Red Hat Reference Architecture Series

OpenShift v3 Scaling, Performance and Capacity Planning

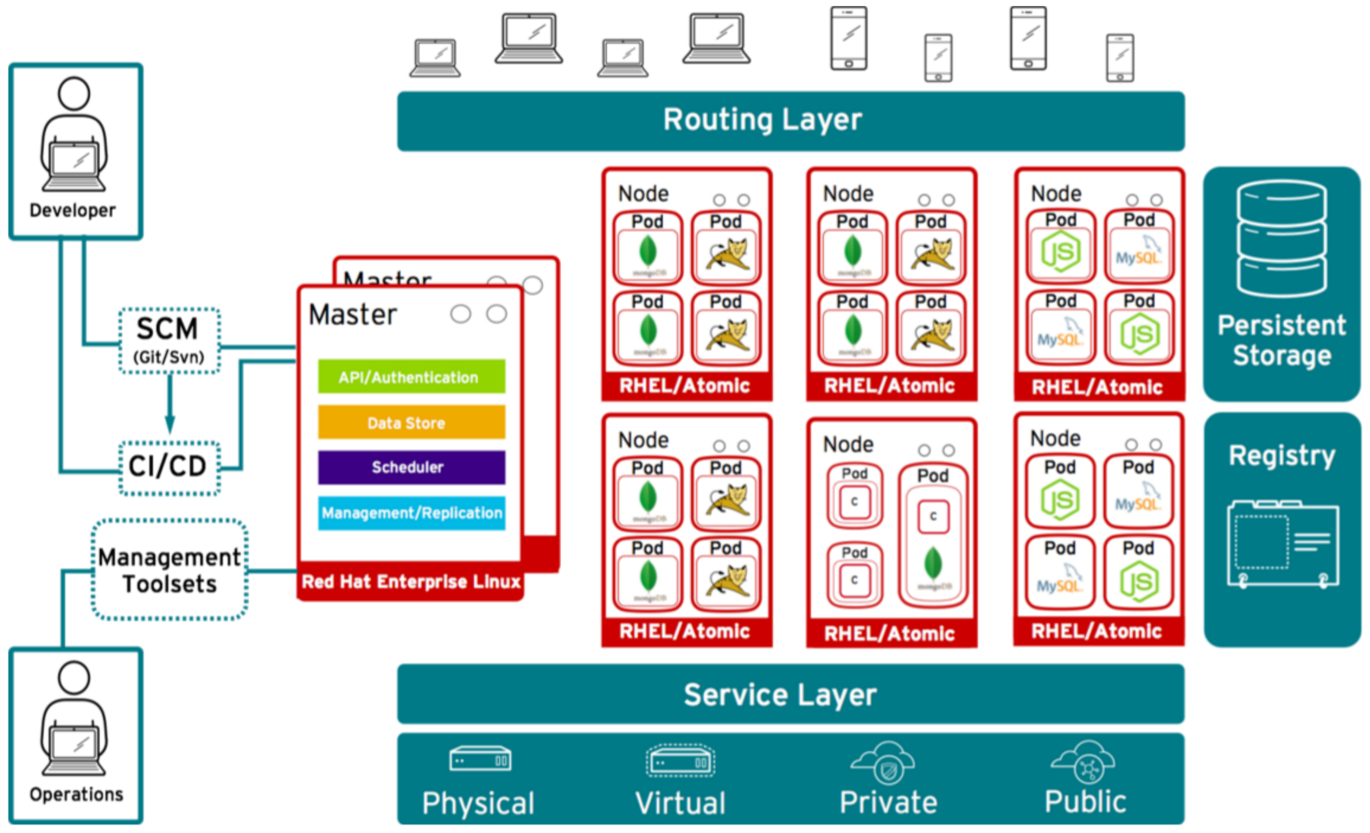
Jeremy Eder

Version 1.0, 2016-03-04

Table of Contents

1. Introduction

OpenShift is a platform-as-a-service (PaaS) product built on open source software including Red Hat Enterprise Linux (RHEL), Kubernetes, Docker, etcd and OpenvSwitch. From the HAproxy routing layer down through the kernel itself, there are opportunities to push the limits of scale and performance at each level in the product stack.



This paper is meant to assist customers who are interested in deploying scalable OpenShift-based platform-as-a-service clusters. It includes best practices, tuning options and recommendations for building reliable, performant systems at scale.

OpenShift and its component projects are rapidly evolving to provide both an ever-expanding feature set as well as steadily increased scalability.

