# Scale a StatefulSet

This task shows how to scale a StatefulSet. Scaling a StatefulSet refers to increasing or decreasing the number of replicas.

## Before you begin

- StatefulSets are only available in Kubernetes version 1.5 or later. To check your version of Kubernetes, run `kubectl version`.
- Not all stateful applications scale nicely. If you are unsure about whether to scale your StatefulSets, see StatefulSet concepts or StatefulSet tutorial for further information.
- You should perform scaling only when you are confident that your stateful application cluster is completely healthy.

## Scaling StatefulSets

### Use kubectl to scale StatefulSets

First, find the StatefulSet you want to scale.

```
kubectl get statefulsets <stateful-set-name>
```

Change the number of replicas of your StatefulSet:

```
kubectl scale statefulsets <stateful-set-name> --replicas=<new-replicas>
```

### Make in-place updates on your StatefulSets

Alternatively, you can do in-place updates on your StatefulSets.

If your StatefulSet was initially created with `kubectl apply`, update `.spec.replicas` of the StatefulSet manifests, and then do a `kubectl apply`:

```
kubectl apply -f <stateful-set-file-updated>
```

Otherwise, edit that field with `kubectl edit`:

```
kubectl edit statefulsets <stateful-set-name>
```

Or use `kubectl patch`:

```
kubectl patch statefulsets <stateful-set-name> -p '{"spec":{"replicas":<new-replicas>}}'
```

## Troubleshooting

### Scaling down does not work right

You cannot scale down a StatefulSet when any of the stateful Pods it manages is unhealthy. Scaling down only takes place after those stateful Pods become running and ready.

If spec.replicas > 1, Kubernetes cannot determine the reason for an unhealthy Pod. It might be the result of a permanent fault or of a transient fault. A transient fault can be caused by a restart required by upgrading or maintenance.

If the Pod is unhealthy due to a permanent fault, scaling without correcting the fault may lead to a state where the StatefulSet membership drops below a certain minimum number of replicas that are needed to function correctly. This may cause your StatefulSet to become unavailable.

If the Pod is unhealthy due to a transient fault and the Pod might become available again, the transient error may interfere with your scale-up or scale-down operation. Some distributed databases have issues when nodes join and leave at the same time. It is better to reason about scaling operations at the application level in these cases, and perform scaling only when you are sure that your stateful application cluster is completely healthy.

## What's next

- Learn more about deleting a StatefulSet.

# Run a Single-Instance Stateful Application

This page shows you how to run a single-instance stateful application in Kubernetes using a PersistentVolume and a Deployment. The application is MySQL.

## Objectives

- Create a PersistentVolume referencing a disk in your environment.
- Create a MySQL Deployment.
- Expose MySQL to other pods in the cluster at a known DNS name.

## Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

    - iximiuz Labs
    - Killercoda
    - KodeKloud
    - Play with Kubernetes

    To check the version, enter `kubectl version`.

- You need to either have a dynamic PersistentVolume provisioner with a default StorageClass, or statically provision PersistentVolumes yourself to satisfy the PersistentVolumeClaims used here.

## Deploy MySQL

You can run a stateful application by creating a Kubernetes Deployment and connecting it to an existing PersistentVolume using a PersistentVolumeClaim. For example, this YAML file describes a Deployment that runs MySQL and references the PersistentVolumeClaim. The file defines a volume mount for /var/lib/mysql, and then creates a PersistentVolumeClaim that looks for a 20G volume. This claim is satisfied by any existing volume that meets the requirements, or by a dynamic provisioner.

Note: The password is defined in the config yaml, and this is insecure. See Kubernetes Secrets for a secure solution.

