cluster. You can monitor the decommission progress with **nodetool netstats -H**.

2. Once the previous command succeeds, scale down the **rc** for the Cassandra instance to **0**.

   ```
   # oc scale -n openshift-infra --replicas=0 rc <hawkular-cassandra-rc>
   ```

   This will remove the Cassandra pod.

IMPORTANT

If the scale down process completed and the existing Cassandra nodes are functioning as expected, you can also delete the **rc** for this Cassandra instance and its corresponding persistent volume claim (PVC). Deleting the PVC can permanently delete any data associated with this Cassandra instance, so if the scale down did not fully and successfully complete, you will not be able to recover the lost data.

# CHAPTER 10. SCALING CLUSTER MONITORING OPERATOR

## 10.1. OVERVIEW

OpenShift Container Platform exposes metrics that can be collected and stored in back-ends by the **cluster-monitoring-operator**. As an OpenShift Container Platform administrator, you can view system resources, containers and components metrics in one dashboard interface, Grafana.

This topic provides information on scaling the cluster monitoring operator.

If you want to use Prometheus with persistent storage, you must set the **openshift_cluster_monitoring_operator_prometheus_storage_enabled** variable in your Ansible inventory file to **true**.

## 10.2. RECOMMENDATIONS FOR OPENSHIFT CONTAINER PLATFORM

- Use at least three infrastructure (infra) nodes.
- Use at least three **openshift-container-storage** nodes with non-volatile memory express (NVMe) drives.
- Use persistent block storage, such as OpenShift Container Storage (OCS) Block .

## 10.3. CAPACITY PLANNING FOR CLUSTER MONITORING OPERATOR

Various tests were performed for different scale sizes. The Prometheus database grew, as reflected in the table below.

### NOTE

The Prometheus storage requirements below are not prescriptive. Higher resource consumption might be observed in your cluster depending on workload activity and resource use.

**Table 10.1. Prometheus Database storage requirements based on number of nodes/pods in the cluster**

| Number of Nodes | Number of Pods | Prometheus storage growth per day | Prometheus storage growth per 15 days | RAM Space (per scale size) | Network (per tsdb chunk) |
|---|---|---|---|---|---|
| 50 | 1800 | 6.3 GB | 94 GB | 6 GB | 16 MB |
| 100 | 3600 | 13 GB | 195 GB | 10 GB | 26 MB |
| 150 | 5400 | 19 GB | 283 GB | 12 GB | 36 MB |
| 200 | 7200 | 25 GB | 375 GB | 14 GB | 46 MB |

In the above calculation, approximately 20 percent of the expected size was added as overhead to ensure that the storage requirements do not exceed the calculated value.

The above calculation was developed for the default OpenShift Container Platform **cluster-monitoring-operator**. For higher scale, edit the **openshift_cluster_monitoring_operator_prometheus_storage_capacity** variable in the Ansible inventory file, which defaults to **50Gi**.

NOTE

CPU utilization has minor impact. The ratio is approximately 1 core out of 40 per 50 nodes and 1800 pods.

### 10.3.1. Lab Environment

All experiments were performed in an OpenShift Container Platform on OpenStack environment:

- Infra nodes (VMs) - 40 cores, 157 GB RAM.
- CNS nodes (VMs) - 16 cores, 62 GB RAM, NVMe drives.

### 10.3.2. Prerequisites

Based on your scale destination, compute and set the relevant PV size for the Prometheus data store. Since the default Prometheus pods replicas is 2, for 100 nodes with 3600 pods you will need 188 GB.

For example:

```
195 GB (space per 15 days ) * 2 (pods) = 390 GB free
```

Based on this equation, set **openshift_cluster_monitoring_operator_prometheus_storage_capacity=195Gi**.

# CHAPTER 11. TESTED MAXIMUMS PER CLUSTER

Consider the following tested cluster object maximums when you plan your OpenShift Container Platform cluster.

These guidelines are based on the largest possible cluster. For smaller clusters, the maximums are proportionally lower. There are many factors that influence the stated thresholds, including the etcd version or storage data format.

