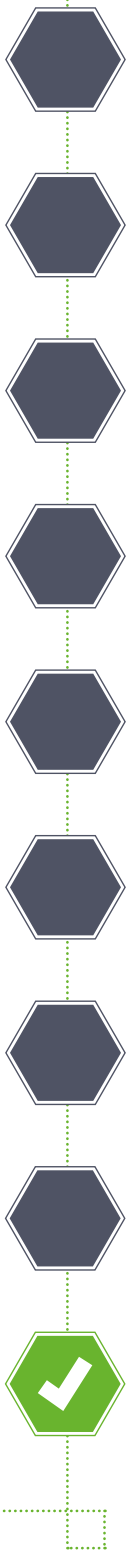
STORAGEOS



Self-Evaluation Guide.

Introduction

StorageOS Self-Evaluation Guide v1.1

The purpose of this document is to provide a step-by-step guide to testing StorageOS as such it provides a one size fits all guide to a StorageOS self-evaluation.

If you have any specific or more complex requirements please contact StorageOS as we'd be happy to organise a POC in conjunction with our Engineering team. You can join our slack channel or email us at: info@storageos.com

Table of Contents

Installation

The first phase of the self-evaluation is to install StorageOS. This section of the document aims to layout what options are exposed to you during installation and why some options may be preferable to you over others.

A standard StorageOS installations uses the StorageOS operator, so as much of the necessary configuration is handled for you. The StorageOS operator has been certified by Red Hat and is open source.

StorageOS Operator

The StorageOS operator is a Kubernetes native application that manages the StorageOS cluster lifecycle. It simplifies cluster installation, cluster removal and other operations.

The StorageOS operator watches for the creation of StorageOSCluster Custom Resources. A StorageOSCluster is a declarative representation of a StorageOS cluster. For example if CSI is enabled in the StorageOSCluster resource, a StorageOS cluster will be created that uses the CSI driver.

StorageOS Operator features

Specific configuration options for the StorageOS Operator that we believe to be important during a self-evaluation will be laid out in this guide.

A set of example StorageOSCluster are listed here. For an exhaustive list of configuration settings for the StorageOS operator please see our documentation.

Native Driver vs CSI

Communication between Kubernetes and StorageOS can use one of two drivers; the StorageOS Native Driver or the CSI (Container Storage Interface) driver. CSI provides a standardized interface for storage providers to use and is considered GA from Kubernetes 1.13. Therefore, StorageOS uses CSI as the default driver for Kubernetes 1.13+.

As the StorageOS native driver is implemented in the Kubernetes trunk by StorageOS, it is tied to Kubernetes releases. Whereas using CSI we can iterate more quickly and make improvements independent of Kubernetes releases.

Importantly CSI is only considered generally available in Kubernetes 1.13+ and is still in technology preview in Openshift 3.11.

External Etcd

StorageOS highly recommends an external etcd cluster is used for production deployments. In this configuration the etcd cluster would run on separate boxes from the rest of the Kubernetes and StorageOS cluster ensuring stability and resilience of the etcd cluster. However for the purposes of a self-evaluation it is acceptable to run etcd as a container inside Kubernetes.

We do not recommend running etcd on the same nodes as StorageOS when node failure will be tested, as if the majority of etcd nodes fail then the etcd cluster cannot be recovered automatically. Therefore it is better to run etcd on separate nodes.

Prerequisites

StorageOS has some prerequisites that must be met to complete a successful installation

- Machines intended to run StorageOS have at least 1 CPU core, 2GB RAM
- Docker 1.10 or later with mount propagation enabled
- TCP ports 5701-5710 and TCP & UDP 5711 open between all nodes in the cluster
- A 64bit supported operating system - By default StorageOS supports Debian 9, RancherOS, RHEL7.5 and CentOS7.

> Note Ubuntu 16.04 and 18.04 are supported but additional packages are required. Ubuntu 16.04/18.04 with the AWS kernel and Ubuntu 18.04 with the GCE kernel do not provide the required packages and are therefore *NOT* supported.

