Red Hat

OpenShift Virtualization

With Red Hat Ceph Storage 5 external storage

Large-scale tuning and performance

Reference architecture

Boaz Ben Shabat

Contents

Contents 1

Audience 3

Executive summary 3

Software components 4

Physical components 6

- RHCS cluster 6
- OpenShift cluster 7

Architecture 8

RHCS network tuning 9

- Address Resolution Protocol (ARP) tuning 9
  - ARP flux 9
  - ARP cache 10
- TCP/IP tuning 10
  - TCP window scaling 10
  - Buffer size tuning 11
    - Latency method 11
    - Packet size method 12
- Adapter tuning 12
  - NIC buffers 12
  - Backlog queue 13

RHCS tuning 14

- Placement groups (PGs) tuning 14
- Prometheus tuning 15

OpenShift Virtualization 16

- Introduction 16
- KubeletConfig 16
- Templates 18
  - Red Hat Linux 18
  - Fedora 19

Windows 20
Pod 22
VMs deployment 22
VMs boot storm 25
VMs latency 28
VMs migration 32
VMs migration added latency 37
Cluster upgrade at scale 39
Conclusion 40
Additional resources 40

Audience

The purpose of this document is to assist those who are responsible for infrastructure services, which includes customers, sales engineers, field consultants, and solution architects.
This document showcases an example of a successful large-scale deployment of OpenShift Virtualization, which is a feature of Red Hat OpenShift® Container Platform, with RHCS as a high-availability (HA) external network storage solution.

Executive summary

This document describes the learnings of the Red Hat OpenShift Virtualization Performance and Scale team from a successful large-scale deployment incorporating both an external Red Hat® Ceph® Storage 5 (RHCS) cluster (47 nodes) and Red Hat OpenShift Virtualization (100 nodes) with the external Ceph cluster providing storage to the OpenShift Virtualization virtual machines (VMs) accommodating a total of 3,000 VMs along with 21,400 pods.

This reference architecture will go through the steps we took to tune RHCS and Red Hat OpenShift Virtualization, allowing the creation of a resilient 100-node OpenShift cluster.

We will also explain the reasoning behind those steps and provide information that will allow the application of those recommendations to any cluster.

This table showcases the performance results for the most important scenarios that might occur in any production environment:

Scenario	Description	Result
VM deployments	**Parallel deployment of up to 800 VMs**	**Testing results show that the fastest deployment times can be achieved when cloning in 100 VMs bulks.**
VMs boot storms	**Parallel boot storms of up to 1000 VMs**	**Near linear boot times started at 01:42 (MM:SS) for 100 VMs, and ended at**

		17:45 for 1000 VMs.
VMs latency	Sustained idle latency compared to workload latency for both reads and writes	Idle VMs latency is not affected by the number of IO threads accessing RHCS, with 1 million IOPS, read latency reduced by up to 30%, while write latency increased by up to 88%.
VMs migration	1000 VMs migration	1000 VMs migration + 7000 Pod evictions took approximately 118 minutes (HH:MM).
VMs migration added latency	1000 Red Hat Enterprise Linux® (RHEL) VMs migration with workload	IO Latency during migration was increased by 9% for reads and 13% for writes, migration time was increased by 3%, and the actual IOPS rate was not impacted.
OpenShift cluster upgrade	Updating OpenShift cluster version	Minor upgrade took 35 minutes, major upgrade took 136 minutes.

Software components

Product	Version	Description
Red Hat OpenShift	4.9.15	Leading enterprise Kubernetes platform that enables a cloud-like experience everywhere it's deployed. Whether it's in the cloud, on-premise, or at the edge of the network, Red Hat OpenShift gives you the ability to choose where you build, deploy, and run applications through a consistent experience.
Red Hat Ceph Storage	5	Open, massively scalable, simplified storage solution for modern data pipelines. Engineered for data analytics, artificial intelligence/machine learning (AI/ML), and emerging workloads, Red Hat Ceph Storage delivers software-defined

		storage on your choice of industry-standard hardware.
Red Hat OpenShift Data Foundation	4.9.2	Software-defined storage for containers. Engineered as the data and storage services platform for Red Hat OpenShift, Red Hat OpenShift Data Foundation helps teams develop and deploy applications quickly and efficiently across clouds and bare-metal hosts.
Red Hat OpenShift Virtualization	4.9.2	Red Hat's solution for running VMs on a Kubernetes cluster. OpenShift Virtualization is set to achieve two goals: The first is to help all users consolidate their workloads on one platform, thus reducing the operational overhead of managing an additional virtualization platform alongside a container platform, whether they are long-term virtual machines users or new to the VM world. The second is taking advantage of the strength of the Kubernetes engine and ecosystem to help users modernize their traditional workload capabilities, orchestration, and architecture.

Red Hat

Physical components

RHCS cluster

10 * DELL PowerEdge R640 Rack Servers :

Component	Specifications	Comments
CPU	40 cores	2* Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz
Memory	384GB ECC RAM	12 * SK Hynix 1x 32GB DDR4-3200 RDIMM PC4-25600R Dual Rank x4 Module
SSD (root disk)	446.63 GB - 6 Gbps	MICRON SSD MTFDDAK480TDT
SSD (storage)	3574 GB - 12 Gbps	2 * TOSHIBA SSD KPM5XVUG1T92 1787.88 GB
NVME (storage)	2980.82 GB - 8 GT/s	Samsung NVME S5CXNA0N607551

37 * DELL PowerEdge R650 Rack Servers:

Component	Specifications	Comments
CPU	56 cores	2 * Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz
Memory	384GB ECC RAM	12 * SK Hynix 1x 32GB DDR4-3200 RDIMM PC4-25600R Dual Rank x4 Module
SSD (root disk)	446.63 GB - 6 Gbps	MICRON SSD MTFDDAK480TDT
SSD (storage)	3574 GB - 12 Gbps	2 * TOSHIBA SSD KPM5XVUG1T92 1787.88 GB
NVME (storage)	2980.82 GB - 8 GT/s	Samsung NVME S5CXNA0N607551

Note: The hardware used for RHCS was not perfect for the task since it had non-homogeneous disk sizes and architecture across the RHCS cluster, which impacted Ceph's performance. However, it is yet another strong testament to the versatility that Ceph can offer.

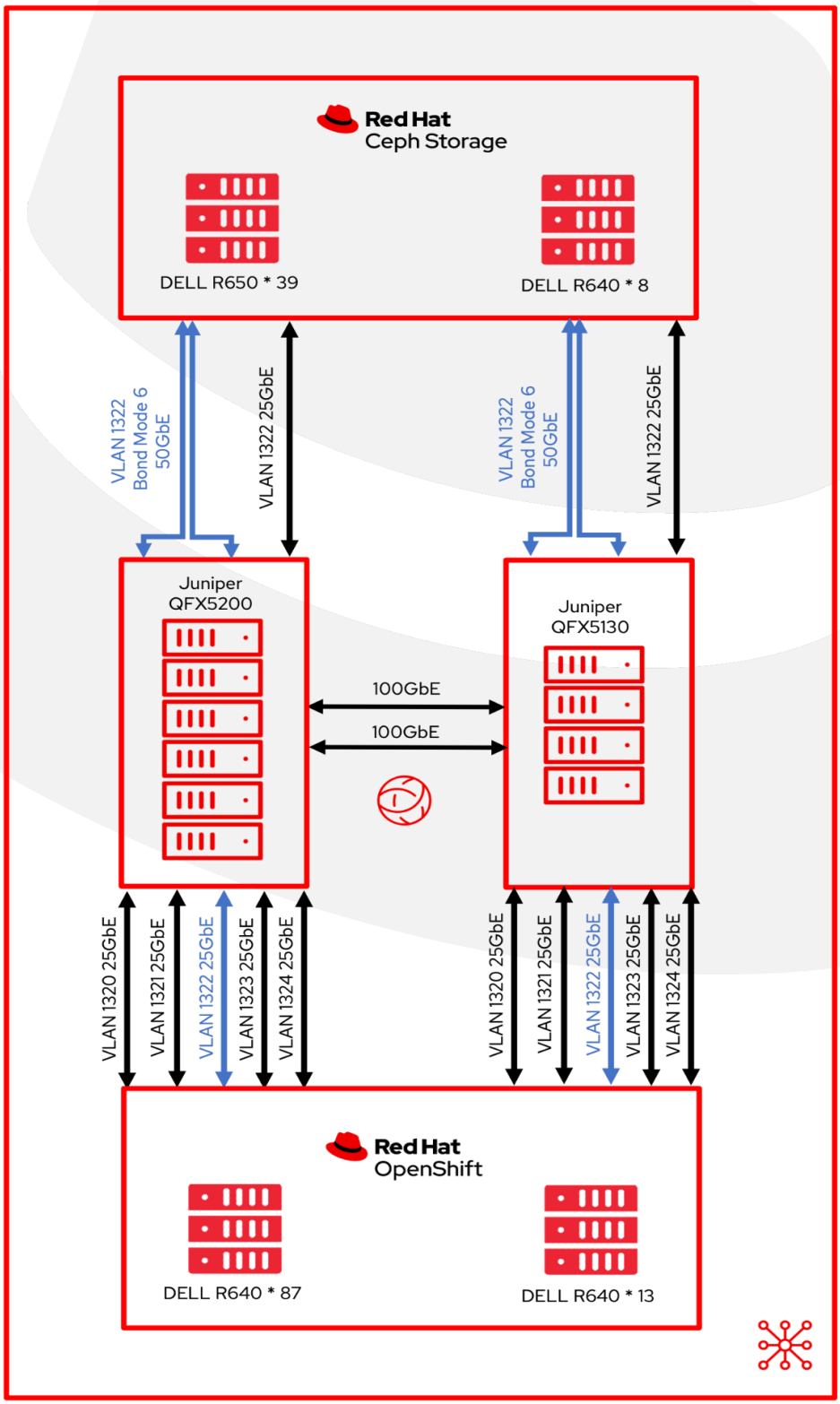
OpenShift cluster

100 * DELL PowerEdge R640 Rack Servers :

Component	Specifications	Comments
CPU	40 cores	2 * Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz 20 Cores
Memory	384GB ECC RAM	12 * SK Hynix 1x 32GB DDR4-3200 RDIMM PC4-25600R Dual Rank x4 Module
SSD (root disk)	446.63 GB - 6 Gbps	MICRON SSD MTFDDAK480TDT

Architecture

This diagram shows the networking architecture for the OpenShift and the Ceph clusters. Note that the data path between the Ceph and the Red Hat OpenShift Container Platform (OCP) cluster on a private lab VLAN uses the balance-alb bond with 2 * 25 GbE ports.



Red Hat

RHCS network tuning

This section describes the Linux network tuning performed on the Ceph nodes to cater to this large-scale environment.