`application/mysql/mysql-deployment.yaml` Copy application/mysql/mysql-deployment.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  name: mysqlspec:  ports:  - port: 3306  selector:    app: mysql  clusterIP: None---apiVersion: apps/v1kind
```

`application/mysql/mysql-pv.yaml` Copy application/mysql/mysql-pv.yaml to clipboard

```
apiVersion: v1
kind: PersistentVolumemetadata:  name: mysql-pv-volume  labels:    type: localspec:  storageClassName: manual  capacity:    storag
```

1. Deploy the PV and PVC of the YAML file:

    ```
    kubectl apply -f https://k8s.io/examples/application/mysql/mysql-pv.yaml
    ```

2. Deploy the contents of the YAML file:

    ```
    kubectl apply -f https://k8s.io/examples/application/mysql/mysql-deployment.yaml
    ```

3. Display information about the Deployment:

    ```
    kubectl describe deployment mysql
    ```

    The output is similar to this:

    ```
    Name:                 mysql
    Namespace:            default
    CreationTimestamp:    Tue, 01 Nov 2016 11:18:45 -0700
    Labels:               app=mysql
    Annotations:          deployment.kubernetes.io/revision=1
    Selector:             app=mysql
    Replicas:             1 desired | 1 updated | 1 total | 0 available | 1 unavailable
    StrategyType:         Recreate
    MinReadySeconds:      0
    Pod Template:
      Labels:       app=mysql
      Containers:
        mysql:
        Image:      mysql:9
        Port:       3306/TCP
        Environment:
          MYSQL_ROOT_PASSWORD:      password
        Mounts:
          /var/lib/mysql from mysql-persistent-storage (rw)
      Volumes:
        mysql-persistent-storage:
        Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)
        ClaimName:  mysql-pv-claim
        ReadOnly:   false
    Conditions:
      Type          Status  Reason
      ----          ------  ------
      Available     False   MinimumReplicasUnavailable
      Progressing   True    ReplicaSetUpdated
    OldReplicaSets:       <none>
    NewReplicaSet:        mysql-63082529 (1/1 replicas created)
    Events:
      FirstSeen    LastSeen    Count    From                SubobjectPath    Type        Reason            Message
      ---------    --------    -----    ----                -------------    --------    ------            -------
      33s          33s         1        {deployment-controller }             Normal      ScalingReplicaSet Scaled up replica set
    ```

4. List the pods created by the Deployment:

    ```
    kubectl get pods -l app=mysql
    ```

The output is similar to this:

```
NAME                  READY     STATUS    RESTARTS   AGE
mysql-63082529-2z3ki   1/1       Running   0          3m
```

5. Inspect the PersistentVolumeClaim:

```
kubectl describe pvc mysql-pv-claim
```

The output is similar to this:

```
Name:          mysql-pv-claim
Namespace:     default
StorageClass:
Status:        Bound
Volume:        mysql-pv-volume
Labels:        <none>
Annotations:     pv.kubernetes.io/bind-completed=yes
                 pv.kubernetes.io/bound-by-controller=yes
Capacity:      20Gi
Access Modes:  RWO
Events:        <none>
```

## Accessing the MySQL instance

The preceding YAML file creates a service that allows other Pods in the cluster to access the database. The Service option `clusterIP: None` lets the Service DNS name resolve directly to the Pod's IP address. This is optimal when you have only one Pod behind a Service and you don't intend to increase the number of Pods.

Run a MySQL client to connect to the server:

```
kubectl run -it --rm --image=mysql:9 --restart=Never mysql-client -- mysql -h mysql -ppassword
```

This command creates a new Pod in the cluster running a MySQL client and connects it to the server through the Service. If it connects, you know your stateful MySQL database is up and running.

```
Waiting for pod default/mysql-client-274442439-zyp6i to be running, status is Pending, pod ready: false
If you don't see a command prompt, try pressing enter.

mysql>
```

## Updating

The image or any other part of the Deployment can be updated as usual with the `kubectl apply` command. Here are some precautions that are specific to stateful apps:

- Don't scale the app. This setup is for single-instance apps only. The underlying PersistentVolume can only be mounted to one Pod. For clustered stateful apps, see the StatefulSet documentation.
- Use `strategy: type: Recreate` in the Deployment configuration YAML file. This instructs Kubernetes to *not* use rolling updates. Rolling updates will not work, as you cannot have more than one Pod running at a time. The `Recreate` strategy will stop the first pod before creating a new one with the updated configuration.

## Deleting a deployment

Delete the deployed objects by name:

```
kubectl delete deployment,svc mysql
kubectl delete pvc mysql-pv-claim
kubectl delete pv mysql-pv-volume
```

If you manually provisioned a PersistentVolume, you also need to manually delete it, as well as release the underlying resource. If you used a dynamic provisioner, it automatically deletes the PersistentVolume when it sees that you deleted the PersistentVolumeClaim. Some dynamic provisioners (such as those for EBS and PD) also release the underlying resource upon deleting the PersistentVolume.

## What's next

- Learn more about Deployment objects.
- Learn more about Deploying applications
- kubectl run documentation
- Volumes and Persistent Volumes

# Run a Replicated Stateful Application

This page shows how to run a replicated stateful application using a StatefulSet. This application is a replicated MySQL database. The example topology has a single primary server and multiple replicas, using asynchronous row-based replication.

**Note:**

**This is not a production configuration**. MySQL settings remain on insecure defaults to keep the focus on general patterns for running stateful applications in Kubernetes.

## Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:
    - iximiuz Labs
    - Killercoda
    - KodeKloud
    - Play with Kubernetes
- You need to either have a dynamic PersistentVolume provisioner with a default StorageClass, or statically provision PersistentVolumes yourself to satisfy the PersistentVolumeClaims used here.
- This tutorial assumes you are familiar with PersistentVolumes and StatefulSets, as well as other core concepts like Pods, Services, and ConfigMaps.
- Some familiarity with MySQL helps, but this tutorial aims to present general patterns that should be useful for other systems.
- You are using the default namespace or another namespace that does not contain any conflicting objects.
- You need to have a AMD64-compatible CPU.

## Objectives

- Deploy a replicated MySQL topology with a StatefulSet.
- Send MySQL client traffic.
- Observe resistance to downtime.
- Scale the StatefulSet up and down.

## Deploy MySQL

The example MySQL deployment consists of a ConfigMap, two Services, and a StatefulSet.

### Create a ConfigMap

Create the ConfigMap from the following YAML configuration file:

application/mysql/mysql-configmap.yaml Copy application/mysql/mysql-configmap.yaml to clipboard

```
apiVersion: v1
kind: ConfigMapmetadata:  name: mysql  labels:    app: mysql    app.kubernetes.io/name: mysqldata:  primary.cnf: |    # Apply this
```

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-configmap.yaml
```

This ConfigMap provides `my.cnf` overrides that let you independently control configuration on the primary MySQL server and its replicas. In this case, you want the primary server to be able to serve replication logs to replicas and you want replicas to reject any writes that don't come via replication.

There's nothing special about the ConfigMap itself that causes different portions to apply to different Pods. Each Pod decides which portion to look at as it's initializing, based on information provided by the StatefulSet controller.

### Create Services

Create the Services from the following YAML configuration file:

application/mysql/mysql-services.yaml Copy application/mysql/mysql-services.yaml to clipboard

```
# Headless service for stable DNS entries of StatefulSet members.
apiVersion: v1kind: Servicemetadata:  name: mysql  labels:    app: mysql    app.kubernetes.io/name: mysqlspec:  ports:  - name: my
```

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-services.yaml
```

The headless Service provides a home for the DNS entries that the StatefulSet controllers creates for each Pod that's part of the set. Because the headless Service is named `mysql`, the Pods are accessible by resolving `<pod-name>.mysql` from within any other Pod in the same Kubernetes cluster and namespace.

The client Service, called `mysql-read`, is a normal Service with its own cluster IP that distributes connections across all MySQL Pods that report being Ready. The set of potential endpoints includes the primary MySQL server and all replicas.

Note that only read queries can use the load-balanced client Service. Because there is only one primary MySQL server, clients should connect directly to the primary MySQL Pod (through its DNS entry within the headless Service) to execute writes.

### Create the StatefulSet

Finally, create the StatefulSet from the following YAML configuration file:

application/mysql/mysql-statefulset.yaml Copy application/mysql/mysql-statefulset.yaml to clipboard

```
apiVersion: apps/v1
kind: StatefulSetmetadata:  name: mysqlspec:  selector:    matchLabels:      app: mysql      app.kubernetes.io/name: mysql  servic
```

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-statefulset.yaml
```

You can watch the startup progress by running:

```
kubectl get pods -l app=mysql --watch
```

After a while, you should see all 3 Pods become `Running`:

```
NAME      READY     STATUS    RESTARTS   AGE
mysql-0   2/2       Running   0          2m
mysql-1   2/2       Running   0          1m
mysql-2   2/2       Running   0          1m
```

Press **Ctrl+C** to cancel the watch.