To install the required kernel modules on Ubuntu 16.04:

```
sudo apt -y update
sudo apt -y install linux-image-extra-$(uname -r)
```

To install the required kernel modules on Ubuntu 18.04+:

```
sudo apt -y update
sudo apt -y install linux-modules-extra-$(uname -r)
```

Installing StorageOS

Installation steps are as follows:

- Install etcd
- Install StorageOS Operator
- Create a Kubernetes secret detailing the default StorageOS administrator account
- Install StorageOS using a StorageOSCluster Custom Resource

Install etcd

In order to get a StorageOS cluster stood up quickly, a single node etcd cluster can be installed in Kubernetes, on a Kubernetes master. The reason for installing on a master is that master nodes generally have predictable lifetimes and low Pod scheduling churn. As such there is a lesser risk of the etcd pod being evicted ensuring a stable etcd cluster.

Note that if the etcd pod is stopped for any reason the etcd cluster will cease to function pending manual intervention. Please take this into account during testing of failure scenarios.

Etcd inside Kubernetes

1. Download repo

```
$ git clone https://github.com/coreos/etcd-operator.git
```

2. Configure NS, Role and RoleBinding

```
$ export ROLE_NAME=etcd-operator
$ export ROLE_BINDING_NAME=etcd-operator
$ export NAMESPACE=etcd
```

3. Create Namespace

```
$ kubectl create namespace $NAMESPACE
```

4. Deploy Operator

```
$ ./etcd-operator/example/rbac/create_role.sh
$ kubectl -n $NAMESPACE create -f ./etcd-operator/example/deployment.yaml
```

5. The Kubernetes masters should then be labelled so a nodeSelector can be used in the EtcdCluster manifest

```
$ kubectl label nodes <NODES> etcd-cluster=storageos-etcd
```

Once the master node taints are known and the nodes have been labelled you can deploy an EtcdCluster manifest that contains tolerations for all taints on the master nodes and selects for the node label applied in the previous step. A sample manifest is below. Edit the size to match the number of masters you will deploy on and edit the tolerations to match all taints on the master nodes where etcd will be deployed.

6. Create the EtcdCluster resource

```
$ kubectl -n etcd create -f - <<END
apiVersion: "etcd.database.coreos.com/v1beta2"
kind: "EtcdCluster"
metadata:
  name: "storageos-etcd"
spec:
  size: 1
  version: "3.3.13"
  pod:
    etcdEnv:
    - name: ETCD_QUOTA_BACKEND_BYTES
      value: "2147483648"  # 2 GB
    - name: ETCD_AUTO_COMPACTION_RETENTION
      value: "100" # Keep 100 revisions
    - name: ETCD_AUTO_COMPACTION_MODE
      value: "revision" # Set the revision mode
    resources:
      requests:
        cpu: 200m
        memory: 300Mi
    securityContext:
      runAsNonRoot: true
      runAsUser: 9000
      fsGroup: 9000
    tolerations:
    - operator: "Exists"
    nodeSelector:
      etcd-cluster: storageos-etcd
    affinity:
      podAntiAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 100
          podAffinityTerm:
            labelSelector:
              matchExpressions:
              - key: etcd_cluster
                operator: In
                values:
                - storageos-etcd
            topologyKey: kubernetes.io/hostname
END
```

Install StorageOS Operator

In order to install the StorageOS operator download the requisite yaml manifests or apply them with kubectl.

```
$ kubectl create -f https://github.com/storageos/cluster-operator/releases/download/1.4.0/storageos-operator.yaml
```

Install StorageOS

Once the StorageOS operator has been installed a StorageOS cluster can be generated by creating a StorageOSCluster resource.

A StorageOSCluster resource describes the state of the StorageOS cluster that is desired and the StorageOS operator will create the desired StorageOS cluster. For examples of StorageOSCluster resources please see our examples page here. For a full list of the configurable `spec` parameters of the StorageOSCluster resource please see here.