2. Supported Product Limits

Supported product limits are evaluated for each minor release of OpenShift. As with any software, achieving maximum density requires careful capacity planning and coordination with the infrastructure. Supported product limits stated below are valid as of the general availability dates of each minor version of OpenShift. [Forward-looking statements are engineering estimates.]

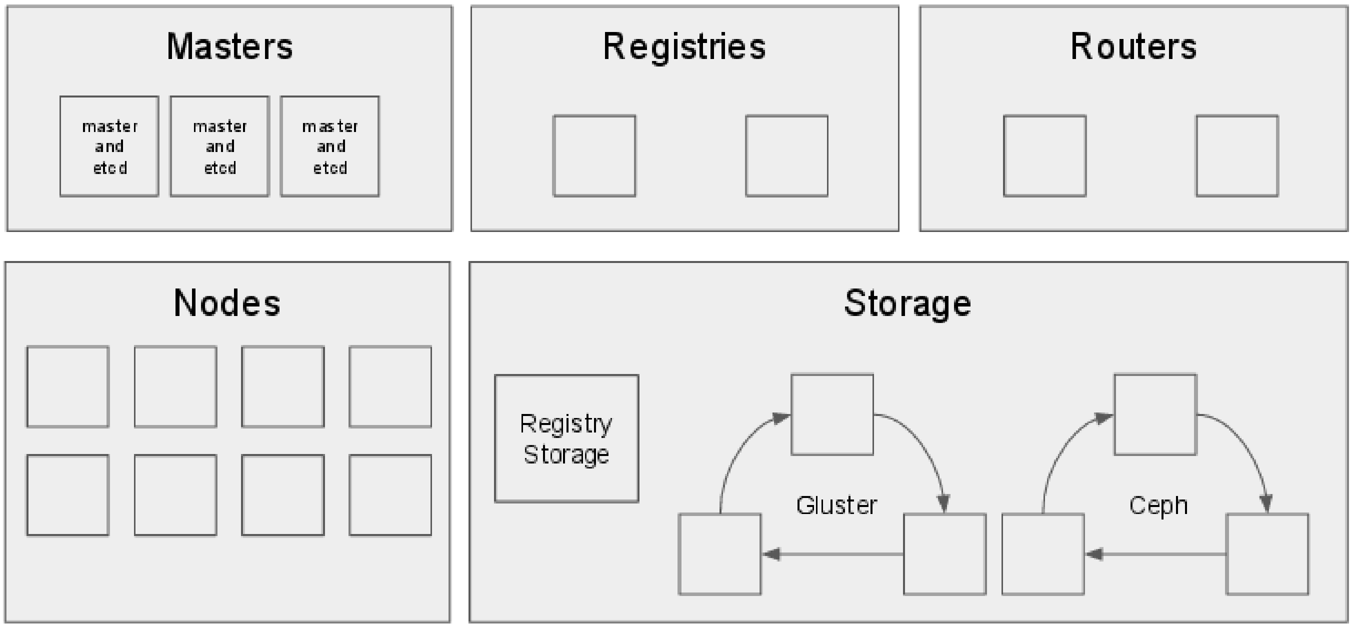
OpenShift Version	Maximum Nodes per Cluster	Total Pods per Cluster	Maximum Pods per Node
3.0 (kube 1.0)	100	3,000	40
3.1 (kube 1.1)	250	10,000	40
3.2 (kube 1.2)	300	30,000	110
3.3 (kube 1.3)	Target 1000	Target 55,000	110

3. Capacity Planning and Architecture

When planning your OpenShift environment, account for the following variables:

- Is the environment for development or production?
- What infrastructure will OpenShift run on?
- Do you require a redundant, high-availability architecture?
- What type of applications will be hosted and what are their resource requirements?
- Does your application require persistent storage?

Answers to these questions inform architectural choices. Here is a sample highly-available, scale-out OpenShift infrastructure. This infrastructure provides high availability for every service and requires seven machines for infrastructure, plus at least 1 node and (depending on infrastructure and application requirements) storage environment for persistent data.



3.1. Sizing the OpenShift Infrastructure

The OpenShift infrastructure includes master servers, container image registry servers, routers, and storage. Each of these components requires a certain amount of resources to provide the services that OpenShift nodes request from them.

Resource estimates for the infrastructure are given when running fully populated nodes:

OpenShift Version	Maximum Pods Per Node	Avg Master Resource Usage @ max-pods	Avg Node Resource Usage @ max-pods
3.0	30	0.5 vCPU, 325MB	2 vCPU, 2GB RAM
3.1	40	0.5 vCPU, 325MB	2 vCPU, 2GB RAM
3.2	100	0.25 vCPU, 175MB	0.5 vCPU, 1.5GB RAM

For environments nearing these maximums, particular attention must be paid to the Network Subnetting section within this document.