**Note:**

If you don't see any progress, make sure you have a dynamic PersistentVolume provisioner enabled, as mentioned in the prerequisites.

This manifest uses a variety of techniques for managing stateful Pods as part of a StatefulSet. The next section highlights some of these techniques to explain what happens as the StatefulSet creates Pods.

## Understanding stateful Pod initialization

The StatefulSet controller starts Pods one at a time, in order by their ordinal index. It waits until each Pod reports being Ready before starting the next one.

In addition, the controller assigns each Pod a unique, stable name of the form `<statefulset-name>-<ordinal-index>`, which results in Pods named `mysql-0`, `mysql-1`, and `mysql-2`.

The Pod template in the above StatefulSet manifest takes advantage of these properties to perform orderly startup of MySQL replication.

### Generating configuration

Before starting any of the containers in the Pod spec, the Pod first runs any init containers in the order defined.

The first init container, named `init-mysql`, generates special MySQL config files based on the ordinal index.

The script determines its own ordinal index by extracting it from the end of the Pod name, which is returned by the `hostname` command. Then it saves the ordinal (with a numeric offset to avoid reserved values) into a file called `server-id.cnf` in the MySQL `conf.d` directory. This translates the unique, stable identity provided by the StatefulSet into the domain of MySQL server IDs, which require the same properties.

The script in the `init-mysql` container also applies either `primary.cnf` or `replica.cnf` from the ConfigMap by copying the contents into `conf.d`. Because the example topology consists of a single primary MySQL server and any number of replicas, the script assigns ordinal `0` to be the primary server, and everyone else to be replicas. Combined with the StatefulSet controller's deployment order guarantee, this ensures the primary MySQL server is Ready before creating replicas, so they can begin replicating.

### Cloning existing data

In general, when a new Pod joins the set as a replica, it must assume the primary MySQL server might already have data on it. It also must assume that the replication logs might not go all the way back to the beginning of time. These conservative assumptions are the key to allow a running StatefulSet to scale up and down over time, rather than being fixed at its initial size.

The second init container, named `clone-mysql`, performs a clone operation on a replica Pod the first time it starts up on an empty PersistentVolume. That means it copies all existing data from another running Pod, so its local state is consistent enough to begin replicating from the primary server.

MySQL itself does not provide a mechanism to do this, so the example uses a popular open-source tool called Percona XtraBackup. During the clone, the source MySQL server might suffer reduced performance. To minimize impact on the primary MySQL server, the script instructs each Pod to clone from the Pod whose ordinal index is one lower. This works because the StatefulSet controller always ensures Pod `N` is Ready before starting Pod `N+1`.

### Starting replication

After the init containers complete successfully, the regular containers run. The MySQL Pods consist of a `mysql` container that runs the actual `mysqld` server, and an `xtrabackup` container that acts as a sidecar.

The `xtrabackup` sidecar looks at the cloned data files and determines if it's necessary to initialize MySQL replication on the replica. If so, it waits for `mysqld` to be ready and then executes the `CHANGE MASTER TO` and `START SLAVE` commands with replication parameters extracted from the XtraBackup clone files.

Once a replica begins replication, it remembers its primary MySQL server and reconnects automatically if the server restarts or the connection dies. Also, because replicas look for the primary server at its stable DNS name (`mysql-0.mysql`), they automatically find the primary server even if it gets a new Pod IP due to being rescheduled.

Lastly, after starting replication, the `xtrabackup` container listens for connections from other Pods requesting a data clone. This server remains up indefinitely in case the StatefulSet scales up, or in case the next Pod loses its PersistentVolumeClaim and needs to redo the clone.