1. Create a secret defining the API username and password

```
$ kubectl create -f - <<END
apiVersion: v1
kind: Secret
metadata:
  name: "storageos-api"
  namespace: "default"
  labels:
    app: "storageos"
type: "kubernetes.io/storageos"
data:
  apiUsername: c3RvcmFnZW9z
  apiPassword: c3RvcmFnZW9z
END
```

2. Create a StorageOSCluster resource

```
$ kubectl create -f - <<END
apiVersion: "storageos.com/v1"
kind: StorageOSCluster
metadata:
  name: "storageos"
spec:
  secretRefName: "storageos-api"
  secretRefNamespace: "default"
  images:
    nodeContainer: "storageos/node:1.4.0" # StorageOS version
  resources:
    requests:
    memory: "512Mi"
  csi:
    enable: true
    deploymentStrategy: deployment
  kvBackend:
    address: 'storageos-etcd-client.etcd.svc:2379'
    backend: 'etcd'
END
```

3. Confirm that the cluster has been created and that StorageOS pods are running

```
$ kubectl -n storageos get pods
```

   StorageOS pods enter a ready state after a minimum of 65s has passed.

4. Deploy the StorageOS CLI as a container

```
$ kubectl -n storageos run         \
--image storageos/cli:1.2.2        \
--restart=Never                    \
--env STORAGEOS_HOST=storageos     \
--env STORAGEOS_USERNAME=storageos \
--env STORAGEOS_PASSWORD=storageos \
--command cli                      \
-- /bin/sleep 999999
```

5. Confirm that StorageOS is working by creating a PVC

```
$ kubectl create -f - <<END
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-1
  annotations:
    volume.beta.kubernetes.io/storage-class: fast
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
END
```

6. Verify that the CLI is working. `pvc-1` should be listed in the CLI output

```
$ kubectl -n storageos exec -it cli -- storageos volume ls
```

7. The StorageOS web UI can also be used to display information about the state of the cluster. The StorageOS UI can be accessed on any node that is running a StorageOS pod on port 5705. The username/password for the UI is defined by the `storageos-api` secret. For this self-evaluation the username/password is `storageos:storageos`

```
http://<NODE_IP>:5705
```

8. Create a pod that consumes the PVC

```
$ kubectl create -f - <<END
apiVersion: v1
kind: Pod
metadata:
  name: d1
spec:
  containers:
    - name: debian
      image: debian:9-slim
      command: ["/bin/sleep"]
      args: [ "3600" ]
      volumeMounts:
        - mountPath: /mnt
          name: v1
  volumes:
    - name: v1
      persistentVolumeClaim:
        claimName: pvc-1
END
```

9. Check that the pod starts successfully. If the pod starts successfully then the StorageOS cluster is working correctly

```
$ kubectl get pod d1 -w
```

The pod mounts a StorageOS volume under `/mnt` so any files written there will persist the lifetime of the pod. This can be demonstrated using the following commands.

10. Execute a shell inside the pod and write some data to a file

```
$ kubectl exec -it d1 -- bash
root@d1:/# echo Hello World! > /mnt/hello
root@d1:/# cat /mnt/hello
Hello World!
```

11. Delete the pod

```
$ kubectl delete pod d1
```

12. Recreate the pod

```
$ kubectl create -f - <<END
apiVersion: v1
kind: Pod
metadata:
  name: d1
spec:
  containers:
    - name: debian
      image: debian:9-slim
      command: ["/bin/sleep"]
      args: [ "3600" ]
      volumeMounts:
        - mountPath: /mnt
          name: v1
  volumes:
    - name: v1
      persistentVolumeClaim:
        claimName: pvc-1
END
```

13. Open a shell inside the pod and check the contents of `/mnt/hello`

```
$ kubectl exec -it -- cat /mnt/hello
Hello World!
```

Now that StorageOS has been successfully installed, the cluster has a standard license by default which allows for the creation of 100GB of persistent volumes. If you register the cluster then a developer license will be applied and 500GB of persistent volumes can be created. Replicas do not count towards the license total so a 500GB license could be used to created a 500GB volume with 5 replicas. For the purposes of this self-evaluation the standard license is sufficient.