Based on answers to the capacity planning section, you might have a deployment such as:

- OpenShift will run on VMs on top of Red Hat OpenStack Platform.
- With application requirements such as:

Pod Type	Pod Quantity	Max Memory	CPU cores	Persistent Storage	Network Bandwidth
apache	100	500MB	0.5	1GB	100qps
node.js	200	1GB	1	1GB	100qps
postgresql	100	1GB	2	10GB	20Mbit
JBoss EAP	100	1GB	1	1GB	5Mbit

Extrapolated requirements: 550 CPU cores, 450GB RAM, 1.4TB storage and < 1Gbit network.

To fulfill these resource requirements, a highly-available topology might consist of:

Node Type	Quantity	CPUs	RAM (GB)
Masters + etcd	3	8	32
Registries	2	4	16
Routers	2	4	16

Instance size for nodes can be modulated up or down, depending on your preference. Nodes are often resource overcommitted. In this deployment scenario, you may 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 may come into play.

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. An example of applications that would not allow for overcommit would be most Java applications, or those that use huge pages. That memory cannot be used for other applications. In the example above, the environment would be roughly 30% overcommitted, a common ratio. See the section on Overcommit for more details in the trade offs made when overcommitting resources.

3.2. Network Subnetting

OpenShift provides IP address management for both pods and services. The default values allow for 255 nodes, each with 254 IP addresses available for pods, and provides ~64K IP addresses for services.

It is possible to resize subnets after deployment, however this is subject to certain restrictions as detailed in the OpenShift documentation.

As an example on how to increase these limits, add the following to the `[OSE3:vars]` section in your Ansible inventory, prior to installation:

```
osm_cluster_network_cidr: 172.20.0.0/14
osm_host_subnet_length: 8
```

This will allow for ~1000 nodes, each with ~250 IPs available for pods.

4. Best Practices for OpenShift Master(s)

The OpenShift master configuration file is `/etc/origin/master/master-config.yaml`. This file contains important options such as which etcd server to prefer first (if there are multiple, this is configured automatically).

The hottest data-path in the OpenShift infrastructure (excluding pod traffic) is between the OpenShift master(s) and etcd. The OpenShift API server (part of the master binary) consults etcd for node status, network configuration, secrets and much more.

For optimal performance, it is recommended to optimize this traffic path:

- Co-locate masters with etcd servers, or ensure a low latency (LAN), uncongested communication link between them.
- Ensure the first etcd server listed in `master-config.yaml` is the local etcd instance.

5. Best Practices for OpenShift Nodes

The OpenShift node configuration file is `/etc/origin/node/node-config.yaml`. This file contains important options such as the iptables synchronization period, the MTU of the SDN network and the proxy-mode; kube-proxy or iptables.

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

Not all kubelet options are supported by OpenShift.

One of the most commonly discussed node options is `max-pods`. The max-pods option limits the number of pods that can be running on a node at any given time. In kubernetes, even a pod that holds a single container actually uses two containers (see `docker ps` output). The second /pod container is used to set up networking prior to the actual container starting. Because of this, a system running 10 pods will actually have 20 containers running. See the Supported Product Limits table for limits of different versions of OpenShift.

As an example, OpenShift 3.1 defaults to 40 max-pods. There are several reasons for this, chief amongst them is the coordination that goes on between OpenShift and Docker for container status updates. This coordination puts CPU pressure on the master and docker processes, as well as writes a fair amount of log data.

Despite this default limit of 40 pods, both Red Hat and the open source community have tested well beyond that number.

In OpenShift 3.1 and earlier, deployments that exceed the default max-pods values may encounter:

- Increased CPU utilization on both OpenShift and Docker.
- Slow pod scheduling
- Potential out-of-memory scenarios (depends on amount of memory in the node)