## Sending client traffic

You can send test queries to the primary MySQL server (hostname `mysql-0.mysql`) by running a temporary container with the `mysql:5.7` image and running the `mysql` client binary.

```
kubectl run mysql-client --image=mysql:5.7 -i --rm --restart=Never --\
  mysql -h mysql-0.mysql <<EOFCREATE DATABASE test;CREATE TABLE test.messages (message VARCHAR(250));INSERT INTO test.messages VAL
```

Use the hostname `mysql-read` to send test queries to any server that reports being Ready:

```
kubectl run mysql-client --image=mysql:5.7 -i -t --rm --restart=Never --\
  mysql -h mysql-read -e "SELECT * FROM test.messages"
```

You should get output like this:

```
Waiting for pod default/mysql-client to be running, status is Pending, pod ready: false
+---------+
| message |
+---------+
| hello   |
+---------+
pod "mysql-client" deleted
```

To demonstrate that the `mysql-read` Service distributes connections across servers, you can run `SELECT @@server_id` in a loop:

```
kubectl run mysql-client-loop --image=mysql:5.7 -i -t --rm --restart=Never --\
  bash -ic "while sleep 1; do mysql -h mysql-read -e 'SELECT @@server_id,NOW()'; done"
```

You should see the reported `@@server_id` change randomly, because a different endpoint might be selected upon each connection attempt:

```
+-------------+---------------------+
| @@server_id | NOW()               |
+-------------+---------------------+
|         100 | 2006-01-02 15:04:05 |
+-------------+---------------------+
+-------------+---------------------+
| @@server_id | NOW()               |
+-------------+---------------------+
|         102 | 2006-01-02 15:04:06 |
+-------------+---------------------+
+-------------+---------------------+
| @@server_id | NOW()               |
+-------------+---------------------+
|         101 | 2006-01-02 15:04:07 |
+-------------+---------------------+
```

You can press **Ctrl+C** when you want to stop the loop, but it's useful to keep it running in another window so you can see the effects of the following steps.

## Simulate Pod and Node failure

To demonstrate the increased availability of reading from the pool of replicas instead of a single server, keep the `SELECT @@server_id` loop from above running while you force a Pod out of the Ready state.

### Break the Readiness probe

The readiness probe for the `mysql` container runs the command `mysql -h 127.0.0.1 -e 'SELECT 1'` to make sure the server is up and able to execute queries.

One way to force this readiness probe to fail is to break that command:

```
kubectl exec mysql-2 -c mysql -- mv /usr/bin/mysql /usr/bin/mysql.off
```

This reaches into the actual container's filesystem for Pod `mysql-2` and renames the `mysql` command so the readiness probe can't find it. After a few seconds, the Pod should report one of its containers as not Ready, which you can check by running:

```
kubectl get pod mysql-2
```

Look for `1/2` in the `READY` column:

```
NAME      READY     STATUS    RESTARTS   AGE
mysql-2   1/2       Running   0          3m
```

At this point, you should see your `SELECT @@server_id` loop continue to run, although it never reports `102` anymore. Recall that the `init-mysql` script defined `server-id` as `100 + $ordinal`, so server ID `102` corresponds to Pod `mysql-2`.

Now repair the Pod and it should reappear in the loop output after a few seconds:

```
kubectl exec mysql-2 -c mysql -- mv /usr/bin/mysql.off /usr/bin/mysql
```

### Delete Pods

The StatefulSet also recreates Pods if they're deleted, similar to what a ReplicaSet does for stateless Pods.

```
kubectl delete pod mysql-2
```

The StatefulSet controller notices that no `mysql-2` Pod exists anymore, and creates a new one with the same name and linked to the same PersistentVolumeClaim. You should see server ID `102` disappear from the loop output for a while and then return on its own.

### Drain a Node

If your Kubernetes cluster has multiple Nodes, you can simulate Node downtime (such as when Nodes are upgraded) by issuing a drain.

First determine which Node one of the MySQL Pods is on:

```
kubectl get pod mysql-2 -o wide
```

The Node name should show up in the last column:

```
NAME      READY     STATUS    RESTARTS   AGE       IP            NODE
mysql-2   2/2       Running   0          15m       10.244.5.27   kubernetes-node-9l2t
```

Then, drain the Node by running the following command, which cordons it so no new Pods may schedule there, and then evicts any existing Pods. Replace `<node-name>` with the name of the Node you found in the last step.