Setup a Monitoring Stack

StorageOS exposes many metrics about the running of StorageOS and perhaps most importantly the read/write performance of StorageOS volumes. Each StorageOS pod exposes a Prometheus endpoint that exposes metrics; these can be visualized with something like Grafana.

A guided installation of Prometheus, using the Prometheus operator and Grafana, using helm, is available in our deploy repository.

1. To install Prometheus you can run the `install-prometheus` script

```
$ git clone https://github.com/storageos/deploy.git
$ cd deploy/k8s/examples/prometheus
$ ./install-prometheus.sh
```

2. Grafana can be installed using yaml manifests

```
$ ./install-grafana.sh
```

3. In order to view the Grafana or Prometheus UI create the NodePort services in the manifests folder

```
$ kubectl create -f manifest/prometheus/prometheus-svc.yaml.example
$ kubectl create -f manifest/grafana/grafana-svc.yaml.example
```

   The Grafana UI is then available on `<NODE_IP>:3000`. Instructions for retrieving the Grafana password are printed out when the Helm install is complete.

   Once logged in create the Prometheus data source by setting the URL to `http://prometheus-operated:9090` and configure the scrape interval to be 10s and set the query timeout to 30s. The StorageOS example dashboard can then be imported into Grafana.

4. To confirm the dashboard is working try writing some data to the volume that was created previously

```
$ kubectl exec -it -- bash -c 'dd if=/dev/zero of=/mnt/file count=262144 bs=4096'
```

The StorageOS dashboard will show that a volume is being written to, giving metrics for IOPS and bandwidth.

For more information about how to interpret the metrics that we expose please see our documentation about Monitoring StorageOS. And for a full overview of the metrics that we expose please refer to our Prometheus documentation.

We have also created a dashboard for monitoring etcd pods which can be found here. It is important to defragment etcd before the on disk space exceeds the database quota, see the etcd documentation for more information about etcd maintenance.

StorageOS Features

Now that you have a correctly functioning StorageOS cluster we will explain some of our features that may be of use to you as you complete application and synthetic benchmarks.

StorageOS features are all enabled/disabled by applying labels to volumes. These labels can be passed to StorageOS via persistent volume claims (PVCs) or can be applied to volumes using the StorageOS CLI or GUI.

The following is not an exhaustive feature list but outlines features which are commonly of use during a self-evaluation.

Volume Replication

StorageOS enables synchronous replication of volumes using the `storageos.com/replicas` label.

The volume that is active is referred to as the master volume. The master volume and its replicas are always placed on separate nodes. In fact if a replica cannot be placed on a node without a replica of the same volume, the volume will fail to be created. For example, in a three node StorageOS cluster a volume with 3 replicas cannot be created as the third replica cannot be placed on a node that doesn't already contain a replica of the same volume.

The failure mode for a volume affects how many failed replicas can be tolerated before the volume is marked as offline. Replicas are also segregated according to the `iaas/failure-domains` node label. StorageOS will automatically place a master volume and its replicas in separate failure domains where possible.

See our replication documentation for more information on volume replication.

1. To test volume replication create the following PersistentVolumeClaim

```
$ kubectl create -f - <<END
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: pvc-replicated
 labels:
   storageos.com/replicas: 1
 annotations:
   volume.beta.kubernetes.io/storage-class: fast
spec:
 accessModes:
   - ReadWriteOnce
 resources:
   requests:
     storage: 5Gi
END
```

> Note that volume replication is enabled by setting the `storageos.com/replicas` label on the volume.