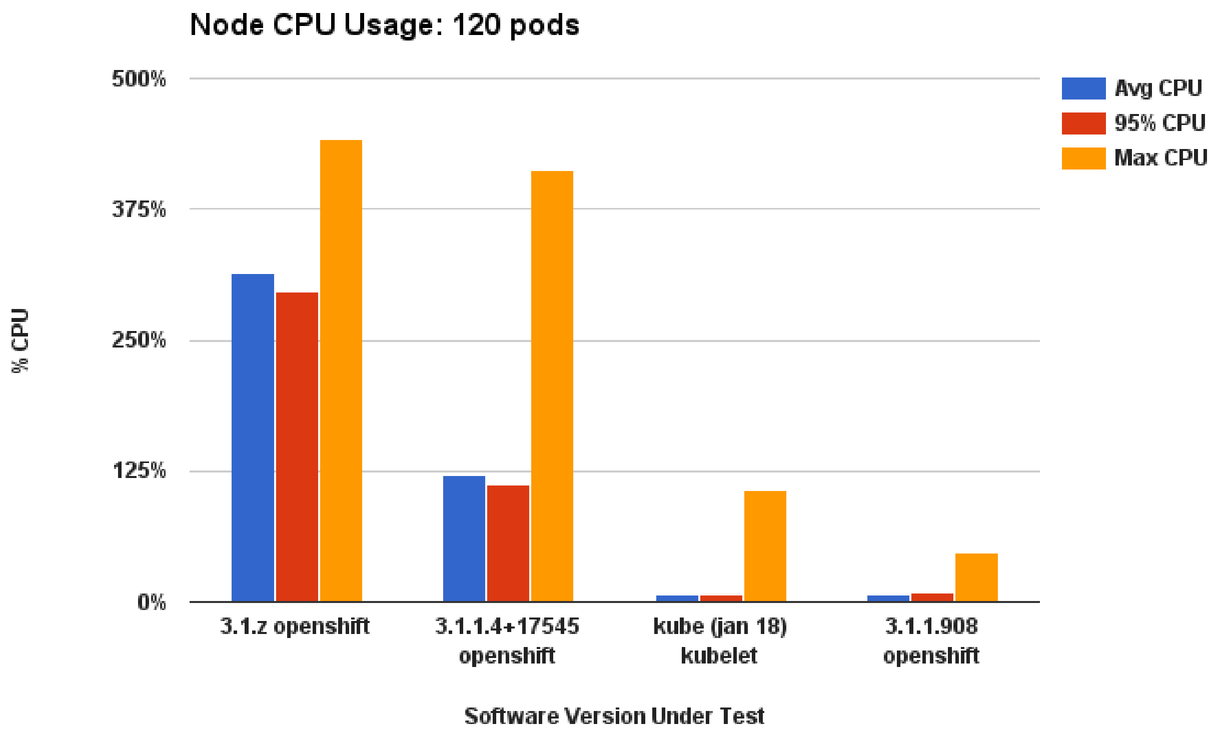
6. Improve Node Vertical Scalability/Density

Extensive work has gone on at all levels of the OpenShift stack to improve both horizontal and vertical scalability. To give you a sense of upcoming improvements in this area, take for example the CPU utilization when running 120 pods on a single node. As mentioned in the Supported Product Limits table, OpenShift 3.1 sets a max-pods value of 40 by default. As of OpenShift 3.2, this limit will be raised by nearly 3x. This is possible because of diligent work by the Kubernetes community during recent development cycles.

One of the most impactful scalability-related change in OpenShift 3.2 is the pod life-cycle event generator. This effort moved the master→docker communication from a polling to event-driven technique. Previously, the master daemon would poll Docker for container information. This behavior has changed as of OpenShift 3.2 such that the master will now listen to the Docker events stream rather than poll the Docker API.