In most cases, exceeding these numbers results in lower overall performance. It does not necessarily mean that the cluster will fail.

Tested Cloud Platforms for OpenShift Container Platform 3.x: Red Hat OpenStack, Amazon Web Services, and Microsoft Azure.

## 11.1. OPENSHIFT CONTAINER PLATFORM TESTED CLUSTER MAXIMUMS FOR MAJOR RELEASES

| Maximum Type | 3.x Tested Maximum |
|---|---|
| Number of Nodes | 2,000 |
| Number of Pods [1] | 150,000 |
| Number of Pods per Node | 250 |
| Number of Pods per Core | There is no default value. |
| Number of Namespaces | 10,000 |
| Number of Builds: Pipeline Strategy | 10,000 (Default pod RAM 512Mi) |
| Number of Pods per Namespace [2] | 25,000 |
| Number of Services [3] | 10,000 |
| Number of Services per Namespace | 5,000 |
| Number of Back-ends per Service | 5,000 |
| Number of Deployments per Namespace [2] | 2,000 |

1. The Pod count displayed here is the number of test Pods. The actual number of Pods depends on the application's memory, CPU, and storage requirements.

2. There are a number of control loops in the system that need to iterate over all objects in a given namespace as a reaction to some changes in state. Having a large number of objects of a given type in a single namespace can make those loops expensive and slow down processing given

state changes. The maximum assumes that the system has enough CPU, memory, and disk to satisfy the application requirements.

3. Each Service port and each Service back-end has a corresponding entry in iptables. The number of back-ends of a given Service impact the size of the endpoints objects, which impacts the size of data that is being sent all over the system.

## 11.2. OPENSHIFT CONTAINER PLATFORM TESTED CLUSTER MAXIMUMS

| Maximum Type | 3.7 Tested Maximum | 3.9 Tested Maximum | 3.10 Tested Maximum | 3.11 Tested Maximum |
|---|---|---|---|---|
| Number of Nodes | 2,000 | 2,000 | 2,000 | 2,000 |
| Number of Pods [1] | 120,000 | 120,000 | 150,000 | 150,000 |
| Number of Pods per Node | 250 | 250 | 250 | 250 |
| Number of Pods per Core | 10 is the default value. | 10 is the default value. | There is no default value. | There is no default value. |
| Number of Namespaces | 10,000 | 10,000 | 10,000 | 10,000 |
| Number of Builds: Pipeline Strategy | N/A | 10,000 (Default pod RAM 512Mi) | 10,000 (Default pod RAM 512Mi) | 10,000 (Default pod RAM 512Mi) |
| Number of Pods per Namespace [2] | 3,000 | 3,000 | 3,000 | 25,000 |
| Number of Services [3] | 10,000 | 10,000 | 10,000 | 10,000 |
| Number of Services per Namespace | N/A | N/A | 5,000 | 5,000 |
| Number of Back-ends per Service | 5,000 | 5,000 | 5,000 | 5,000 |
| Number of Deployments per Namespace [2] | 2,000 | 2,000 | 2,000 | 2,000 |

1. The Pod count displayed here is the number of test Pods. The actual number of Pods depends on the application's memory, CPU, and storage requirements.

2. There are a number of control loops in the system that need to iterate over all objects in a given namespace as a reaction to some changes in state. Having a large number of objects of a given type in a single namespace can make those loops expensive and slow down processing given state changes. The maximum assumes that the system has enough CPU, memory, and disk to satisfy the application requirements.

3. Each Service port and each Service back-end has a corresponding entry in iptables. The number of back-ends of a given service impact the size of the endpoints objects, which impacts the size of data that is being sent all over the system.