Address Resolution Protocol (ARP) tuning

ARP flux

Any Linux host that has multiple network interfaces on the same subnet might be affected by ARP flux issues. The ARP flux problem might occur when a host replies to an ARP request for interfaces on the same subnet. This behavior is not necessarily a problem; however, in some cases ARP flux might cause some applications to misbehave due to incorrect mapping between IPv4 addresses and MAC addresses.

On RHEL-based hosts, we can fix this behavior by editing /etc/sysctl.d/99.8-arp.conf on all RHCS hosts, and adding the following lines:

```
net.ipv4.conf.all.arp_filter=1 #default value 0
net.ipv4.conf.all.arp_ignore=1 #default value 0
net.ipv4.conf.all.arp_announce=1 #default value 0
```

- **filter=1** - This allows you to have multiple network interfaces on the same subnet and have the ARPs for each interface be answered based on whether or not the kernel would route a packet from the ARP'd IP out that interface (therefore you must use source-based routing for this to work). In other words, it allows control of which cards (usually 1) will respond to an arp request.
- **ignore=1** - Reply only if the target IP address is a local address configured on the incoming interface.
- **arp_announce=1** - Try to avoid local addresses that are not in the target's subnet for this interface. This mode is useful when target hosts reachable via this interface require the source IP address in ARP requests to be part of their logical network configured on the receiving interface.

Make sure to load the new network settings by running:

```
$ sysctl -p /etc/sysctl.d/99.8-arp.conf
```

ARP cache

The ARP cache keeps a list of ARP entries that are generated when an IP address is resolved to a MAC address. To avoid large-scale cases in which the ARP cache cannot hold all the entries, we will need to increase the ARP cache size by editing **/etc/sysctl.d/99.7-arpcachesize.conf** and adding these lines:

```
net.ipv4.neigh.default.gc_thresh1 = 4096  #default value 128
net.ipv4.neigh.default.gc_thresh2 = 16384 #default value 512
net.ipv4.neigh.default.gc_thresh3 = 32768 #default value 1024
```

The numeric value is setting the threshold at which we will start garbage collecting for IPv4 destination cache entries. At twice this value, the system will refuse new allocations.

TCP/IP tuning

TCP window scaling

RHEL default network settings might not produce optimum throughput/latency performance for large parallel jobs that are typically found on large-scale setups. This is how to tune the Linux network and certain network devices for better parallel job performance:

For better use of high-bandwidth networks, a larger TCP window size needs to be used. Therefore, we made sure that TCP window scaling is enabled.
This can be verified with - **cat /proc/sys/net/ipv4/tcp_window_scaling**

```
$ sysctl -w net.ipv4.tcp_window_scaling=1
```

Make it persistent through reboots with:

```
$ echo "net.ipv4.tcp_window_scaling=1" >> /etc/sysctl.conf
```

Buffer size tuning

The next step will be to calculate the socket “send buffer size” and “receive buffer size.” Generally speaking, each socket's read/write buffer can hold either a minimum of 2 packets, a default of 4 packets, or a maximum of 10 packets. If the network socket buffer is too small, it might fill up and reduce the effective throughput, which will impact performance. If the network socket buffer is set large enough, it can improve performance to a certain extent.

But first some terminology:

- rmem_max - The maximum receive buffer size.
- wmem_max - The maximum send buffer size.
- wmem_default - Ts the default send buffer size.
- max_backlog - The maximum size of the receive queue.
- Netdev_budget - The maximum number of packets taken from all interfaces in one polling cycle.

We used the packet size method to calculate the optimal buffer size; however, for setups that experience high latency, we advise using the latency method.

Latency method

Optimize by calculating the maximum throughput of a single TCP connection using latency.

Optimal size =(round trip delay in microseconds) x (size of the link in Mb/s) x 1024^2

For example, our bond is running at 50000Mb/s. The latency from the Ceph node to the OpenShift cluster is 0.208 divided by 1000 to convert that into microseconds. Then multiplied by 50000 = 10.4, converted to bytes, it's 10905190.

Or:

```
ping -Ibond0 -c 60 -q 192.168.216.90|grep avg|awk -F"/" '{printf "%f", ($5
/1000) * 50000 * 1024^2}'
```

Now, we just need to set the wmem_default, which contains 4 packets, meaning ¼ of 10905190, and this is what the setting should look like:

```
net.core.rmem_max=10905190 #default value 212992
net.core.wmem_max=10905190 #default value 212992
net.core.wmem_default=4362076 #default value 212992
```

Packet size method

Optimize by packet size - a different method to use is to assume the average packet size per file is 512KB, each socket's optimal size would be: max size = (size of the packet in MB/s) x 1024^2.

```
net.core.rmem_max=5242880 #default value 212992
net.core.wmem_max=5242880 #default value 212992
net.core.wmem_default=2097152 #default value 212992
```

Note that optimizing by latency is the preferred method for networks that experience high latency regularly.

Adapter tuning

NIC buffers

On large-scale setups that contain multiple hosts, the rate of incoming traffic could potentially exceed the kernel capability to drain the buffers fast enough. If that happens, the NIC buffers will overflow, and the traffic will be lost and counted as softirq misses. We can increase the CPU time for the softirq to avoid that scenario, which is known as the netdev_budget, and we can increase the budget as necessary. On our setup, we increased the budget to 1000, which means that the softirq will drain 1000 messages on the NIC before getting off the CPU.

```
net.core.netdev_budget=1000 #default value 300
```

If the 3rd column in **/proc/net/softnet_stat** is gradually increasing over time:

```
01877e29 00000000 00000022 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000005d
0c4a6107 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000005e
01d05820 00000000 00000012 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000005f
092b933a 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000060
```

It indicates that the softirq did not get enough CPU time. In that case, the budget can be increased, preferably by small increments.

Backlog queue

Within the Linux kernel there is a queue where traffic is stored after it was received from the NIC, but before it got processed by one of the protocol stacks (TCP/IP/ISCSI).
Each CPU core has a backlog queue where traffic is stored, if the queue is already at its maximum capacity any additional packets will get dropped.

If the 2rd column in **/proc/net/softnet_stat** is gradually increasing over time:

```
04f88d2c 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
023a354d 00000000 00000018 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
10df99e1 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000002
01ba2dec 00000000 00000011 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000003
```

It indicates that the netdev backlog queue overflows and netdev_max_backlog need to be increased, again preferably by small increments.

In this scale setup, we set the value to 5000.

```
net.core.netdev_max_backlog=5000 #default value 1000
```

Red Hat

RHCS tuning

This section describes the Ceph-specific tuning performed on the Ceph nodes to cater to this large-scale environment.

Placement groups (PGs) tuning

PGs are a collection of objects that are replicated by an object storage device (OSD), each object is a container for storing data and metadata.

We can achieve the optimal number of PGs per pool by setting our target at 100PGs per OSD (according to best practice for rbd & librados), then multiply by the maximum used capacity of the pool (default is 85%), divide by the number of replicas, and round to the nearest power of 2 - 2^(round(log2(x))):

```
( Target PGs per OSD ) x ( OSDs ) x ( %Data (pool max used capacity))
----------------------------------------------------------------------
                            ( 3 replicas )
```

Or in our setup (100 * 141*0.85) / 3 =3995 rounded to a power of 2 is 4096 total PGs.

We scripted it using basic calculator (bc):

```
$ echo "x=l(100*141*0.85/3)/l(2); scale=0; 2^((x+0.5)/1)" | bc -l
```

Note that you can increase the number of PGs per OSD even further, which can potentially reduce the variance in per-OSD load across your cluster, but each PG requires a bit more CPU and memory on the OSDs that are storing it. Therefore, the number of OSDs should be tested and tuned per environment.

The next step is to apply those settings to the cluster:

```
$ ceph osd pool set pool_name pg_autoscaler_mode off
$ ceph osd pool set pool_name pg_num 4096
```

Prometheus tuning

For monitoring the Ceph cluster, we can use the Ceph dashboard to display stats for Ceph pool. We can run:

```
$ ceph config set mgr mgr/prometheus/rbd_stats_pools pool_name
```

To lessen the load on the system for large clusters, we can throttle the pool stats collection with:

```
$ ceph config set mgr mgr/prometheus/rbd_stats_pools_refresh_interval 600
#Default value 300
```

It's also a good idea to lower the polling rate for Prometheus to avoid turning the Ceph manager into a bottleneck that might result in other ceph-mgr plug-ins not getting time to run.

In this case, the command sets the scrape interval to 60 seconds:

```
$ ceph config set mgr mgr/prometheus/scrape_interval 60 #Default value 15
```

Red Hat

OpenShift Virtualization

Introduction

To demonstrate the OpenShift Virtualization capabilities and stability at a large scale, we will demonstrate these workflows:

- VM deployments.
- VM boot storm.
- VM added latency with and without workload.
- VM migration with and without workload.

The density goal for this setup was set to 3000 VMs and 21,400 pods across the cluster. This was achieved with this configuration:

- 1,500 RHEL 8.5 persistent storage VMs.
- 500 Windows10 persistent storage VMs.
- 1,000 Fedora Ephemeral storage VMs.
- 21,400 idle pods.

Or more simply put, a density of 30 VMs and 214 pods per node.

KubeletConfig

To achieve the scale mentioned in the introduction, we needed to bypass the default limit of pods per node by applying this KubeletConfig:

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: set-max-pods
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: enabled
  kubeletConfig:
    maxPods: 500
    kubeAPIBurst: 200
```

```
    kubeAPIQPS: 100
```

Other than increasing the `maxPods` to 500 (default 250), we also increased the default `kubeAPIBurst` to 200 (default 100) and `kubeAPIQPS` to 100 (default 50) to accommodate the higher burst potential. For general comparison, the default maximum number of pods for standard Kubernetes is 110 Pods per node.

Note that none of the above changes are required. Going above 250 pods per node is currently not recommended because it hasn't had any long-term testing. However, throughout our testing, we have not experienced any density-related issues.

An additional Kubeletconfig that we applied is related to BZ#1984442, which allows us to get an even distribution of VMs pods across all nodes:

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: custom-scheduling
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: enabled
  kubeletConfig:
    nodeStatusMaxImages: -1
```

Both custom Kubeletconfigs can be enabled for worker nodes by using a label:

```
oc label machineconfigpool worker custom-kubelet=enable
```

Note that KubeletConfig modifications will reboot the associated nodes.

Templates

Note that all the OS templates we used are the default templates available through the OpenShift Virtualization templates wizard, with a few changes to our custom network.