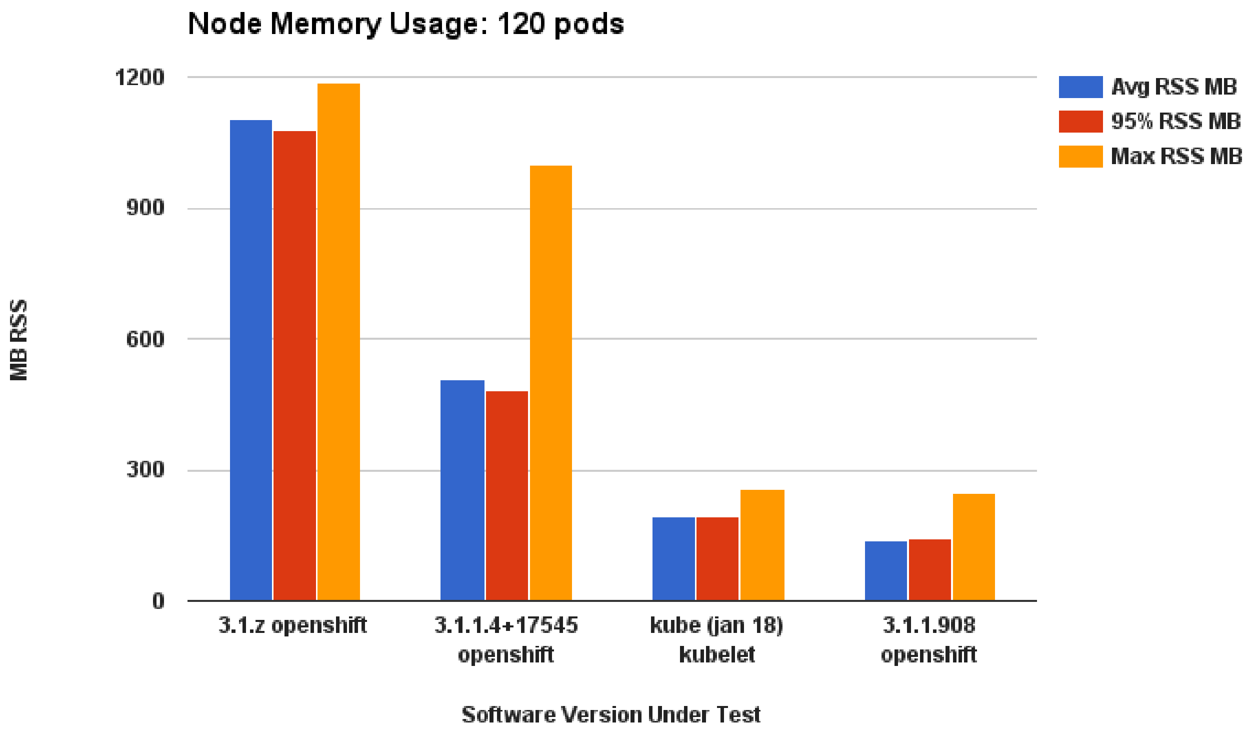
Red Hat evaluated backporting the pod life-cycle event generator, or PLEG, to OpenShift 3.1. However — rather than potentially destabilize OpenShift, we opted to concentrate effort on improving future versions of OpenShift and request users evaluate upgrading if they encounter scalability limits.

In this graph, the Y-axis is the number of CPU cores used to run 120 "sleep" pods and the X-axis are several different versions of OpenShift and upstream Kubernetes after critical patch sets (such as PLEG) were incorporated.



This represents a nearly 4x reduction in CPU usage.

At the same time, memory usage was reduced as well, also by approximately 4x:



6.1. Increasing max-pods

For proof-of-concept or testing purposes you may want to increase the max-pods limit.

To modify the max-pods argument, append the following to bottom of `/etc/origin/node/node-config.yaml` and restart the atomic-openshift-node service:

```
kubeletArguments:
  max-pods:
  - "110"
```

The openshift-ansible Advanced installer also supports setting kubeletArguments through the `openshift_node_kubelet_args` variable.

Red Hat is a significant upstream contributor to all projects upon which OpenShift is based. Our goals, along with the upstream community are to continually push the scalability limits in all dimensions: pods-per-node, total-pods-per-cluster and total-nodes-per-cluster.

7. Creating a Gold Image

Similar to creating virtual machine templates or public cloud instance templates, the purpose of creating a "gold image" is to reduce repetitive tasks as well as improve installation efficiency.

Consider an environment with 100 nodes. Each of those nodes will attempt to download a set of pods from the registry when a pod is scheduled to run on that node. This can lead to "thundering herd" behavior, resource contention on the image registry, may be considered a waste of network bandwidth and increase pod launch times.

Here are some items to consider when building a gold image:

- Create an instance of the type and size required.
  - Ensure a dedicated storage device is available for Docker local image/container storage.