### 11.2.1. Route Maximums

In OpenShift Container Platform 3.11.53, router tests were completed in a 3-node environment on Amazon Web Services (AWS). There were 100 HTTP routes, specifically 100 back-end Nginx pods, with **keepalive** set to **100**. The results were:

- 1 connection per target route = 24,327 requests per second
- 40 connections per target route = 20,729 requests per second
- 200 connections per target route = 17,253 requests per second

## 11.3. ENVIRONMENT AND CONFIGURATION ON WHICH OPENSHIFT CONTAINER PLATFORM CLUSTER MAXIMUMS ARE TESTED

Infrastructure as a service provider: OpenStack

| Node | vCPU | RAM(MiB) | Disk size(GiB) | pass-through disk | Count |
|---|---|---|---|---|---|
| Master/Etcd [1] | 16 | 124672 | 128 | Yes, NVMe | 3 |
| Infra [2] | 40 | 163584 | 256 | Yes, NVMe | 3 |
| Cluster DNS | 1 | 1740 | 71 | No | 1 |
| Load Balancer | 4 | 16128 | 96 | No | 1 |
| Container Native Storage [3] | 16 | 65280 | 200 | Yes, NVMe | 3 |
| Bastion [4] | 16 | 65280 | 200 | No | 1 |
| Worker | 2 | 7936 | 96 | No | 2000 |

1. The master/etcd nodes are backed by NVMe disks as etcd is I/O intensive and latency sensitive.

2. Infra nodes host the Router, Registry, Logging and Monitoring and are backed by NVMe disks.

3. Container Native Storage or Ceph storage nodes are backed by NVMe disks.
4. The Bastion node is part of the OpenShift Container Platform network and is used to orchestrate the performance and scale tests.

## 11.4. PLANNING YOUR ENVIRONMENT ACCORDING TO CLUSTER MAXIMUMS

**IMPORTANT**

Oversubscribing the physical resources on a node affects resource guarantees the Kubernetes scheduler makes during pod placement. Learn what measures you can take to avoid memory swapping.

Some of the tested maximums are stretched only in a single dimension, so they might vary when a lot of objects are running on the cluster.

The numbers noted in this documentation are based on Red Hat's test methodology, setup, configuration, and tunings. These numbers can vary based on your own individual setup and environments.

While planning your environment, determine how many pods are expected to fit per node:

```
Maximum Pods per Cluster / Expected Pods per Node = Total Number of Nodes
```

The number of pods expected to fit on a node is dependent on the application itself. Consider the application's memory, CPU, and storage requirements.

### Example Scenario

If you want to scope your cluster for 2200 pods per cluster, you would need at least nine nodes, assuming that there are 250 maximum pods per node:

```
2200 / 250 = 8.8
```

If you increase the number of nodes to 20, then the pod distribution changes to 110 pods per node:

```
2200 / 20 = 110
```

## 11.5. PLANNING YOUR ENVIRONMENT ACCORDING TO APPLICATION REQUIREMENTS

Consider an example application environment:

| Pod Type | Pod Quantity | Max Memory | CPU Cores | Persistent Storage |
|---|---|---|---|---|
| apache | 100 | 500MB | 0.5 | 1GB |
| node.js | 200 | 1GB | 1 | 1GB |

| Pod Type | Pod Quantity | Max Memory | CPU Cores | Persistent Storage |
|---|---|---|---|---|
| postgresql | 100 | 1GB | 2 | 10GB |
| JBoss EAP | 100 | 1GB | 1 | 1GB |

Extrapolated requirements: 550 CPU cores, 450GB RAM, and 1.4TB storage.

Instance size for nodes can be modulated up or down, depending on your preference. Nodes are often resource overcommitted. In this deployment scenario, you can choose to run additional smaller nodes or fewer larger nodes to provide the same amount of resources. Factors such as operational agility and cost-per-instance should be considered.

| Node Type | Quantity | CPUs | RAM (GB) |
|---|---|---|---|
| Nodes (option 1) | 100 | 4 | 16 |
| Nodes (option 2) | 50 | 8 | 32 |
| Nodes (option 3) | 25 | 16 | 64 |

Some applications lend themselves well to overcommitted environments, and some do not. Most Java applications and applications that use huge pages are examples of applications that would not allow for overcommitment. That memory can not be used for other applications. In the example above, the environment would be roughly 30 percent overcommitted, a common ratio.