Red Hat Linux

The template being used can be retrieved by:

```
oc get templates -n openshift rhel8-server-medium -o yaml
```

Copied here, with our changes, for completeness:

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  labels:
    kubevirt.io/vm: node-os-vm
  name: node-os-vm
spec:
  running: false
  template:
    metadata:
      labels:
        kubevirt.io/vm:
    spec:
      terminationGracePeriodSeconds: 60
      evictionStrategy: LiveMigrate
      domain:
        cpu:
          cores: 1
          model: host-passthrough
          sockets: 1
          threads: 1
        devices:
          disks:
          - disk:
              bus: virtio
            name:
          interfaces:
          - bridge: {}
            model: virtio
            name: nic-0
          networkInterfaceMultiqueue: true
          rng: {}
        machine:
          type: pc-q35-rhel8.4.0
        resources:
          requests:
            cpu: "1"
            memory: 4G
      networks:
      - multus:
         networkName: linux-bridge
```

```
        name: nic-0
      volumes:
      - dataVolume:
          name:
        name:
  dataVolumeTemplates:
  - metadata:
      annotations
      name:
    spec:
      pvc:
        accessModes:
        - ReadWriteMany
        resources:
          requests:
            storage: 40Gi
        volumeMode: Block
        storageClassName: ocs-external-storagecluster-ceph-rbd
      source:
        pvc:
          namespace: "default"
          name: "rhel-dv"
```

Fedora

The template being used can be retrieved by:

```
oc get templates -n openshift fedora-desktop-medium -o yaml
```

Copied here, with our changes, for completeness:

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  labels:
    app:
    kubevirt-vm:
  name:
spec:
  annotations:
    descheduler.alpha.kubernetes.io/evict: "true"
    kubevirt.io/provisionOnNode:
  terminationGracePeriodSeconds: 0
  evictionStrategy: Restart
  running: true
  template:
    metadata:
      labels:
        kubevirt-vm: node-os-vm
    spec:
      domain:
        cpu:
          cores: 1
```

```
          sockets: 1
          threads: 1
        devices:
          disks:
            - disk:
                bus: virtio
              name: containerdisk
            - disk:
                bus: virtio
              name: cloudinitdisk
        machine:
          type: pc-q35-rhel8.4.0
        resources:
          requests:
            memory: 256Mi
            cpu: 100m
          limits:
            cpu: 100m
      terminationGracePeriodSeconds: 0
      volumes:
        - containerDisk:
            image: quay.io/kubevirt/fedora-container-disk-images:35
          name: containerdisk
        - cloudInitNoCloud:
            userData: |-
              #cloud-config
              Password: "password"
              chpasswd: { expire: False }
              runcmd:
                - sed -i -e "s/PasswordAuthentication.*/PasswordAuthentication yes/" /etc/ssh/sshd_config
                - systemctl restart sshd
          name: cloudinitdisk
status: {}
```

Windows

The template being used can be retrieved by:

```
oc get templates -n openshift windows10-desktop-medium -o yaml
```

Copied here, with our changes, for completeness:

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  labels:
    kubevirt.io/vm:
  name:
spec:
  running: false
  template:
    metadata:
      labels:
        kubevirt.io/vm:
    spec:
```

```
    terminationGracePeriodSeconds: 0
    evictionStrategy: LiveMigrate
    domain:
      clock:
        timer:
          hpet:
            present: false
          hyperv: {}
          pit:
            tickPolicy: delay
          rtc:
            tickPolicy: catchup
        utc: {}
      cpu:
        cores: 1
        model: host-passthrough
        sockets: 1
        threads: 1
      devices:
        blockMultiQueue: false
        disks:
        - disk:
            bus: virtio
          name:
        interfaces:
        - bridge: {}
          model: virtio
          name: nic-0
      features:
        acpi: {}
        apic: {}
        hyperv:
          frequencies: {}
          ipi: {}
          reenlightenment: {}
          relaxed: {}
          reset: {}
          runtime: {}
          spinlocks:
            spinlocks: 8191
          synic: {}
          synictimer:
            direct: {}
          vapic: {}
          vpindex: {}
      machine:
        type: pc-q35-rhel8.4.0
      resources:
        requests:
          cpu: "1"
          memory: 4G
        limits:
    networks:
    - multus:
        networkName: linux-bridge
      name: nic-0
    volumes:
    - dataVolume:
        name:
      name:
dataVolumeTemplates:
- metadata:
    annotations:
    name:
```

```
  spec:
    pvc:
      accessModes:
      - ReadWriteMany
      resources:
        requests:
          storage: 40Gi
      volumeMode: Block
      storageClassName: ocs-external-storagecluster-ceph-rbd
    source:
      pvc:
        namespace: "default"
        name: "win10-dv"
```

Pod

```
kind: Pod
apiVersion: v1
metadata:
  name: vdpod-pod-name
  namespace: pods-space
  labels:
    name: vdpod-density
spec:
  nodeSelector:
    node-role.kubernetes.io/worker: ""
  restartPolicy: "Always"
  containers:
  - name: vdpod-pod-name
    image: gcr.io/google_containers/pause-amd64:3.0
    ports:
    imagePullPolicy: IfNotPresent
    securityContext:
      privileged: false
```

VMs deployment

VM deployment is the base of any virtual environment. When we aim for a high scale, the faster we can deploy a large number of VMs directly affects the efficiency of our production.

In this section, we will showcase what kind of performance one can expect when cloning multiple VM images from a golden image source using the Ceph CSI cloning method.

Note that if you clone more than 100 VMs using the CSI-clone cloning strategy, then the Ceph CSI might not purge the clones, and manually deleting the clones might also fail (BZ#2055595). So it's best to avoid snapshot cloning at this time and use `cloneStrategy: copy` instead.

To enable CSI snapshot cloning, editing the OpenShift Virtualization storage profile is needed:

```
oc edit -n openshift-cnv storageprofile <storage class name>
```

And adding this spec:

```
spec:
  cloneStrategy: csi-clone
```

We start by importing a RHEL QCOW image from one of our hosts into a data volume (DV):

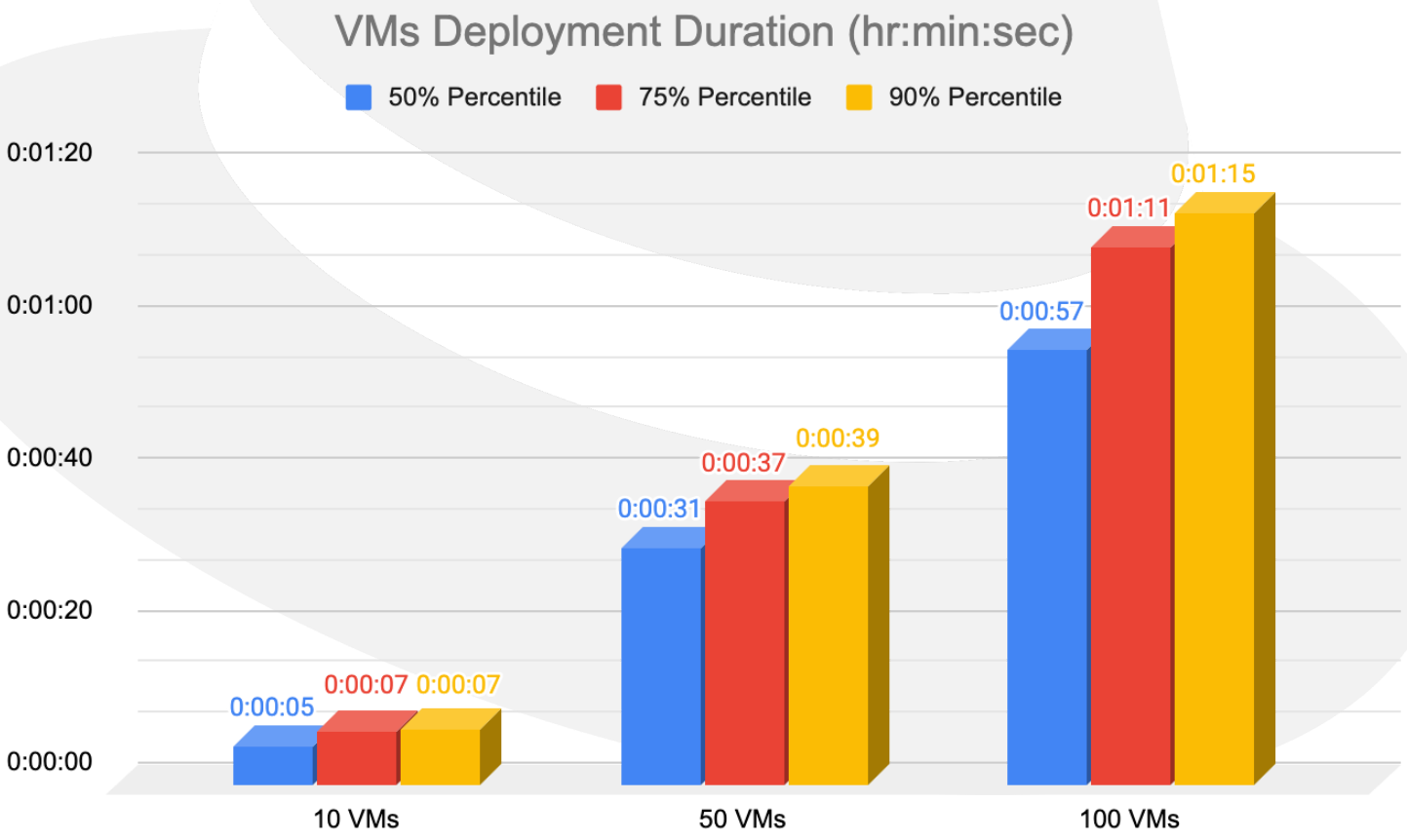
```
apiVersion: cdi.kubevirt.io/v1alpha1
kind: DataVolume
metadata:
  name: rhel-clone-dv
spec:
  source:
      http:
         url: http://internel.server.com/ISO/rhel8.qcow2
  pvc:
    accessModes:
      - ReadWriteMany
    resources:
      requests:
        storage: 40Gi
    volumeMode: Block
    storageClassName: ocs-external-storagecluster-ceph-rbd
```

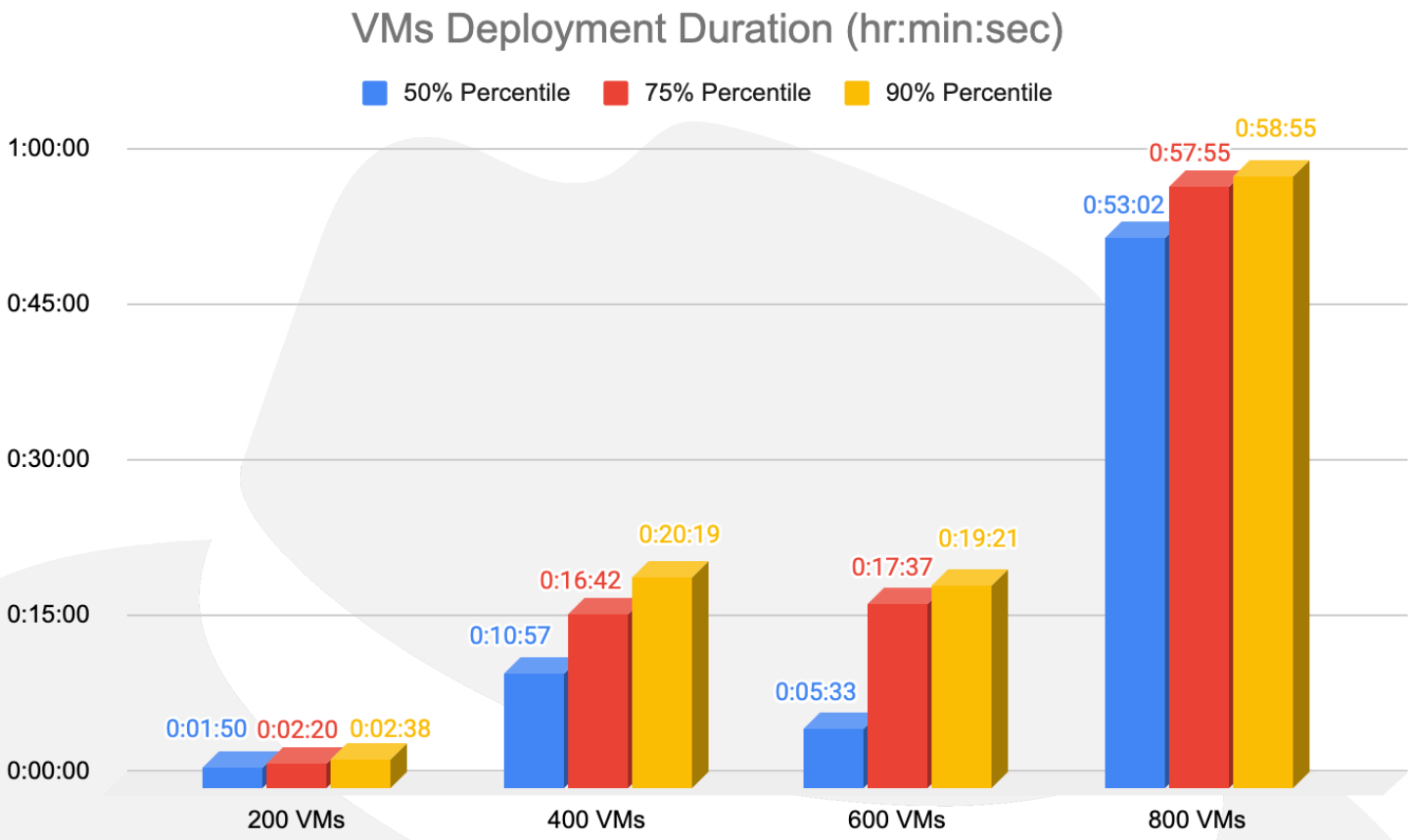
Once the import is done, we deploy the desired number of VMs in parallel and measure how long it takes for every VM to complete the cloning. We do that by querying every VM with a 2-second interval window until the clone is completed.

The most effective way to deploy VMs will be in groups of 100, meaning deploy 100 VMs, wait for the cloning to complete, and deploy the next 100 VMs.

Generally speaking, once we go above the parallel deployment of 10 VMs, a penalty will occur that happens because at the Ceph CSI level, it acquires a lock on parent image vol-id, so cloning from the same parent image is not parallel but actually serial, and the external provisioner can only send 10 gRPC parallel calls to CSI driver at any time.

In addition, once we pass the 250 clones mark, rbd images will start flattening, which will increase the cloning penalty much further. The time it takes to flatten a clone increases with the size of the snapshot.





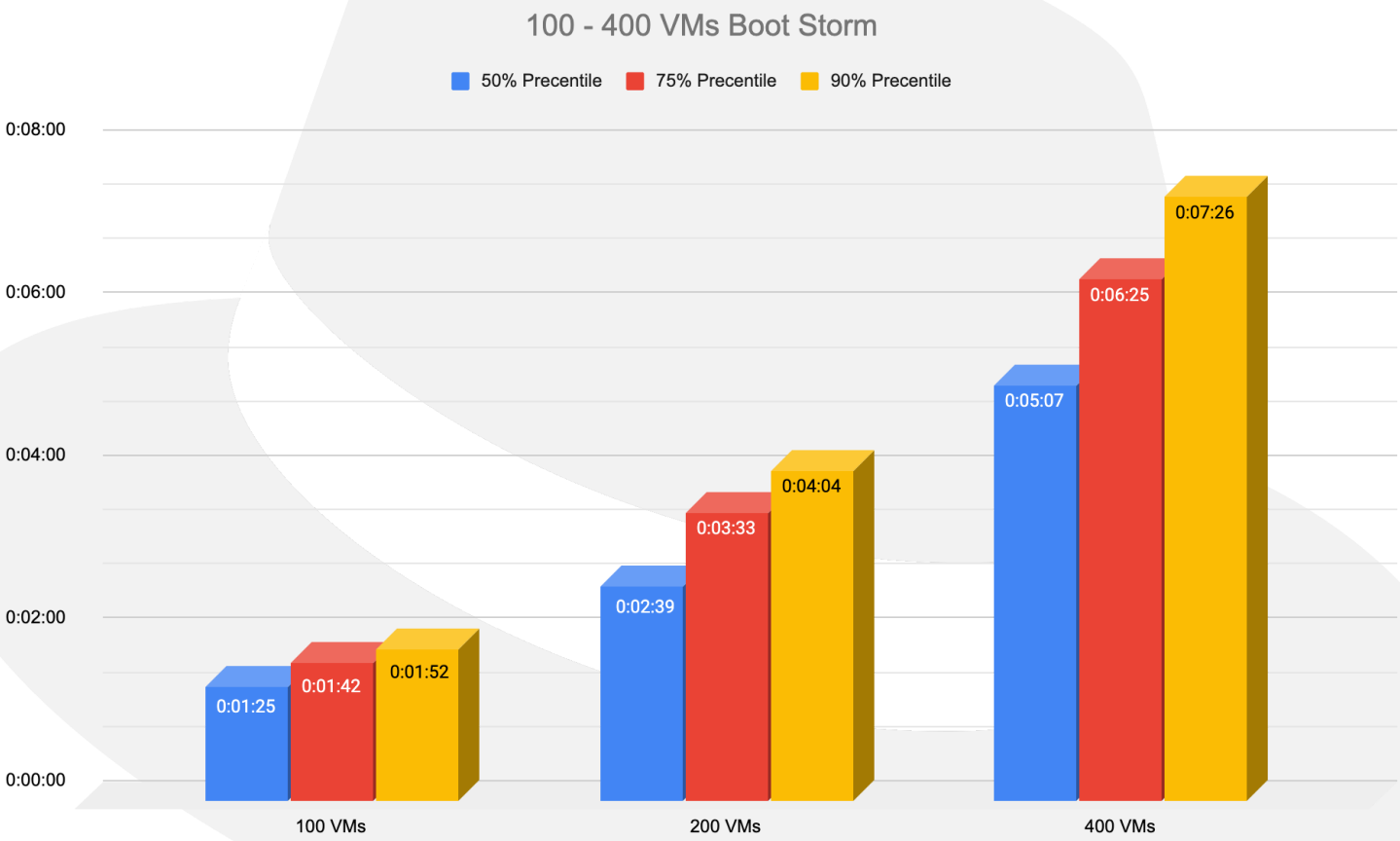
VMs boot storm

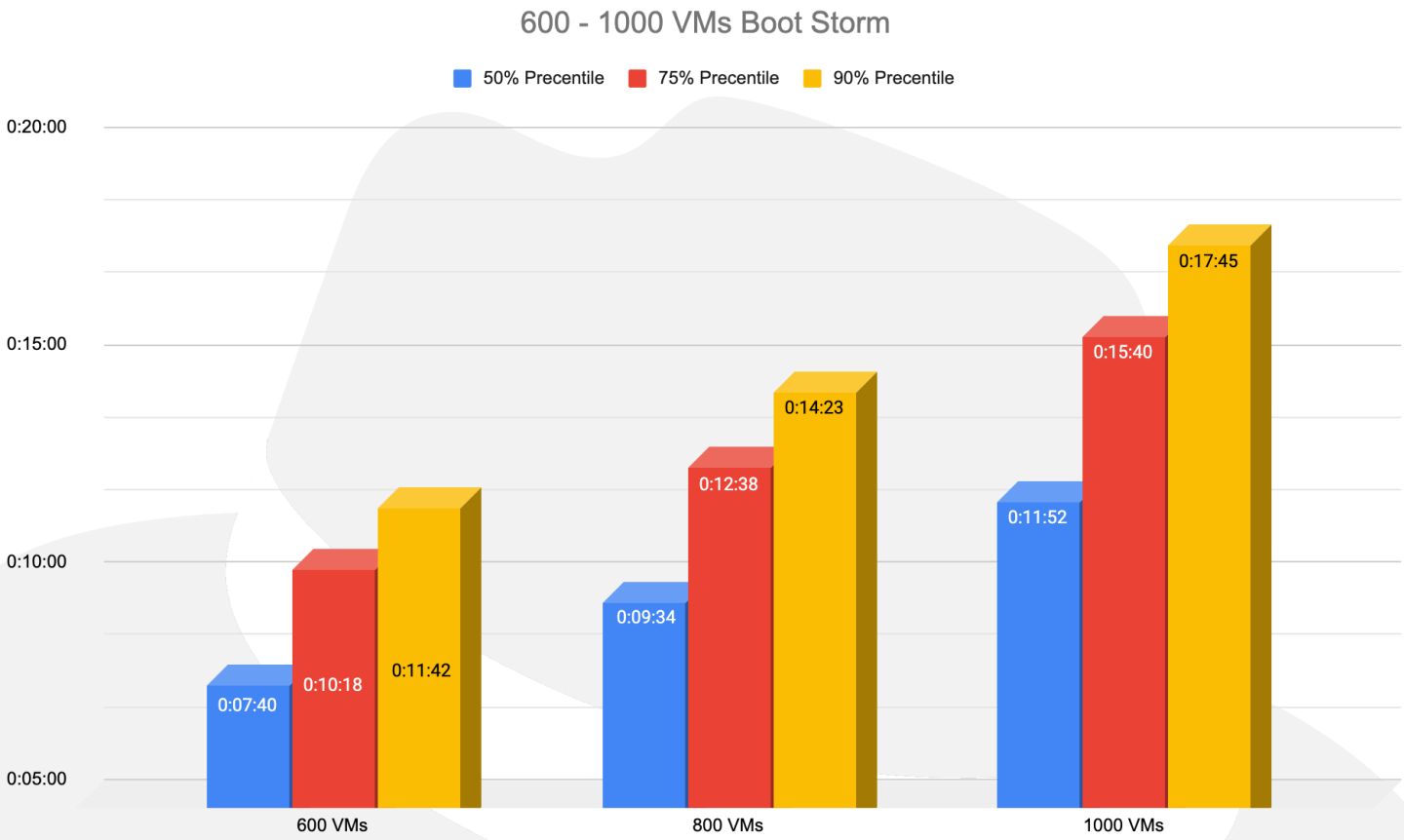
In this scenario, we tested how long it took to boot a large number of VMs, which showcases how both OpenShift Virtualization and the control plane are resilient. It's a scenario that often occurs when recovering an environment from a disaster—such as a power outage, for example.

The measurement is for every VM starting at request time and stopping once the VM is running and accessible through SSH access (meaning the host booted successfully and the SSH daemon is up). We measured the times by running a query against each VM. When the status changes to "Running," it then attempts to SSH to the VM every 2 seconds, until the SSH connection is successful.

Like the deployment scenario, all VMs start requests run in parallel to stress all the moving parts of the cluster.

As we can see in the charts below, with our homogeneous OCP cluster, the boot times are almost linear up to 1000 VMs, but the larger queues might result in slower boot times.




600 - 1000 VMs Boot Storm
50% Precentile
75% Precentile
90% Precentile
0:20:00
0:15:00
0:10:00
0:05:00
0:07:40
0:10:18
0:11:42
0:09:34
0:12:38
0:14:23
0:11:52
0:15:40
0:17:45
600 VMs
800 VMs
1000 VMs

VMs latency

Each VM has its own IO thread for processing IO unless there are multiple persistent volume claims (PVC) and `dedicatedIOThread:` is set to true. In that case, each PVC will have its own IO thread.

In the following scenario, we used 15 VMs per worker node and tested up to 64 worker nodes, or 960 VMs, to demonstrate that although there are multiple concurrent threads that are accessing the RHCS cluster, those by themselves will not cause any latency penalties.

For the purpose of this scenario, we choose to use the 4KB block size for both random reads and random writes, and we ran these tests:

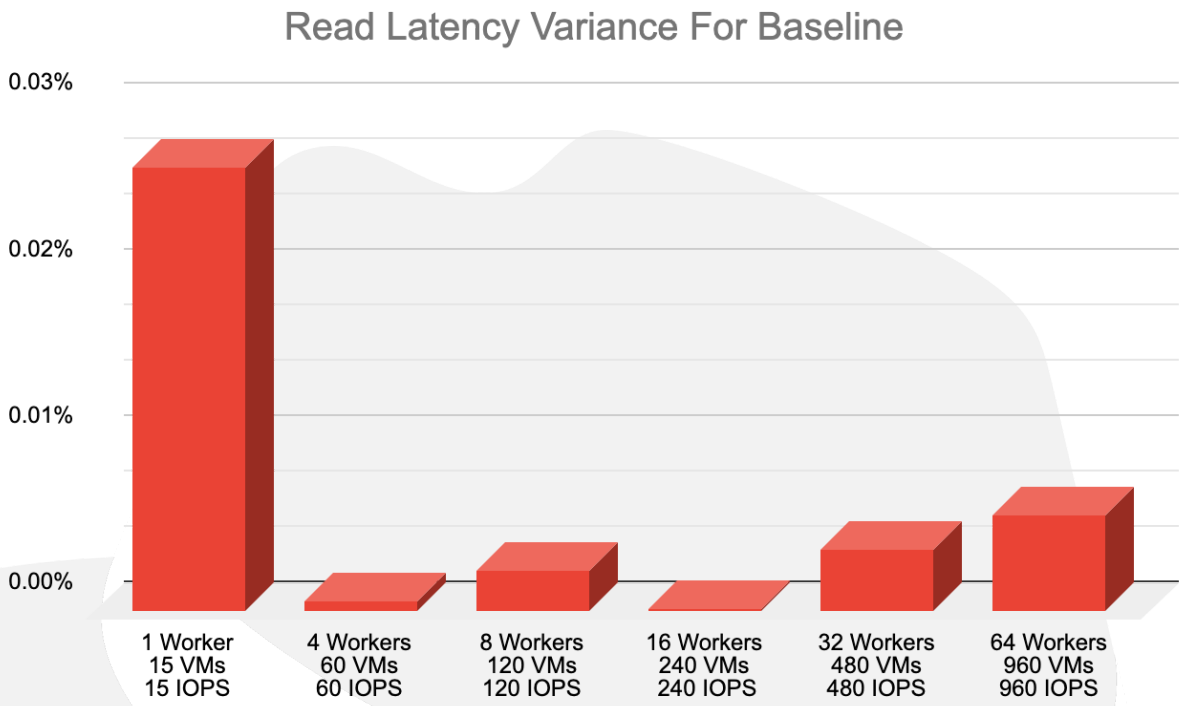