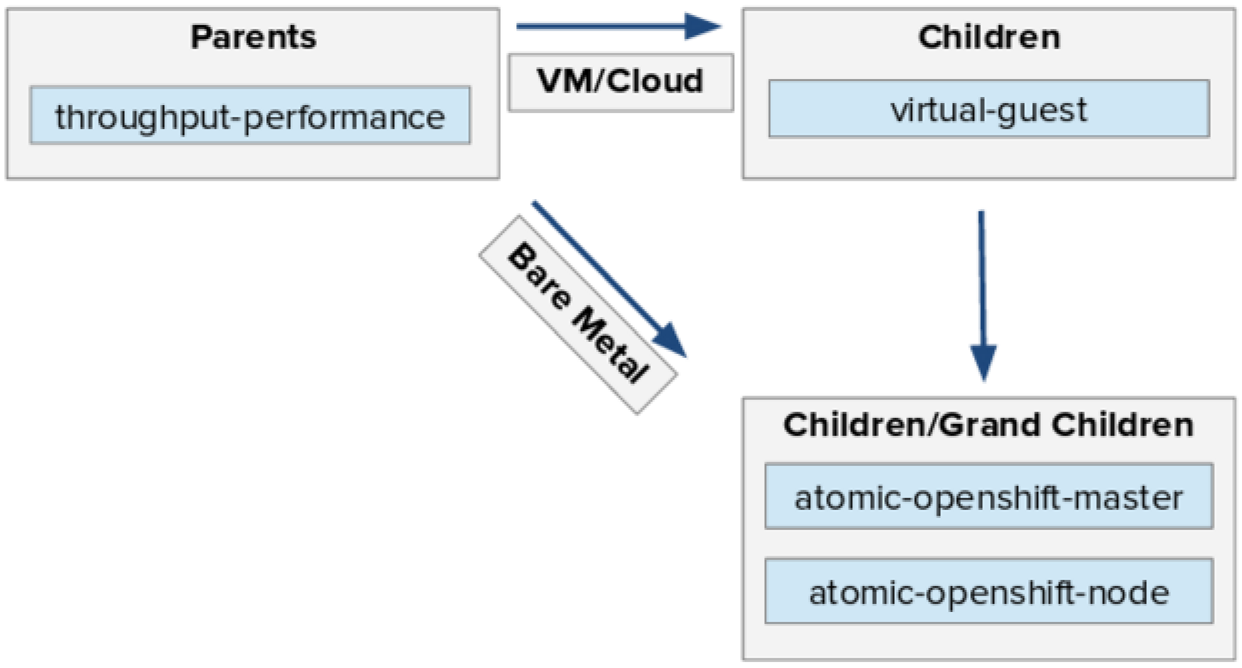
    This is separate from any persistent volumes that may be eventually be attached to containers.

  - Fully update the system.
  - Ensure machine has access to all products/yum repos.
- Install Docker (be sure not to start the docker service yet).
  - Follow the documentation to setup thin-provisioned LVM storage.
- Pre-seed your commonly used images (i.e. rhel7 base image), as well as OpenShift infrastructure container images (ose-pod, ose-deployer etc...) into your gold image.

8. Tuned

OpenShift includes `tuned` profiles called `atomic-openshift-master` and `atomic-openshift-node`. 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.

Tuned supports inheritance between profiles. The default profile for RHEL 7 is called `throughput-performance`. On an OpenShift system, the tunings delivered by Tuned will be the union of throughput-performance and atomic-openshift-node. If you are running OpenShift on a VM, tuned will notice that, and automatically apply virtual-guest tuning as well.



8.1. OpenShift Installer Tuning

OpenShift uses Ansible as its installer. Ansible is well optimized for parallel operations. There are additional tuning options that can improve installation speed and efficiency.

Ansible provides guidance and best-practices for performance and scale-out. Important notes are to use RHEL 6.6 or later (to ensure the version of openssh supports ControlPersist), and run the installer from the same LAN as the cluster. It is also important that the installer should NOT be run from a machine in the cluster.

Benefits of parallel operations are reduced total cluster deployment time. Be aware that parallel behavior may overwhelm a content source such as your image registry or Red Hat Satellite server. Pre-seeding your servers with as much content as possible (infrastructure pods, operating system patches) can help avoid this situation.

See the Configuring Ansible section of the OpenShift documentation for more information.

Here is an example Ansible configuration for large cluster installation and administration, which incorporates best practices documented by Ansible:

```
# cat /etc/ansible/ansible.cfg
# config file for ansible -- http://ansible.com/
# ==============================================
[defaults]
forks = NNN # 100 is typically safe
host_key_checking = False
remote_user = root
roles_path = roles/
gathering = smart
fact_caching = jsonfile
fact_caching_timeout = 600

[privilege_escalation]
become = False

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

8.2. OpenShift Ansible Example Inventory File

The advanced installer for OpenShift allows you to specify an Ansible “inventory” to allow for unattended installations. Below is an example that makes a few key changes:

- Enables the OpenShift-SDN Multi-tenant plug-in. This plug-in provides [network isolation] between projects through the use of OpenFlow rules and per tenant VXLAN VNID’s programmed into OpenvSwitch.
- Sets the image garbage collection threshold.
- Ensures that infrastructure machines (master, etcd and load balancers) do not run pods, by marking them as unschedulable. To allow pods to run on the infrastructure machines (not recommended for production), use the oadm manage-node command.

This example includes only those sections adjusted for performance and scale, and is not a complete inventory file.