2. Confirm that a replicated volume has been created by using the StorageOS CLI or UI

```
$ kubectl -n storageos exec -it cli -- storageos volume ls
```

3. Create a pod that uses the PVC

```
$ kubectl create -f - <<END
apiVersion: v1
kind: Pod
metadata:
 name: replicated-pod
spec:
 containers:
   - name: debian
     image: debian:9-slim
     command: ["/bin/sleep"]
     args: [ "3600"  ]
     volumeMounts:
       - mountPath: /mnt
         name: v1
 volumes:
   - name: v1
     persistentVolumeClaim:
       claimName: pvc-replicated
END
```

4. Write data to the volume

```
$ kubectl exec -it replicated-pod -- bash
root@replicated-pod:/# echo Hello World! > /mnt/hello
root@replicated-pod:/# cat /mnt/hello
Hello World!
```

5. Find the location of the master volume and drain the node using the StorageOS CLI. Draining a node causes all volumes on the node to be evicted. For replicated volumes this immediately promotes a replica to become the new master, and for unreplicated volumes a replica is created and fully synchronized before the volume fails over

```
$ kubectl get pvc
NAME           STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pvc-replicated Bound    pvc-29e2ad6e-8c4e-11e9-8356-027bfbbece86   5Gi        RWO            fast           1m
```

```
$ kubectl exec -it -n storageos cli -- storageos volume ls
NAMESPACE/NAME                                  SIZE   MOUNT           STATUS  REPLICAS  LOCATION
default/pvc-29e2ad6e-8c4e-11e9-8356-027bfbbece86  50GiB  ip-10-0-11-175  active  1/1       ip-10-0-11-167 (healthy)

$ kubectl exec -it -n storageos cli -- storageos node drain ip-10-0-11-167
ip-10-0-11-167
```

6. Check the location of the master volume and notice that it is on a new node

```
$ kubectl exec -it -n storageos cli -- storageos volume ls
NAMESPACE/NAME                                  SIZE   MOUNT           STATUS  REPLICAS  LOCATION
default/pvc-29e2ad6e-8c4e-11e9-8356-027bfbbece86  50GiB  ip-10-0-11-175  active  1/1       ip-10-0-11-189 (synching)
```

7. Check that the data is still accessible to the pod

```
$ kubectl exec -it replicated-pod -- bash
root@replicated-pod:/# cat /mnt/hello
Hello World!
```

Fencing

StorageOS enables fencing of pods using the `storageos.com/fenced=true` label. Pods must have the fencing label set and be using at least one StorageOS volume. Any StorageOS volume that the pod is using must have at least one healthy replica.

StatefulSets are the de facto controller for stateful workloads in Kubernetes. They provide a variety of useful guarantees but chief among them is that pods are unique. This guarantee means that if Kubernetes detects that a node is segregated from the master, StatefulSet pods will not be rescheduled unless the StatefulSet pods on the failed node are manually force terminated. However as StorageOS pods communicate via a gossip protocol, StorageOS can determine whether the node is truly offline or just partitioned from the master. In the case that the node is no longer participating in gossip, StorageOS can intervene and terminate StatefulSet pods that are using StorageOS volumes thus improving the time to recover for StatefulSets.

For more information about StatefulSets and fencing please see our Fencing concepts page. For information on how to enable Fencing see our Fencing operations page.

1. To test fencing create a StatefulSet from the StorageOS deploy repository

```
$ git clone https://github.com/storageos/deploy.git
$ kubectl create -f deploy/k8s/examples/fencing
```

2. Check what node the `mysql-0` pod is running on and make that node unavailable e.g. shutdown the node or stop the kubelet on the node. Now watch as the `mysql-0` pod is rescheduled onto a different node

```
$ kubectl get pods -l app=mysql -o wide
NAME      READY   STATUS    RESTARTS   AGE    IP           NODE                           NOMINATED NODE   READINESS GA
client    1/1     Running   0          1m     10.244.2.4   ip-10-1-10-112.storageos.net   <none>           <none>
mysql-0   1/1     Running   0          1m     10.244.1.6   ip-10-1-10-235.storageos.net   <none>           <none>
```