# CHAPTER 12. USING CLUSTER LOADER

## 12.1. WHAT CLUSTER LOADER DOES

Cluster Loader is a tool that deploys large numbers of various objects to a cluster, which creates user-defined cluster objects. Build, configure, and run Cluster Loader to measure performance metrics of your OpenShift Container Platform deployment at various cluster states.

## 12.2. INSTALLING CLUSTER LOADER

Cluster Loader is included in the **atomic-openshift-tests** package. To install it, run:

```
$ yum install atomic-openshift-tests
```

After installation, the test executable ***extended.test*** is located in ***/usr/libexec/atomic-openshift/extended.test***.

## 12.3. RUNNING CLUSTER LOADER

1. Set the **KUBECONFIG** variable to the location of the administrator **kubeconfig**:

   ```
   $ export KUBECONFIG=${KUBECONFIG-$HOME/.kube/config}
   ```

2. Execute Cluster Loader using the built-in test configuration, which deploys five template builds and waits for them to complete:

   ```
   $ cd /usr/libexec/atomic-openshift/
   ```

   ```
   $ ./extended.test --ginkgo.focus="Load cluster"
   ```

   Alternatively, execute Cluster Loader with a user-defined configuration by adding the flag for **--viper-config**:

   ```
   $ ./extended.test --ginkgo.focus="Load cluster" --viper-config=config/test 1
   ```

   1 In this example, there is a subdirectory called ***config/*** with a configuration file called ***test.yml***. In the command line, exclude the extension of the configuration file, as the tool will automatically determine the file type and extension.

## 12.4. CONFIGURING CLUSTER LOADER

Create multiple namespaces (projects), which contain multiple templates or pods.

Locate the configuration files for Cluster Loader in the ***config/*** subdirectory. The pod files and template files referenced in these configuration examples are found in the ***content/*** subdirectory.

### 12.4.1. Configuration Fields

**Table 12.1. Top-level Cluster Loader Fields**

| Field | Description |
|---|---|
| **cleanup** | Set to **true** or **false**. One definition per configuration. If set to **true**, **cleanup** will delete all namespaces (projects) created by Cluster Loader at the end of the test. |
| **projects** | A sub-object with one or many definition(s). Under **projects**, each namespace to create is defined and **projects** has several mandatory subheadings. |
| **tuningsets** | A sub-object with one definition per configuration. **tuningsets** allows the user to define a tuning set to add configurable timing to project or object creation (pods, templates, and so on). |
| **sync** | An optional sub-object with one definition per configuration. Adds synchronization possibilities during object creation. |

Table 12.2. Fields under **projects**

| Field | Description |
|---|---|
| **num** | An integer. One definition of the count of how many projects to create. |
| **basename** | A string. One definition of the base name for the project. The count of identical namespaces will be appended to **Basename** to prevent collisions. |
| **tuning** | A string. One definition of what tuning set you want to apply to the objects, which you deploy inside this namespace. |
| **ifexists** | A string containing either **reuse** or **delete**. Defines what the tool does if it finds a project or namespace that has the same name of the project or namespace it creates during execution. |
| **configmaps** | A list of key-value pairs. The key is the ConfigMap name and the value is a path to a file from which you create the ConfigMap. |
| **secrets** | A list of key-value pairs. The key is the secret name and the value is a path to a file from which you create the secret. |
| **pods** | A sub-object with one or many definition(s) of pods to deploy. |

| Field | Description |
|---|---|
| **templates** | A sub-object with one or many definition(s) of templates to deploy. |

Table 12.3. Fields under **pods** and **templates**

| Field | Description |
|---|---|
| **total** | This field is not used. |
| **num** | An integer. The number of pods or templates to deploy. |
| **image** | A string. The container image URL to a repository where it can be pulled. |
| **basename** | A string. One definition of the base name for the template (or pod) that you want to create. |
| **file** | A string. The path to a local file, which is either a PodSpec or template to be created. |
| **parameters** | Key-value pairs. Under **parameters**, you can specify a list of values to override in the pod or template. |