**Caution:**

Draining a Node can impact other workloads and applications running on the same node. Only perform the following step in a test cluster.

```
# See above advice about impact on other workloads
kubectl drain <node-name> --force --delete-emptydir-data --ignore-daemonsets
```

Now you can watch as the Pod reschedules on a different Node:

```
kubectl get pod mysql-2 -o wide --watch
```

It should look something like this:

```
NAME      READY   STATUS          RESTARTS   AGE       IP            NODE
mysql-2   2/2     Terminating     0          15m       10.244.1.56   kubernetes-node-9l2t
[...]
mysql-2   0/2     Pending         0          0s        <none>        kubernetes-node-fjlm
mysql-2   0/2     Init:0/2        0          0s        <none>        kubernetes-node-fjlm
mysql-2   0/2     Init:1/2        0          20s       10.244.5.32   kubernetes-node-fjlm
mysql-2   0/2     PodInitializing 0          21s       10.244.5.32   kubernetes-node-fjlm
mysql-2   1/2     Running         0          22s       10.244.5.32   kubernetes-node-fjlm
mysql-2   2/2     Running         0          30s       10.244.5.32   kubernetes-node-fjlm
```

And again, you should see server ID `102` disappear from the `SELECT @@server_id` loop output for a while and then return.

Now uncordon the Node to return it to a normal state:

```
kubectl uncordon <node-name>
```

## Scaling the number of replicas

When you use MySQL replication, you can scale your read query capacity by adding replicas. For a StatefulSet, you can achieve this with a single command:

```
kubectl scale statefulset mysql  --replicas=5
```

Watch the new Pods come up by running:

```
kubectl get pods -l app=mysql --watch
```

Once they're up, you should see server IDs `103` and `104` start appearing in the `SELECT @@server_id` loop output.

You can also verify that these new servers have the data you added before they existed:

```
kubectl run mysql-client --image=mysql:5.7 -i -t --rm --restart=Never --\
  mysql -h mysql-3.mysql -e "SELECT * FROM test.messages"
```

```
Waiting for pod default/mysql-client to be running, status is Pending, pod ready: false
+---------+
| message |
+---------+
| hello   |
+---------+
pod "mysql-client" deleted
```

Scaling back down is also seamless:

```
kubectl scale statefulset mysql --replicas=3
```

**Note:**

Although scaling up creates new PersistentVolumeClaims automatically, scaling down does not automatically delete these PVCs.

This gives you the choice to keep those initialized PVCs around to make scaling back up quicker, or to extract data before deleting them.

You can see this by running:

```
kubectl get pvc -l app=mysql
```

Which shows that all 5 PVCs still exist, despite having scaled the StatefulSet down to 3:

```
NAME           STATUS    VOLUME                                     CAPACITY   ACCESSMODES   AGE
data-mysql-0   Bound     pvc-8acbf5dc-b103-11e6-93fa-42010a800002   10Gi       RWO           20m
data-mysql-1   Bound     pvc-8ad39820-b103-11e6-93fa-42010a800002   10Gi       RWO           20m
data-mysql-2   Bound     pvc-8ad69a6d-b103-11e6-93fa-42010a800002   10Gi       RWO           20m
data-mysql-3   Bound     pvc-50043c45-b1c5-11e6-93fa-42010a800002   10Gi       RWO           2m
data-mysql-4   Bound     pvc-500a9957-b1c5-11e6-93fa-42010a800002   10Gi       RWO           2m
```

If you don't intend to reuse the extra PVCs, you can delete them:

```
kubectl delete pvc data-mysql-3
kubectl delete pvc data-mysql-4
```

## Cleaning up

1. Cancel the `SELECT @@server_id` loop by pressing **Ctrl+C** in its terminal, or running the following from another terminal:

   ```
   kubectl delete pod mysql-client-loop --now
   ```

2. Delete the StatefulSet. This also begins terminating the Pods.

   ```
   kubectl delete statefulset mysql
   ```

3. Verify that the Pods disappear. They might take some time to finish terminating.

   ```
   kubectl get pods -l app=mysql
   ```

   You'll know the Pods have terminated when the above returns:

   ```
   No resources found.
   ```

4. Delete the ConfigMap, Services, and PersistentVolumeClaims.

   ```
   kubectl delete configmap,service,pvc -l app=mysql
   ```

5. If you manually provisioned PersistentVolumes, you also need to manually delete them, as well as release the underlying resources. If you used a dynamic provisioner, it automatically deletes the PersistentVolumes when it sees that you deleted the PersistentVolumeClaims. Some dynamic provisioners (such as those for EBS and PD) also release the underlying resources upon deleting the PersistentVolumes.