3. Once the node is in a NotReady state you'll see that the `mysql-0` pod has been rescheduled on a different node

```
$ kubectl get nodes
NAME                           STATUS     ROLES    AGE    VERSION
ip-10-1-10-112.storageos.net   Ready      master   107m   v1.14.3
ip-10-1-10-118.storageos.net   Ready      <none>   107m   v1.14.3
ip-10-1-10-235.storageos.net   NotReady   <none>   107m   v1.14.3
```

```
$ kubectl get pods -o wide
NAME      READY   STATUS        RESTARTS   AGE   IP           NODE                              NOMINATED NODE   READINESS
client    1/1     Running       0          1m    10.244.2.4   ip-10-1-10-112.storageos.net   <none>           <none>
mysql-0   1/1     Running       0          30s   10.244.1.6   ip-10-1-10-118.storageos.net   <none>           <none>
```

Benchmarking

As a rule the best performance is obtained by using unreplicated volumes that are co-located with the application or benchmarking tool writing to the volume. See Volume Placement below for more information.

When running benchmarks in the cloud, benchmarks need to be run multiple times and nodes should be destroyed and recreated so that the underlying machine changes. This should be done to reduce the impact that noisy neighbours might have on benchmark results.

Considerations

Application vs StorageOS replication

Certain applications are able to natively replicate or shard data between application instances. When using these applications it is worth considering whether application replication, StorageOS replication or a mixture of both should be used.

When StorageOS replication is enabled the time to recover in cases of node failure can be lessened. This is because StorageOS will promote a replica, Kubernetes will reschedule the application instance and the amount of data the application needs to catch up on is limited to whatever data was modified while the application was being rescheduled. Without StorageOS replication the application would have to rebuild an entire copy of data. Some applications have their performance greatly impacted by having to rebuild shards/replicas so this is also avoided.

StatefulSets

StatefulSets are the de facto controller for stateful applications. As such, when deploying applications that will use StorageOS volumes, StatefulSets should be used. You can find more information about StatefulSets here.

Volume Placement

StorageOS volumes give the best performance when the application pod and the master volume are co-located on the same node. When benchmarking applications, it is useful to take into account that using remote volumes and replicas impact the overall performance of a volume.

Going from 0 to 1 replica has the greatest performance impact for writes as now the latency of the operation is equal to the round trip time to the node with the replica over the network. Adding additional replicas poses less of a performance impact as writes to replicas are done in parallel, and the round trip time to each node is unlikely to greatly increase unless replicas land on nodes that are geographically distant to the master volumes' node.

Even when volumes are replicated co-location of pod and master volume is still desirable because application writes are first sent to the master and then sent from the master volume to the replicas. Writing to a local master therefore saves network latency between the application and the master volume. As reads are always served from the master volume a remote master volume will add latency to reads as well as writes.

When testing applications, such as databases, it is also necessary to run benchmarks for a sufficiently long time to account for caching, and cache flushing that databases do. We recommend running application benchmarks over a 20-30min period for this reason.

How to land a volume and a pod on the same node

StorageOS has an automatic co-location feature on our development roadmap that we are calling pod locality. Until the feature is GA co-location of a master volume and a pod can be achieved by leveraging existing StorageOS and Kubernetes features.

`storageos.com/hint.master` is a volume label that influences the placement of a StorageOS master volume. By setting this label to the same value as a `nodeSelector` on a StatefulSet or Pod the master volume and the pod should co-locate on the same node. You can reference our FIO local volumes job for an example of how to do this.

StorageOS Pools can be used to restrict volume placement to a subset of nodes. Nodes can be included in a specific pool by matching a pools `nodeSelector`. Pools can be created using the StorageOS GUI or CLI. The pool that a volume will be created from is specified in the StorageClass `pool` parameter.

It is also possible to cordon StorageOS nodes using the GUI or the CLI in order to force the placement of volumes on a specific node. A further possibility is to use of the `storageos.com/auto-follow` label. This label enables StorageOS to promote a replica volume to being a master when the pod and the replica volume are co-located.

Synthetic Benchmarks

Synthetic benchmarks using tools such as FIO are a useful way to begin measuring StorageOS performance. While not fully representative of application performance, they allow us to reason about the performance of storage devices without the added complexity of simulating real world workloads, and provide results easily comparable across platforms.