Table 12.4. Fields under **tuningsets**

| Field | Description |
|---|---|
| **name** | A string. The name of the tuning set which will match the name specified when defining a tuning in a project. |
| **pods** | A sub-object identifying the **tuningsets** that will apply to pods. |
| **templates** | A sub-object identifying the **tuningsets** that will apply to templates. |

Table 12.5. Fields under **tuningsets pods** or **tuningsets templates**

| Field | Description |
|---|---|
| **stepping** | A sub-object. A stepping configuration used if you want to create an object in a step creation pattern. |

| Field | Description |
|---|---|
| **rate_limit** | A sub-object. A rate-limiting tuning set configuration to limit the object creation rate. |

Table 12.6. Fields under **tuningsets pods** or **tuningsets templates, stepping**

| Field | Description |
|---|---|
| **stepsize** | An integer. How many objects to create before pausing object creation. |
| **pause** | An integer. How many seconds to pause after creating the number of objects defined in **stepsize**. |
| **timeout** | An integer. How many seconds to wait before failure if the object creation is not successful. |
| **delay** | An integer. How many milliseconds (ms) to wait between creation requests. |

Table 12.7. Fields under **sync**

| Field | Description |
|---|---|
| **server** | A sub-object with **enabled** and **port** fields. The boolean **enabled** defines whether to start a HTTP server for pod synchronization. The integer **port** defines the HTTP server port to listen on (**9090** by default). |
| **running** | A boolean. Wait for pods with labels matching **selectors** to go into **Running** state. |
| **succeeded** | A boolean. Wait for pods with labels matching **selectors** to go into **Completed** state. |
| **selectors** | A list of selectors to match pods in **Running** or **Completed** states. |
| **timeout** | A string. The synchronization timeout period to wait for pods in **Running** or **Completed** states. For values that are not **0**, use units: [ns\|us\|ms\|s\|m\|h]. |

### 12.4.2. Example Cluster Loader Configuration File

Cluster Loader's configuration file is a basic YAML file:

```
provider: local 1
ClusterLoader:
  cleanup: true
  projects:
    - num: 1
      basename: clusterloader-cakephp-mysql
      tuning: default
      ifexists: reuse
      templates:
        - num: 1
          file: ./examples/quickstarts/cakephp-mysql.json

    - num: 1
      basename: clusterloader-dancer-mysql
      tuning: default
      ifexists: reuse
      templates:
        - num: 1
          file: ./examples/quickstarts/dancer-mysql.json

    - num: 1
      basename: clusterloader-django-postgresql
      tuning: default
      ifexists: reuse
      templates:
        - num: 1
          file: ./examples/quickstarts/django-postgresql.json

    - num: 1
      basename: clusterloader-nodejs-mongodb
      tuning: default
      ifexists: reuse
      templates:
        - num: 1
          file: ./examples/quickstarts/nodejs-mongodb.json

    - num: 1
      basename: clusterloader-rails-postgresql
      tuning: default
      templates:
        - num: 1
          file: ./examples/quickstarts/rails-postgresql.json

  tuningsets: 2
    - name: default
      pods:
        stepping: 3
          stepsize: 5
          pause: 0 s
        rate_limit: 4
          delay: 0 ms
```

1 Optional setting for end-to-end tests. Set to **local** to avoid extra log messages.

2 The tuning sets allow rate limiting and stepping, the ability to create several batches of pods while pausing in between sets. Cluster Loader monitors completion of the previous step before

pausing in between sets. Cluster Loader monitors completion of the previous step before continuing.