```
[OSEv3:vars]
os_sdn_network_plugin_name='redhat/openshift-ovs-multitenant'
openshift_node_kubelet_args={'max-pods': ['110'], 'image-gc-high-threshold': ['90'],
'image-gc-low-threshold': ['80']}

[masters]
ose3-master-[1:3].example.com

[etcd]
ose3-master-[1:3].example.com

[lb]
ose3-lb.example.com

[nodes]
ose3-master-[1:3].example.com openshift_node_labels="{'region': 'infra', 'zone':
'default'}" openshift_schedulable=False
ose3-lb.example.com openshift_node_labels="{'region': 'infra', 'zone': 'default'}"
openshift_schedulable=False
ose3-node-[1:100].example.com openshift_node_labels="{'region': 'primary', 'zone':
'default'}"
```

9. Storage Optimization

Docker stores images and containers in a pluggable storage backend known as a graph driver. Some examples of graph drivers are devicemapper, overlayFS and btrfs. Each have distinct advantages and disadvantages.

For example, while OverlayFS is faster than devicemapper at starting and stopping containers, it is fairly new code, is not POSIX-compliant and does not yet support SELinux. Red Hat is working to add SELinux support for OverlayFS, but many of the POSIX compliance issues are architectural limitations of a union filesystem. For more information about OverlayFS including supportability and usage caveats, please see the Release Notes for RHEL 7.2.

For a comprehensive deep-dive on managing Docker storage, see this document:

Managing Storage with Docker Formatted Containers on RHEL and RHEL Atomic Host

In production environments, Red Hat recommends the usage of lvm thin pool on top of regular block devices (not loop devices) for storage of container images and container root filesystems. To ease the configuration of backend storage for Docker, Red Hat has written a utility called docker-storage-setup which automates much of the configuration details.

For example, if you had a separate disk drive dedicated to Docker storage (i.e. /dev/xvdb), you would add the following to /etc/sysconfig/docker-storage-setup:

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

And then restart the docker-storage-setup service:

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

After the restart, docker-storage-setup will set up a volume group docker_vg and will carve out a thin pool logical volume. Documentation for thin provisioning on RHEL is available in the LVM Administrator Guide.

Thin-provisioned volumes are not mounted and have no filesystem (individual containers do have an XFS filesystem), thus they will not show up in “df” output. View the new volumes with the lsblk command:

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

To verify that Docker is using a lvm thin pool and to monitor disk space utilization, use the “docker info” command. The Pool Name will correspond with the VG= name you specified in /etc/sysconfig/docker-storage-setup:

```
# docker info | egrep -i 'storage|pool|space|filesystem'
Storage Driver: devicemapper
 Pool Name: docker_vg-docker--pool
 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
```

By default, thin pool is configured to use 40% of the underlying block device size. As you use the storage, LVM automatically extends the thin pool up to 100%.

This is why the “Data Space Total” does not match the full size of the underlying LVM device. This auto-extend technique was used to unify the storage approach taken in both RHEL and Red Hat Atomic Host (which only uses a single partition).

For the development use-case, Docker in Red Hat distributions defaults to a loopback mounted sparse file. To see if your system is using the loopback mode, use docker info:

```
# docker info|grep loop0
 Data file: /dev/loop0
```

Red Hat strongly recommends using the Device Mapper storage driver in thin pool mode for production workloads.

OverlayFS is also supported for Docker use cases as of RHEL 7.2, and provides faster startup time, page cache sharing, which can potentially improve density by reducing overall memory utilization.

10. Debugging

Red Hat distributes a container called the rhel-tools container. The purpose of this container is two-fold:

- Allow users to deploy minimal footprint container hosts by moving packages out of the base distribution and into this support container.
- Provide debugging capabilities for Atomic Host, whose package tree is immutable.

rhel-tools includes utilities such as `tcpdump, sosreport, git, gdb, perf` and many more common system administration utilities.

Using rhel-tools is as simple as:

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

See the rhel-tools documentation for more information including examples on how to use this super-privileged container.

11. Network Performance Optimization

OpenShift SDN uses OpenvSwitch, VXLAN tunnels, OpenFlow rules and iptables to provide performant, multi-tenant software-defined network infrastructure. In many ways this network behaves like any other network, where tuning such as using jumbo frames, NIC offloads, multi-queue and ethtool settings come into play.

VXLAN provides benefits over VLANs such as an increase in networks from 4096 to over 16 million. It also provides seamless 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.

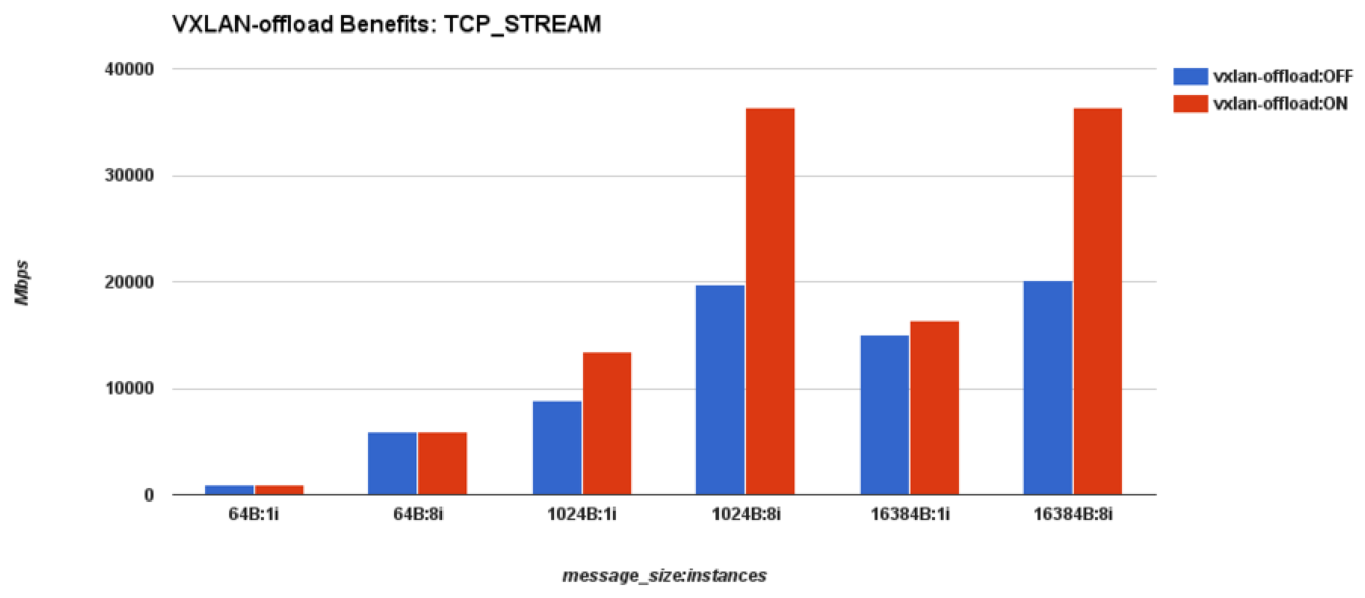
The trade-off for this simplicity is increased CPU utilization. VXLAN encapsulates all tunneled traffic in UDP packets. Both these outer- and inner-packets are subject to normal checksumming rules to guarantee data has not been corrupted during transit. Depending on CPU performance, this additional processing overhead may cause a reduction in throughput and increased latency when compared to traditional, non-Overlay networks.

We have observed typical cloud, VM and bare metal CPU performance to be capable of handling well over 1Gbps network throughput. Reduced performance may be visible when using higher bandwidth links such as 10 or 40Gbps. This is a known issue in VXLAN-based environments and is not specific to containers or OpenShift. Any network that relies on VXLAN tunnels will perform similarly because of the VXLAN implementation.

Linux kernel networking developers are working hard on improving performance in a generic fashion. Until a software-based solution is available, customers looking to push beyond 1Gbps have several options:

- 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 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. Red Hat has evaluated many of these adapters and has had excellent results.

The benefits of VXLAN-offload NICs on throughput are quite dramatic. In this example, we achieve over 37Gbps of a 40Gbps network link.



We have found that VXLAN-offload does not reduce latency, however CPU utilization is reduced even in latency tests.

12. Overcommit

Both Linux and OpenShift allow you to overcommit resources such as CPU and memory. When you overcommit, you take the risk that an application may not have access to the resources it requires when it needs them. This potentially results in reduced performance. It is however often the case that this is an acceptable tradeoff in favor of increased density and reduce costs. For example, development, QA or test environments may be overcommited, whereas production might not be.

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

13. etcd

etcd is a distributed key-value store used as a configuration store for OpenShift.

After profiling etcd under OpenShift, it was noted that etcd does a frequent amount of small storage I/O. While it is not required, etcd will perform best on storage that handles small read/write operations quickly, such as SSD. As mentioned in the Example Inventory section, we recommend optimizing communication between etcd and master servers either by co-locating them on the same machine, or providing a dedicated network.

14. Conclusion

Red Hat Performance Engineering's charter is to help our customers build the most secure, reliable, scalable, and performant systems possible by providing a solid foundation including supporting detail to allow users to confidently expand upon our work. Implementing the guidelines and best practices laid out in this paper will help you do that.

OpenShift and its component projects are an exciting and extremely fast-moving area of the open source ecosystem. Red Hat adds value to this ecosystem such as streamlined developer workflow, ISV certification, security updates, and upstream research and development resources to an already feature-rich code base.

®