- Baseline - We used 15 VMs from every worker node with each VM generating a single IOPS starting at 15 VMs (15 IOPS, Single node) and ending at 64 nodes for a total of 960 VMs (960 IOPS).
- Workload - We used exactly 15 VMs from every worker node with each VM generating 1000 IOPS starting at 15 VMs (15,000 IOPS, Single node) and ending at 64 nodes for a total of 960 VMs (960,000 IOPS).

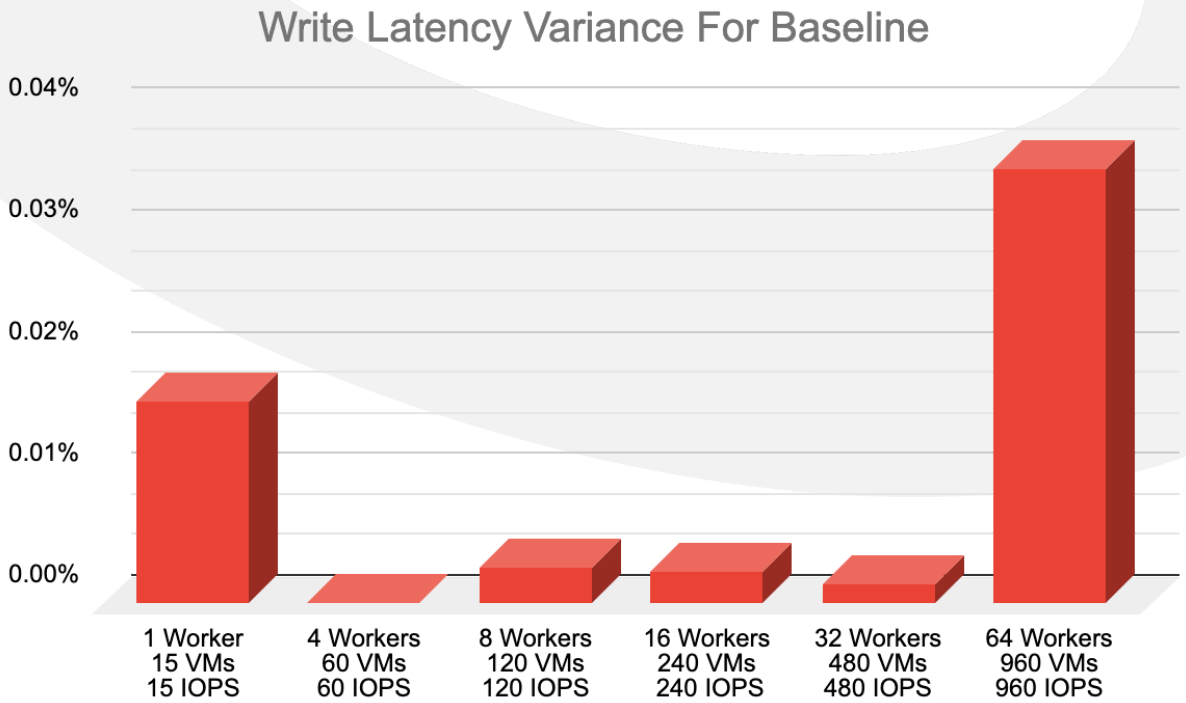
Each VM filesystem dataset consists of 300 directories, each directory contains 8 files, and each file is 20MiB in size, or more simply put each VM has a 4.8 GiB dataset.

We chose a small block to avoid, as much as possible, any inconsistencies that might occur due to networking and Ceph non-homogeneous disks on the RHCS cluster.

Note that although we used up to 960 VMs in our testing, in total there were actually 3000 VMs running on the cluster along with 21,400 pods.

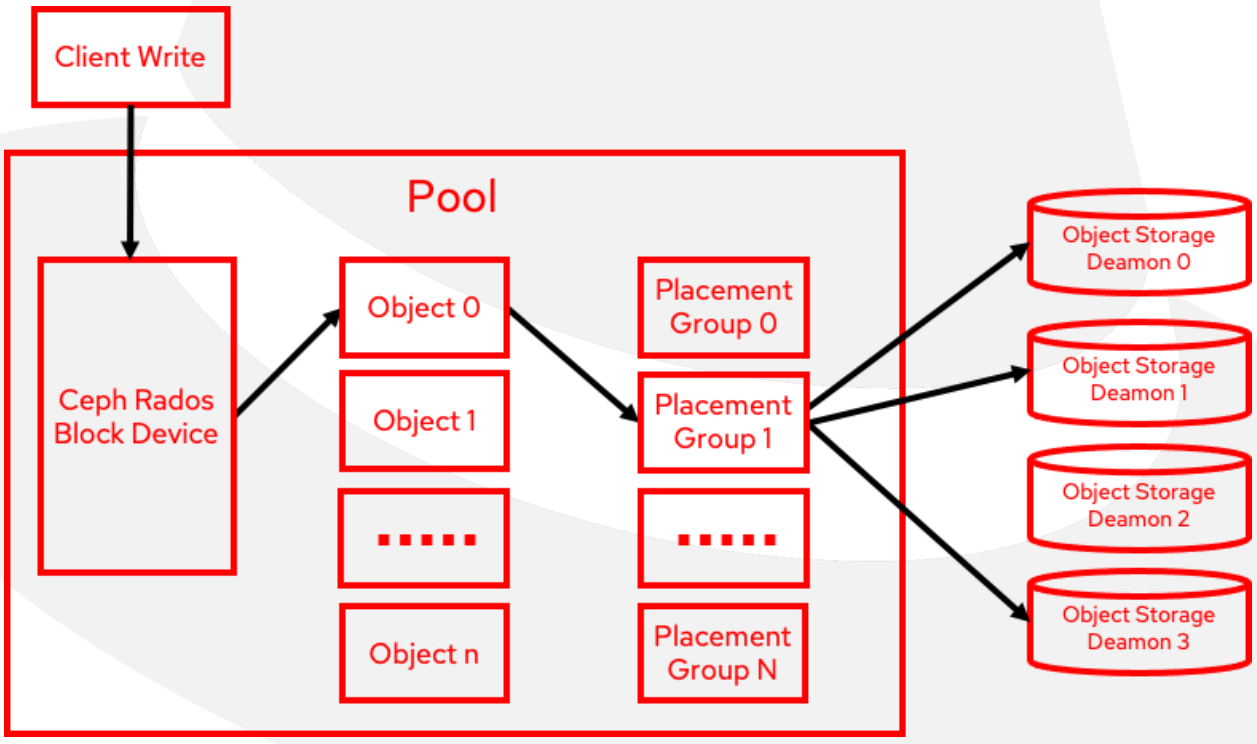
As we can see in both charts below, while running the baseline tests, the variance in latency for both random reads and random writes was less than 0.04%.


Read Latency Variance For Baseline
0.03%
0.02%
0.01%
0.00%
1 Worker
15 VMs
15 IOPS
4 Workers
60 VMs
60 IOPS
8 Workers
120 VMs
120 IOPS
16 Workers
240 VMs
240 IOPS
32 Workers
480 VMs
480 IOPS
64 Workers
960 VMs
960 IOPS

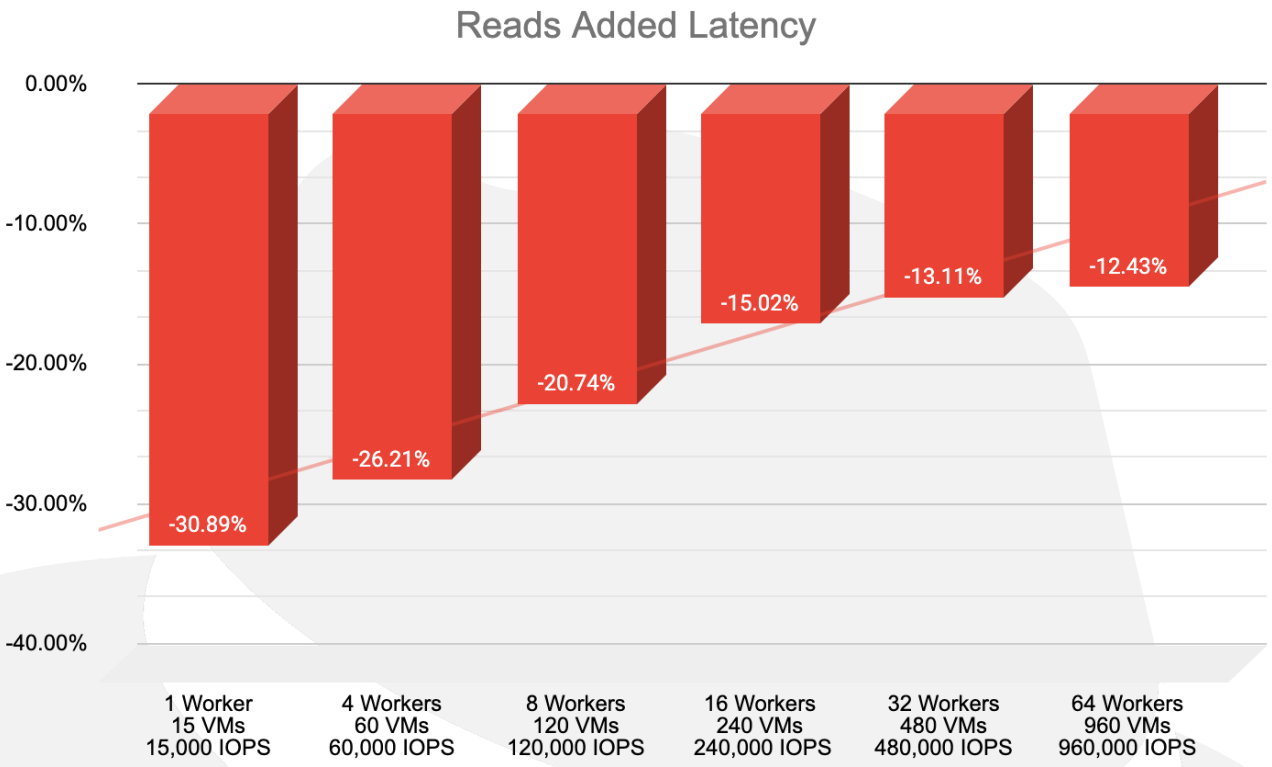

Write Latency Variance For Baseline
0.04%
0.03%
0.02%
0.01%
0.00%
1 Worker
15 VMs
15 IOPS
4 Workers
60 VMs
60 IOPS
8 Workers
120 VMs
120 IOPS
16 Workers
240 VMs
240 IOPS
32 Workers
480 VMs
480 IOPS
64 Workers
960 VMs
960 IOPS

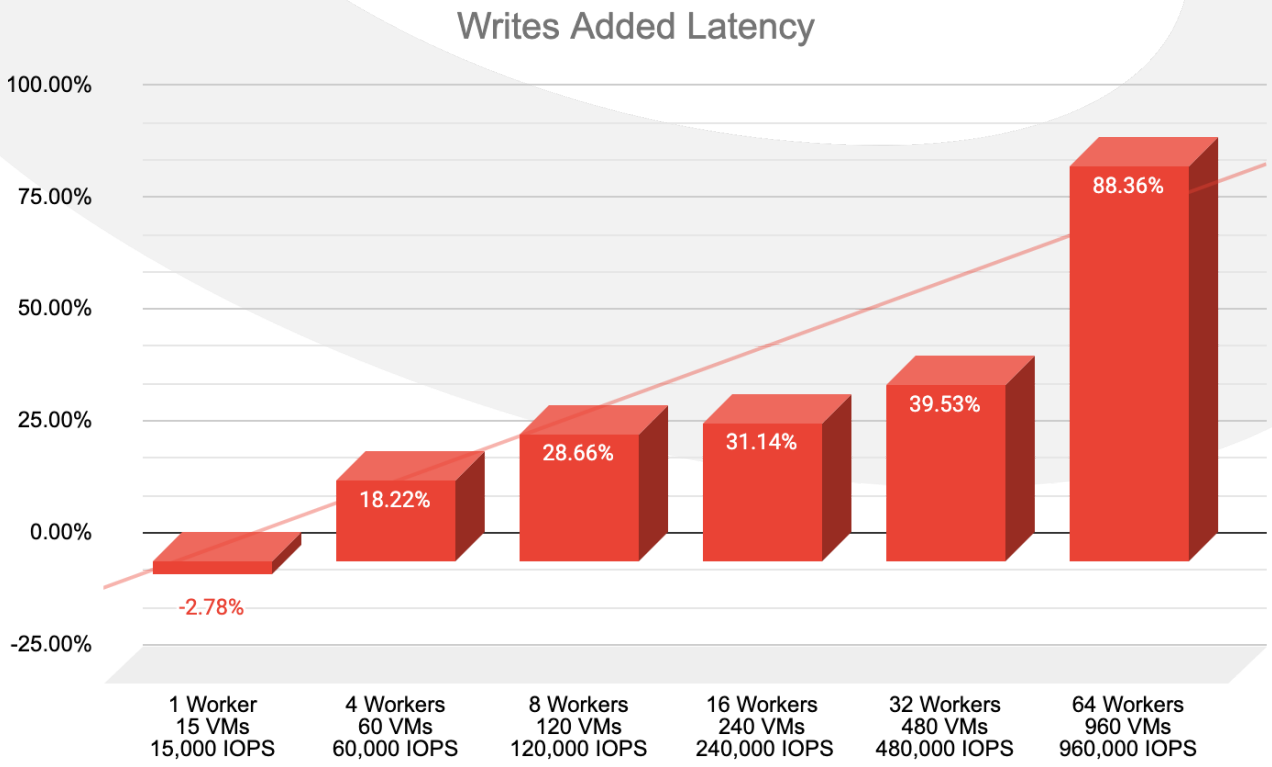
The chart below shows the latency trend of the workload scenario in comparison to the baseline results (lower means better). When it comes to read performance, a higher IOPS rate will yield lower latency to some extent, which happens due to the resources allocation that occurs during higher bursts when compared to idle/low IOPS workload on the VM.

However, writes have a different dataflow, which is required to maintain high availability. As shown in the diagram, for every write Ceph generates, 3 copies of the data go through the network to each of their respective OSDs. Therefore, the write latency will be impacted.



Note that in cases of latency-sensitive applications, the write latency overhead can be reduced by one-third for every data replica reduced.


Reads Added Latency
0.00%
-10.00%
-20.00%
-30.00%
-40.00%
-30.89%
-26.21%
-20.74%
-15.02%
-13.11%
-12.43%
1 Worker
15 VMs
15,000 IOPS
4 Workers
60 VMs
60,000 IOPS
8 Workers
120 VMs
120,000 IOPS
16 Workers
240 VMs
240,000 IOPS
32 Workers
480 VMs
480,000 IOPS
64 Workers
960 VMs
960,000 IOPS


Writes Added Latency
100.00%
75.00%
50.00%
25.00%
0.00%
-25.00%
-2.78%
18.22%
28.66%
31.14%
39.53%
88.36%
1 Worker
15 VMs
15,000 IOPS
4 Workers
60 VMs
60,000 IOPS
8 Workers
120 VMs
120,000 IOPS
16 Workers
240 VMs
240,000 IOPS
32 Workers
480 VMs
480,000 IOPS
64 Workers
960 VMs
960,000 IOPS

VMs migration

In the following scenario, we tested a 1000 VMs migration. To simulate a realistic migration, we didn't just migrate VMs, but we also rebooted the worker nodes that those VMs reside on. We achieved that by dividing the nodes into 3 zones and then applying an empty machine config to a specific zone, which results in a reboot of all nodes that were associated with that specific zone once all the VMs that resided on the worker nodes are evicted.

We started by labeling every worker to a specific zone:

```
oc label node worker01 node-role.kubernetes.io/zone-0=""
oc label node worker02 node-role.kubernetes.io/zone-1=""
oc label node worker03 node-role.kubernetes.io/zone-2=""
```

Then we created a machine config pool for every zone. Note that `maxUnavailable: 10` sets the number of nodes that are allowed to go down at any given time of the zone life:

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  name: zone-0
spec:
  machineConfigSelector:
    matchExpressions:
      - {key: machineconfiguration.openshift.io/role, operator: In, values:
[worker, zone-2]}
  nodeSelector:
    matchLabels:
      node-role.kubernetes.io/zone-0: ""
  paused: false
  maxUnavailable: 10
```

We also edited the hyperconverged-cluster operator:

```
oc edit hco -n openshift-cnv kubevirt-hyperconverged
```

And set the following migration settings to increase the amount of parallel migrations :

```
  liveMigrationConfig:
    completionTimeoutPerGiB: 800
    parallelMigrationsPerCluster: 20 # default 5
    parallelOutboundMigrationsPerNode: 4 # default 2
    progressTimeout: 150
```

Through our testing, we found that it is highly recommended to increase the number of virt-api pods to a ratio of **1 kubevirt-api pod for every 750 VMs**. Since we were running 3000 VMs in this setup, we scaled the number of kubevirt API Pods to 4.