3 Stepping will pause for **M** seconds after each **N** objects are created.

4 Rate limiting will wait **M** milliseconds between the creation of objects.

## 12.5. KNOWN ISSUES

If the **IDENTIFIER** parameter is not defined in user templates, template creation fails with **error: unknown parameter name "IDENTIFIER"**. If you deploy templates, add this parameter to your template to avoid this error:

```
{
  "name": "IDENTIFIER",
  "description": "Number to append to the name of resources",
  "value": "1"
}
```

If you deploy pods, adding the parameter is unnecessary.

# CHAPTER 13. USING CPU MANAGER

## 13.1. WHAT CPU MANAGER DOES

CPU Manager manages groups of CPUs and constrains workloads to specific CPUs.

CPU Manager is useful for workloads that have some of these attributes:

- Require as much CPU time as possible.
- Are sensitive to processor cache misses.
- Are low-latency network applications.
- Coordinate with other processes and benefit from sharing a single processor cache.

## 13.2. SETTING UP CPU MANAGER

To set up CPU Manager:

1. Optionally, label a node:

```
# oc label node perf-node.example.com cpumanager=true
```

2. Enable CPU manager support on the target node:

```
# oc edit configmap <name> -n openshift-node
```

For example:

```
# oc edit cm node-config-compute -n openshift-node
```

**Example Output**

```
...
kubeletArguments:
...
  feature-gates:
  - CPUManager=true
  cpu-manager-policy:
  - static
  cpu-manager-reconcile-period:
  - 5s
  system-reserved: 1
  - cpu=500m
```

```
# systemctl restart atomic-openshift-node
```

1 **system-reserved** is a required setting. The value might need to be adjusted depending on your environment.

3. Create a pod that requests a core or multiple cores. Both limits and requests must have their CPU value set to a whole integer. That is the number of cores that will be dedicated to this pod:

```
# cat cpumanager.yaml
```

**Example Output**

```
apiVersion: v1
kind: Pod
metadata:
  generateName: cpumanager-
spec:
  containers:
  - name: cpumanager
    image: gcr.io/google_containers/pause-amd64:3.0
    resources:
      requests:
        cpu: 1
        memory: "1G"
      limits:
        cpu: 1
        memory: "1G"
  nodeSelector:
    cpumanager: "true"
```

4. Create the pod:

```
# oc create -f cpumanager.yaml
```

5. Verify that the pod is scheduled to the node that you labeled:

```
# oc describe pod cpumanager
```

**Example Output**

```
Name:         cpumanager-4gdtn
Namespace:    test
Node:         perf-node.example.com/172.31.62.105
...
    Limits:
      cpu:     1
      memory:  1G
    Requests:
      cpu:        1
      memory:     1G
...
QoS Class:       Guaranteed
Node-Selectors:  cpumanager=true
                 region=primary
```

6. Verify that the **cgroups** are set up correctly. Get the PID of the pause process:

```
# systemd-cgls -l
```

#### Example Output

```
├─1 /usr/lib/systemd/systemd --system --deserialize 20
├─kubepods.slice
│ ├─kubepods-pod0ec1ab8b_e1c4_11e7_bb22_027b30990a24.slice
│ │ ├─docker-
b24e29bc4021064057f941dc5f3538595c317d294f2c8e448b5e61a29c026d1c.scope
│ │ │ └─44216 /pause
```

Pods of QoS tier **Guaranteed** are placed within the **kubepods.slice**. Pods of other QoS tiers end up in child **cgroups** of **kubepods**.

```
# cd /sys/fs/cgroup/cpuset/kubepods.slice/kubepods-
pod0ec1ab8b_e1c4_11e7_bb22_027b30990a24.slice/docker-
b24e29bc4021064057f941dc5f3538595c317d294f2c8e448b5e61a29c026d1c.scope
# for i in `ls cpuset.cpus tasks` ; do echo -n "$i "; cat $i ; done
```