As with application benchmarks, when testing in public clouds multiple runs on newly created nodes should be considered to account for the impact of noisy neighbours.

StorageOS has created a test suite for running FIO tests against StorageOS volumes that can be found here. The test suite can be deployed into a Kubernetes cluster using the instructions below.

1. Clone the StorageOS deploy repo

```
$ git clone https://github.com/storageos/deploy.git
```

2. Move into the FIO local-volumes folder

```
$ cd ./deploy/k8s/examples/FIO/local-volumes
```

3. Get the name of the node that you wish the FIO pods and volumes to be created on. Make sure that the node name and the label `kubernetes.io/hostname` match and that the node has enough storage capacity to create 8Gi worth of volumes

```
$ kubectl get node --show-labels
```

4. Generate the FIO jobs by passing in the node name that the job should run on. The number is the number of volumes that FIO will test concurrently

```
$ ./job-generator-per-volumecount.sh 4 $NODE_NAME
```

5. Upload the FIO profiles as ConfigMaps

```
$ ./upload-fio-profiles.sh
```

6. Run the FIO tests

```
$ kubectl create -f ./jobs
```

7. Check the PVCs have been provisioned

```
$ kubectl get pvc
```

8. Use the StorageOS CLI to check the location of the volumes

```
$ kubectl -n storageos exec cli -- storageos volume ls
```

9. Verify that the Pod is running on the same node

```
$ kubectl get pod -owide
```

FIO has a number of parameters that can be adjusted to simulate a variety of workloads and configurations. Particularly the queue depth, block size and the number of volumes used affect the FIO results. To tune the FIO parameters the profiles file can be edited or the ConfigMap that is created from the profiles file can be edited directly.

StorageOS configuration also affects the overall volume performance. For example adding a replica to a volume will increase the latency for writes and affect IOPS and bandwidth for the volume.

To see the effect a StorageOS replica has on performance rerun an FIO test but add the `storageos.com/replicas: "1"` label to the PersistentVolumeClaims in the jobs spec. The greatest performance impact from adding replicas comes when moving from 0 to 1 replica. Adding additional replicas does not incur a significant performance penalty.

The remote volumes folder contains a guide for performing the same FIO tests against remote volumes.

Application Benchmarks

While synthetic benchmarks are useful for examining the behaviour of StorageOS with very specific workloads, in order to get a realistic picture of StorageOS performance actual applications should be tested.

Many applications come with test suites which provide standard workloads. For best results, test using your application of choice with a representative configuration and real world data.

As an example of benchmarking an application the following steps lay out how to benchmark a Postgres database backed by a StorageOS volume.

1. Start by cloning the StorageOS deploy repository

```
$ git clone https://github.com/storageos/deploy.git storageos
$ git checkout pgbench
```

2. Move into the Postgres examples folder

```
$ cd storageos/k8s/examples/postgres
```

3. Decide which node you want the Postgres pod and volume to be located on. The node needs to be labelled `app=postgres`

```
$ kubectl label node <NODE> app=postgres
```

4. Then set the `storageos.com/hint.master` label in 20-postgres-statefulset.yaml file to match the node name you have chosen before creating all the files

```
$ kubectl create -f ../postgres
```

5. Confirm that Postgres is up and running

```
$ kubectl get pods -w -l app=postgres
```

6. Use the StorageOS CLI or the GUI to check the master volume location and the mount location. They should match

```
$ kubectl -n storageos exec -it cli -- storageos v ls
```

7. Exec into the pgbench container and run pgbench

```
$ kubectl exec -it pgbench -- bash -c '/opt/cpm/bin/start.sh'
```

Conclusion

After completing these steps you will have benchmark scores for StorageOS. Please keep in mind that benchmarks are only part of the story and that there is no replacement for testing actual production or production like workloads.

StorageOS invites you to provide feedback on your self-evaluation to the slack channel or by directly emailing us at info@storageos.com