Auto-scaling functionality for this scenario is already in the works, and can be tracked at Github#7101. Currently, it can be done manually by patching the hyperconverged operator:

```
oc patch hco -n openshift-cnv kubevirt-hyperconverged --type=merge -p
'{"metadata":{"annotations":{"kubevirt.kubevirt.io/jsonpatch":"[{\"op\":
\"add\", \"path\": \"/spec/customizeComponents/patches\", \"value\":
[{\"resourceType\": \"Deployment\", \"resourceName\": \"virt-api\",
\"type\": \"json\", \"patch\": \"[{\\\"op\\\": \\\"replace\\\",
\\\"path\\\": \\\"/spec/replicas\\\", \\\"value\\\": 4}]\"}]}]"}}}'
```

We then trigger the migration by creating this machine config:

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: zone_target #target zone name
  name: job_name #must be unique every time.
spec:
  config:
    ignition:
      config: {}
      security:
        tls: {}
      timeouts: {}
```

```
    version: 3.1.0
  networkd: {}
  passwd: {}
  storage:
    files:
    - contents:
        source: data:text/plain;charset=utf-8;base64,Zm9vCg==
        verification: {}
      filesystem: root
      mode: 420
      path: /var/tmp/tmp_dir
osImageURL: ""
```

The 1000 migrated VMs consisted of:

- 400 RHEL VMs.
- 400 Fedora VMs.
- 200 Windows VMs.

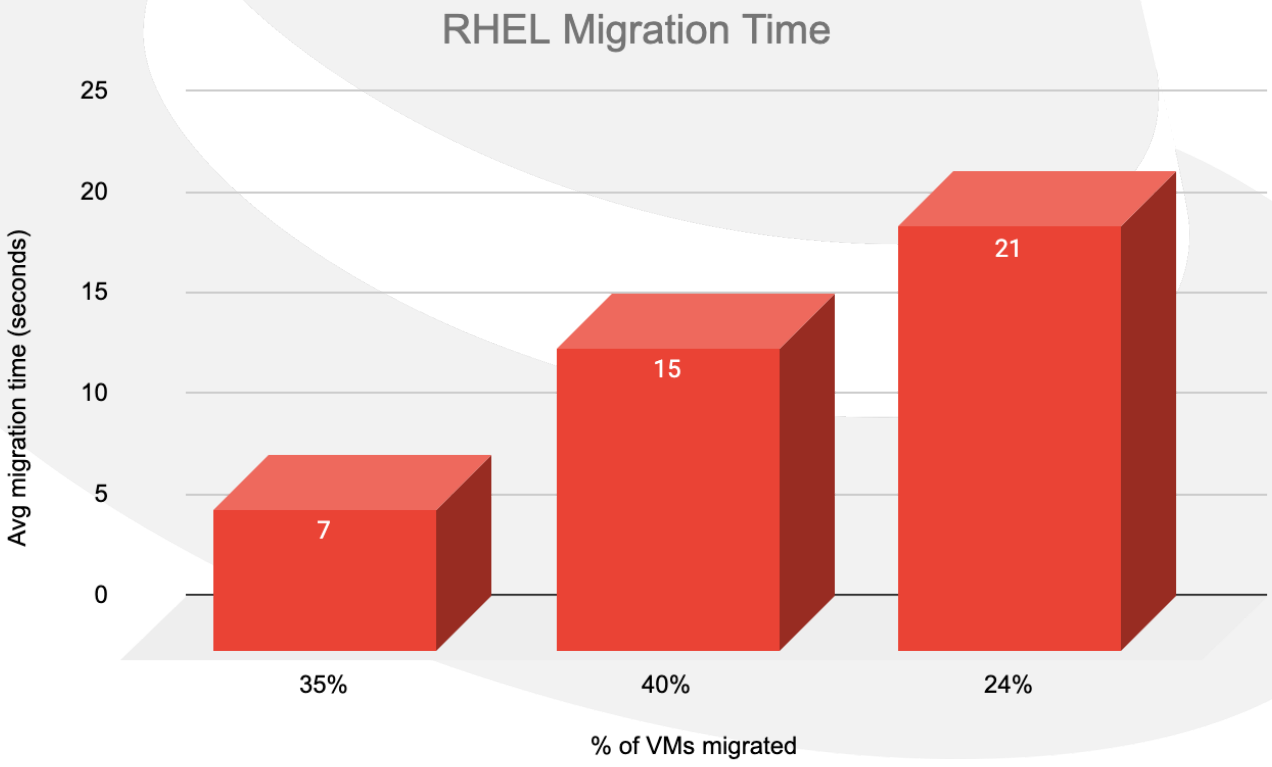
The 7000 pods co-located with the VMs were also restarted on different worker nodes.

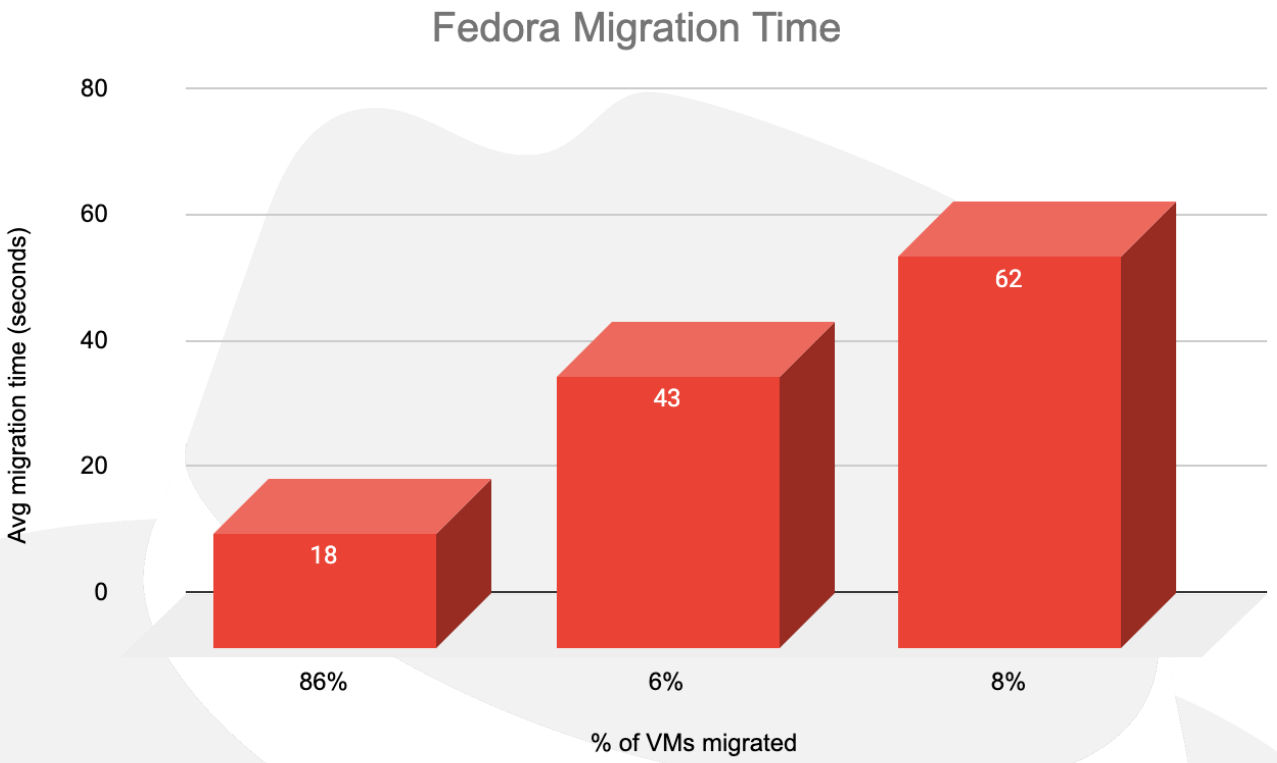
When migrating, we can see in the VMI logs, for any VM, how long it took to complete the migration:

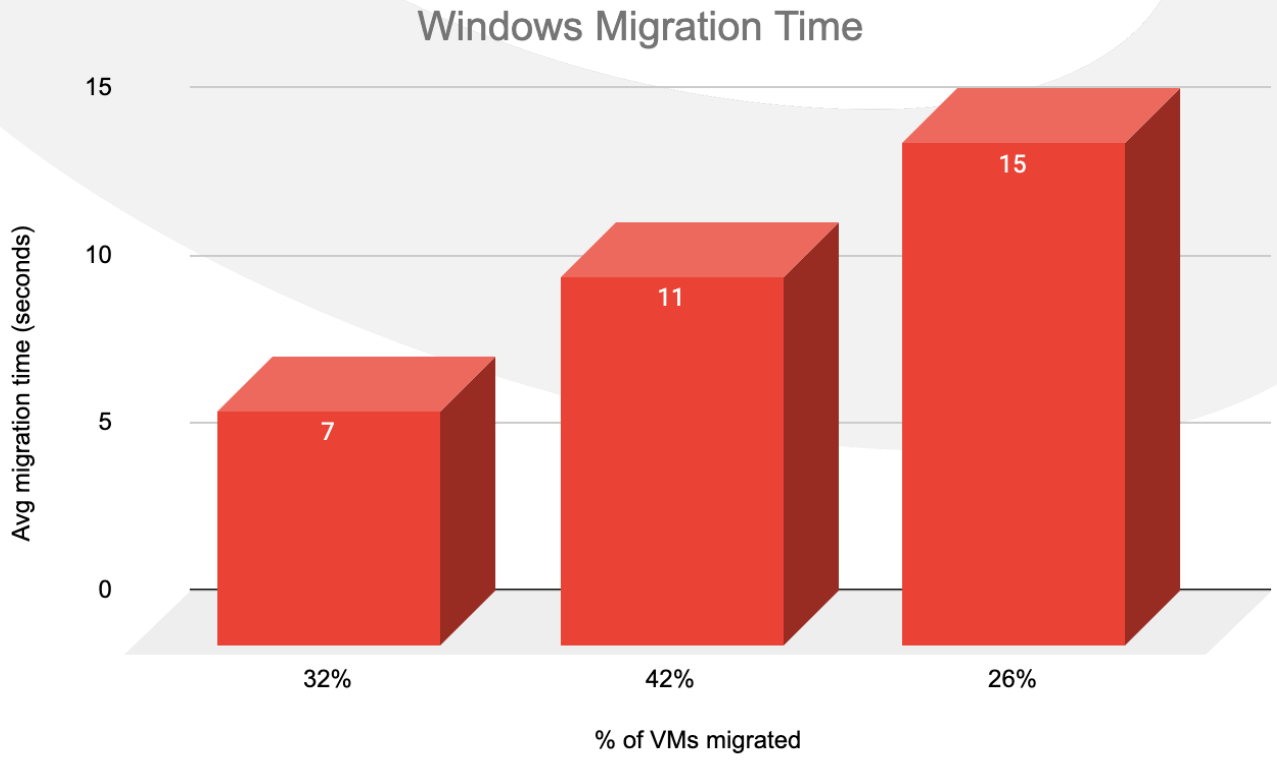
```
Phase Transition Timestamps:
  Phase:                        Scheduling
  Phase Transition Timestamp:   2022-04-10T07:15:13Z
  Phase:                        Scheduled
  Phase Transition Timestamp:   2022-04-10T07:15:23Z
  Phase:                        Running
  Phase Transition Timestamp:   2022-04-10T07:15:25Z
```

The charts below show the migration times of each OS distributed by percentage. For example, on the RHEL VMs migration, 35% of the VMs completed the migration with an average time of 7 seconds.

Note that migration time is not necessarily OS-related but rather related to guest load at the time, host load, network load, storage technology, migration policy, image size, etc. Therefore, those results may vary.




Fedora Migration Time
Avg migration time (seconds)
80
60
40
20
0
18
43
62
86%
6%
8%
% of VMs migrated


Windows Migration Time
Avg migration time (seconds)
15
10
5
0
7
11
15
32%
42%
26%
% of VMs migrated

Average migration time per OS:

OS	Avg migration time (sec)	Comments
RHEL	14	40GiB PVC
Fedora	23	Container disk evictionStrategy: Restart
Windows	12	40GiB PVC

To sum things up, from start to finish, migration took a total of 118 minutes + 35 minutes that we spent waiting for the nodes to reach "Ready" state due to the `maxUnavailable: 10` that we mentioned earlier.

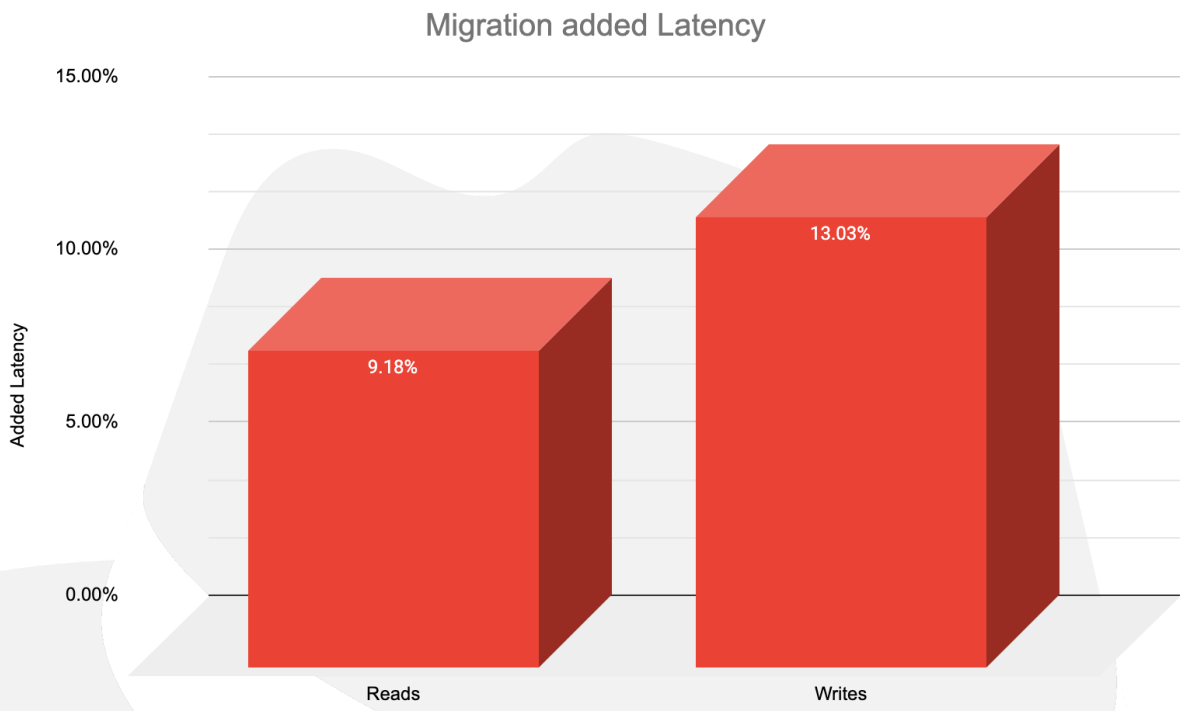
VMs migration added latency

In the following scenario, we tested a 1000 VMs migration, but this time we only used RHEL VMs. We decided to use only RHEL VMs to avoid any discrepancies that might occur due to the way different operating systems handle IO.

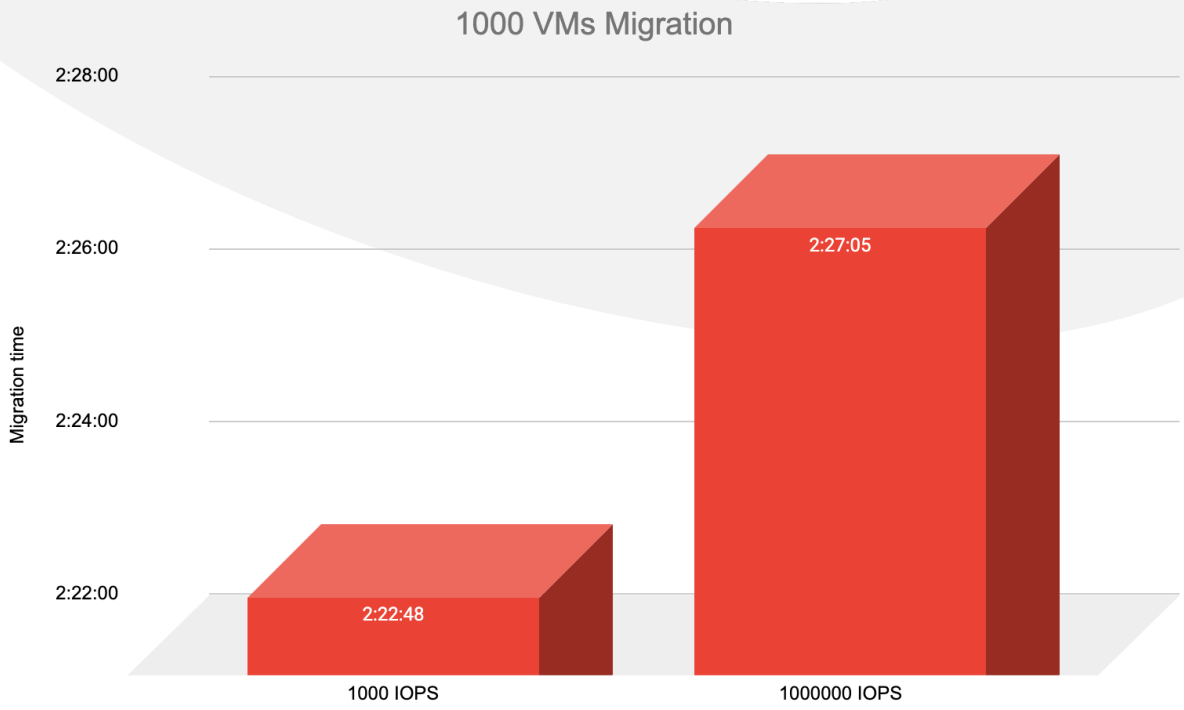
Like before, we choose to use the 4KB block size for both random reads and random writes, and we ran these tests:

- Baseline - each of the 1K VMs is generating a single IOPS to a total of 1K IOPS.
- Workload - each of the 1K VMs is generating 1000 IOPS per second to a total of one million IOPS.

The graph below shows the average latency penalty during migration for both reads and writes compared to the baseline (RHEL VMs only):



In addition, as we can see in the chart below, the migration time was impacted by 5%. Note that migration time results may vary due to BZ#2069098:



Cluster upgrade at scale

In the following scenario, we tested both minor and major upgrades.

We started with upgrading the cluster from version 4.9.15 to 4.9.23 while simulating a real production upgrade, meaning all 3000 VMs and 21,400 pods were running on the cluster. In addition, a light workload of 4KB was generated on 1500 VMs at a rate of 100 IOPS per VM.

We initiated the upgrade by running:

```
$ oc adm upgrade --to 4.9.23
```

The upgrade progress can be tracked using:

```
$ oc get clusterversion
```

```
NAME      VERSION   AVAILABLE   PROGRESSING   SINCE   STATUS
version   4.9.15    True        True          25m     Working towards
4.9.23: 569 of 738 done (77% complete)
```

The total duration for the minor upgrade process took a total of **35 minutes**.

The next step was testing a major upgrade by upgrading the cluster from version 4.9.23 to 4.10.9, again running the upgrade under the same conditions as before. We started the upgrade by running:

```
oc adm upgrade channel candidate-4.10 --allow-explicit-channel
oc adm upgrade --to 4.10.9  --allow-explicit-upgrade
```

Like before, the upgrade progress can be tracked using:

```
NAME      VERSION   AVAILABLE   PROGRESSING   SINCE   STATUS
version   4.9.23    True        True          40m     Working towards
4.10.9: 95 of 771 done (12% complete)
```

The total duration for the entire major upgrade process took a total of **136 minutes**.

Note that some upgrades might include modifications that will require all nodes to go through a soft reset, which will significantly increase upgrade time due to the added migration time.

Conclusion

In this reference architecture, we demonstrated the OpenShift Virtualization capabilities and resilience over a large scale. the OpenShift Virtualization feature of Red Hat OpenShift Container Platform, along with Red Hat Ceph Storage and/or Red Hat OpenShift Data Foundation can offer a complete production solution that incorporates containers, virtual machines, and high-availability storage, and it can be deployed on any host that meets the minimum hardware requirements.

While this reference architecture outlines how the set goal can be achieved, it is important to consider other architectures for appropriate resilience, scalability, and seamless daily operations given environmental conditions and requirements. For example, when crossing certain limits, a multicluster approach should be considered when the number of nodes or the workload becomes too large or too much churn is taking place on the cluster.

Additional resources

System and environment requirements:
https://docs.openshift.com/container-platform/3.11/install/prerequisites.html

OpenShift templates:
https://docs.openshift.com/container-platform/4.9/openshift_images/using-templates.html

Machine config operator:
https://docs.openshift.com/container-platform/4.9/post_installation_configuration/machine-configuration-tasks.html

Live migration and timeouts:
https://docs.openshift.com/container-platform/4.9/virt/live_migration/virt-live-migration-limits.html

Updating a cluster using the CLI:
https://docs.openshift.com/container-platform/4.10/updating/updating-cluster-cli.html

About Red Hat

Red Hat is the world's leading provider of enterprise open source software solutions, using a community-powered approach to deliver reliable and high-performing Linux, hybrid cloud, container, and Kubernetes technologies. Red Hat helps customers develop cloud-native applications, integrate existing and new IT applications, and automate and manage complex environments. A trusted adviser to the Fortune 500, Red Hat provides award-winning support, training, and consulting services that bring the benefits of open innovation to any industry. Red Hat is a connective hub in a global network of enterprises, partners, and communities, helping organizations grow, transform, and prepare for the digital future.