#### Example Output

```
cpuset.cpus 2
tasks 44216
```

7. Check the allowed CPU list for the task:

```
# grep ^Cpus_allowed_list /proc/44216/status
```

#### Example Output

```
Cpus_allowed_list:      2
```

8. Verify that another pod (in this case, the pod in the **burstable** QoS tier) on the system can not run on the core allocated for the **Guaranteed** pod:

```
# cat /sys/fs/cgroup/cpuset/kubepods.slice/kubepods-burstable.slice/kubepods-burstable-
podbe76ff22_dead_11e7_b99e_027b30990a24.slice/docker-
da621bea7569704fc39f84385a179923309ab9d832f6360cccbff102e73f9557.scope/cpuset.cpus

0-1,3
```

```
# oc describe node perf-node.example.com
```

#### Example Output

```
...
Capacity:
 cpu:     4
 memory:  16266720Ki
 pods:    40
Allocatable:
 cpu:     3500m
 memory:  16164320Ki
 pods:    40
```

```
---
  Namespace               Name                 CPU Requests  CPU Limits  Memory Requests
Memory Limits
  ---------               ----                 ------------  ----------  ---------------  -------------
  test                 cpumanager-4gdtn          1 (28%)      1 (28%)     1G (6%)         1G (6%)
  test                 cpumanager-hczts          1 (28%)      1 (28%)     1G (6%)         1G (6%)
  test                 cpumanager-r9wrq          1 (28%)      1 (28%)     1G (6%)         1G (6%)
...
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests  CPU Limits  Memory Requests  Memory Limits
  ------------  ----------  ---------------  -------------
  3 (85%)       3 (85%)     5437500k (32%)   9250M (55%)
```

This VM has four CPU cores. You set **system-reserved** to 500 millicores, meaning half of one core is subtracted from the total capacity of the node to arrive at the **Node Allocatable** amount.

You can see that **Allocatable CPU** is 3500 millicores. This means we can run three of our CPU manager pods since each will take one whole core. A whole core is equivalent to 1000 millicores.

If you try to schedule a fourth pod, the system will accept the pod, but it will never be scheduled:

```
# oc get pods --all-namespaces |grep test
```

### Example Output

```
test          cpumanager-4gdtn          1/1       Running          0         8m
test          cpumanager-hczts          1/1       Running          0         8m
test          cpumanager-nb9d5          0/1       Pending          0         8m
test          cpumanager-r9wrq          1/1       Running          0         8m
```

# CHAPTER 14. MANAGING HUGE PAGES

## 14.1. WHAT HUGE PAGES DO

Memory is managed in blocks known as pages. On most systems, a page is 4Ki. 1Mi of memory is equal to 256 pages; 1Gi of memory is 262,144 pages, and so on. CPUs have a built-in memory management unit that manages a list of these pages in hardware. The Translation Lookaside Buffer (TLB) is a small hardware cache of virtual-to-physical page mappings. If the virtual address passed in a hardware instruction can be found in the TLB, the mapping can be determined quickly. If not, a TLB miss occurs, and the system falls back to slower, software-based address translation, resulting in performance issues. Since the size of the TLB is fixed, the only way to reduce the chance of a TLB miss is to increase the page size.

A huge page is a memory page that is larger than 4Ki. On x86_64 architectures, there are two common huge page sizes: 2Mi and 1Gi. Sizes vary on other architectures. In order to use huge pages, code must be written so that applications are aware of them. Transparent Huge Pages (THP) attempt to automate the management of huge pages without application knowledge, but they have limitations. In particular, they are limited to 2Mi page sizes. THP can lead to performance degradation on nodes with high memory utilization or fragmentation due to defragmenting efforts of THP, which can lock memory pages. For this reason, some applications may be designed to (or recommend) usage of pre-allocated huge pages instead of THP.

In OpenShift Container Platform, applications in a pod can allocate and consume pre-allocated huge pages. This topic describes how.

## 14.2. PREREQUISITES

1. Nodes must pre-allocate huge pages in order for the node to report its huge page capacity. A node can only pre-allocate huge pages for a single size.