## What's next

- Learn more about scaling a StatefulSet.
- Learn more about debugging a StatefulSet.
- Learn more about deleting a StatefulSet.
- Learn more about force deleting StatefulSet Pods.
- Look in the Helm Charts repository for other stateful application examples.

---

# Specifying a Disruption Budget for your Application

FEATURE STATE: `Kubernetes v1.21 [stable]`

This page shows how to limit the number of concurrent disruptions that your application experiences, allowing for higher availability while permitting the cluster administrator to manage the clusters nodes.

## Before you begin

Your Kubernetes server must be at or later than version v1.21.

To check the version, enter `kubectl version`.

- You are the owner of an application running on a Kubernetes cluster that requires high availability.
- You should know how to deploy Replicated Stateless Applications and/or Replicated Stateful Applications.
- You should have read about Pod Disruptions.
- You should confirm with your cluster owner or service provider that they respect Pod Disruption Budgets.

## Protecting an Application with a PodDisruptionBudget

1. Identify what application you want to protect with a PodDisruptionBudget (PDB).
2. Think about how your application reacts to disruptions.
3. Create a PDB definition as a YAML file.
4. Create the PDB object from the YAML file.

## Identify an Application to Protect

The most common use case when you want to protect an application specified by one of the built-in Kubernetes controllers:

- Deployment
- ReplicationController
- ReplicaSet
- StatefulSet

In this case, make a note of the controller's `.spec.selector`; the same selector goes into the PDBs `.spec.selector`.

From version 1.15 PDBs support custom controllers where the scale subresource is enabled.

You can also use PDBs with pods which are not controlled by one of the above controllers, or arbitrary groups of pods, but there are some restrictions, described in Arbitrary workloads and arbitrary selectors.

## Think about how your application reacts to disruptions

Decide how many instances can be down at the same time for a short period due to a voluntary disruption.

- Stateless frontends:
  - Concern: don't reduce serving capacity by more than 10%.
    - Solution: use PDB with minAvailable 90% for example.
- Single-instance Stateful Application:
  - Concern: do not terminate this application without talking to me.
    - Possible Solution 1: Do not use a PDB and tolerate occasional downtime.
    - Possible Solution 2: Set PDB with maxUnavailable=0. Have an understanding (outside of Kubernetes) that the cluster operator needs to consult you before termination. When the cluster operator contacts you, prepare for downtime, and then delete the PDB to indicate readiness for disruption. Recreate afterwards.
- Multiple-instance Stateful application such as Consul, ZooKeeper, or etcd:
  - Concern: Do not reduce number of instances below quorum, otherwise writes fail.
    - Possible Solution 1: set maxUnavailable to 1 (works with varying scale of application).
    - Possible Solution 2: set minAvailable to quorum-size (e.g. 3 when scale is 5). (Allows more disruptions at once).
- Restartable Batch Job:
  - Concern: Job needs to complete in case of voluntary disruption.
    - Possible solution: Do not create a PDB. The Job controller will create a replacement pod.

### Rounding logic when specifying percentages

Values for `minAvailable` or `maxUnavailable` can be expressed as integers or as a percentage.

- When you specify an integer, it represents a number of Pods. For instance, if you set `minAvailable` to 10, then 10 Pods must always be available, even during a disruption.
- When you specify a percentage by setting the value to a string representation of a percentage (eg. `"50%"`), it represents a percentage of total Pods. For instance, if you set `minAvailable` to `"50%"`, then at least 50% of the Pods remain available during a disruption.

When you specify the value as a percentage, it may not map to an exact number of Pods. For example, if you have 7 Pods and you set `minAvailable` to `"50%"`, it's not immediately obvious whether that means 3 Pods or 4 Pods must be available. Kubernetes rounds up to the nearest integer, so in this case, 4 Pods must be available. When you specify the value `maxUnavailable` as a percentage, Kubernetes rounds up the number of Pods that may be disrupted. Thereby a disruption can exceed your defined `maxUnavailable` percentage. You can examine the code that controls this behavior.