## 14.3. CONSUMING HUGE PAGES

Huge pages can be consumed via container level resource requirements using the resource name **hugepages-<size>**, where size is the most compact binary notation using integer values supported on a particular node. For example, if a node supports 2048KiB page sizes, it will expose a schedulable resource **hugepages-2Mi**. Unlike CPU or memory, huge pages do not support overcommitment.

```
kind: Pod
metadata:
  generateName: hugepages-volume-
spec:
  containers:
  - securityContext:
      privileged: true
    image: rhel7:latest
    command:
    - sleep
    - inf
    name: example
    volumeMounts:
    - mountPath: /hugepages
      name: hugepage
    resources:
      limits:
```

```
      hugepages-2Mi: 100Mi (1)
volumes:
- name: hugepage
  emptyDir:
    medium: HugePages
```

(1) Specify the amount of memory for **hugepages** as the exact amount to be allocated. Do not specify this value as the amount of memory for **hugepages** multiplied by the size of the page. For example, given a huge page size of 2MB, if you want to use 100MB of huge-page-backed RAM for your application, then you would allocate 50 huge pages. OpenShift Container Platform handles the math for you. As in the above example, you can specify **100MB** directly.

Some platforms support multiple huge page sizes. To allocate huge pages of a specific size, precede the huge pages boot command parameters with a huge page size selection parameter **hugepagesz=<size>**. The **<size>** value must be specified in bytes with an optional scale suffix [ **kKmMgG**]. The default huge page size can be defined with the **default_hugepagesz=<size>** boot parameter. See Configuring Transparent Huge Pages for more information.

Huge page requests must equal the limits. This is the default if limits are specified, but requests are not.

Huge pages are isolated at a pod scope. Container isolation is planned in a future iteration.

**EmptyDir** volumes backed by huge pages must not consume more huge page memory than the pod request.

Applications that consume huge pages via **shmget()** with **SHM_HUGETLB** must run with a supplemental group that matches ***proc/sys/vm/hugetlb_shm_group***.

# CHAPTER 15. OPTIMIZING ON GLUSTERFS STORAGE

## 15.1. CONVERGED MODE GUIDANCE FOR DATABASES

When you use converged mode for applications, follow the guidance and best practices provided in this topic so that you can make informed choices between gluster-block and GlusterFS modes based on your type of workload.

## 15.2. TESTED APPLICATIONS

In OpenShift Container Platform 3.10, extensive testing was done with these (no)SQL databases:

- Postgresql SQL v9.6
- MongoDB noSQL v3.2

The storage for these databases originated from a converged mode storage cluster.

For Postgresql SQL benchmarking pgbench was used for database benchmarking. For MongoDB noSQL benchmarking YCSB Yahoo! Cloud Serving Benchmark was used for benchmarking and workloada,workloadb,workloadf were tested

## 15.3. SUPPORT MATRIX

**Table 15.1. Table Title - GlusterFS**

| Database | Storage backend: **GlusterFS** | Turn **off** Performance Translators | Turn **on** Performance Translators |
|---|---|---|---|
| Postgresql SQL | Yes | • performance.stat-prefetch<br>• performance.read-ahead<br>• performance.write-behind<br>• performance.readdir-ahead<br>• performance.io-cache<br>• performance.quick-read<br>• performance.open-behind | • performance.strict-o-direct |

| | | | |
|---|---|---|---|
| MongoDB noSQL | Yes | • performance.stat-prefetch<br>• performance.read-ahead<br>• performance.write-behind<br>• performance.readdir-ahead<br>• performance.io-cache<br>• performance.quick-read<br>• performance.open-behind | • performance.strict-o-direct |

**Table 15.2. Table Title - gluster-block**

| Database | Storage backend: **gluster-block** |
|---|---|
| Postgresql | Yes |
| MongoDB | Yes |

The performance translators for GlusterFS, as mentioned above, are already part of the database profile delivered with the latest converged mode images.

## 15.4. TEST RESULTS

For Postgresql SQL databases, GlusterFS and gluster-block showed approximately the same performance results. For MongoDB noSQL databases, gluster-block performed better. Therefore, use gluster-block based storage for MongoDB noSQL databases.