## Specifying a PodDisruptionBudget

A `PodDisruptionBudget` has three fields:

- A label selector `.spec.selector` to specify the set of pods to which it applies. This field is required.
- `.spec.minAvailable` which is a description of the number of pods from that set that must still be available after the eviction, even in the absence of the evicted pod. `minAvailable` can be either an absolute number or a percentage.
- `.spec.maxUnavailable` (available in Kubernetes 1.7 and higher) which is a description of the number of pods from that set that can be unavailable after the eviction. It can be either an absolute number or a percentage.

**Note:**

The behavior for an empty selector differs between the policy/v1beta1 and policy/v1 APIs for PodDisruptionBudgets. For policy/v1beta1 an empty selector matches zero pods, while for policy/v1 an empty selector matches every pod in the namespace.

You can specify only one of `maxUnavailable` and `minAvailable` in a single `PodDisruptionBudget`. `maxUnavailable` can only be used to control the eviction of pods that all have the same associated controller managing them. In the examples below, "desired replicas" is the `scale` of the controller managing the pods being selected by the `PodDisruptionBudget`.

Example 1: With a `minAvailable` of 5, evictions are allowed as long as they leave behind 5 or more healthy pods among those selected by the PodDisruptionBudget's `selector`.

Example 2: With a `minAvailable` of 30%, evictions are allowed as long as at least 30% of the number of desired replicas are healthy.

Example 3: With a `maxUnavailable` of 5, evictions are allowed as long as there are at most 5 unhealthy replicas among the total number of desired replicas.

Example 4: With a `maxUnavailable` of 30%, evictions are allowed as long as the number of unhealthy replicas does not exceed 30% of the total number of desired replica rounded up to the nearest integer. If the total number of desired replicas is just one, that single replica is still allowed for disruption, leading to an effective unavailability of 100%.

In typical usage, a single budget would be used for a collection of pods managed by a controller—for example, the pods in a single ReplicaSet or StatefulSet.

**Note:**

A disruption budget does not truly guarantee that the specified number/percentage of pods will always be up. For example, a node that hosts a pod from the collection may fail when the collection is at the minimum size specified in the budget, thus bringing the number of available pods from the collection below the specified size. The budget can only protect against voluntary evictions, not all causes of unavailability.

If you set `maxUnavailable` to 0% or 0, or you set `minAvailable` to 100% or the number of replicas, you are requiring zero voluntary evictions. When you set zero voluntary evictions for a workload object such as ReplicaSet, then you cannot successfully drain a Node running one of those Pods. If you try to drain a Node where an unevictable Pod is running, the drain never completes. This is permitted as per the semantics of `PodDisruptionBudget`.

You can find examples of pod disruption budgets defined below. They match pods with the label `app: zookeeper`.

Example PDB Using minAvailable:

`policy/zookeeper-pod-disruption-budget-minavailable.yaml` Copy policy/zookeeper-pod-disruption-budget-minavailable.yaml to clipboard

```
apiVersion: policy/v1
kind: PodDisruptionBudgetmetadata:  name: zk-pdbspec:  minAvailable: 2  selector:    matchLabels:      app: zookeeper
```

Example PDB Using maxUnavailable:

policy/zookeeper-pod-disruption-budget-maxunavailable.yaml Copy policy/zookeeper-pod-disruption-budget-maxunavailable.yaml to clipboard

```
apiVersion: policy/v1
kind: PodDisruptionBudgetmetadata:  name: zk-pdbspec:  maxUnavailable: 1  selector:    matchLabels:      app: zookeeper
```

For example, if the above `zk-pdb` object selects the pods of a StatefulSet of size 3, both specifications have the exact same meaning. The use of `maxUnavailable` is recommended as it automatically responds to changes in the number of replicas of the corresponding controller.

## Create the PDB object

You can create or update the PDB object using kubectl.

```
kubectl apply -f mypdb.yaml
```

## Check the status of the PDB

Use kubectl to check that your PDB is created.

Assuming you don't actually have pods matching `app: zookeeper` in your namespace, then you'll see something like this:

```
kubectl get poddisruptionbudgets
```

```
NAME     MIN AVAILABLE   MAX UNAVAILABLE   ALLOWED DISRUPTIONS   AGE
zk-pdb   2               N/A               0                     7s
```

If there are matching pods (say, 3), then you would see something like this:

```
kubectl get poddisruptionbudgets
```

```
NAME     MIN AVAILABLE   MAX UNAVAILABLE   ALLOWED DISRUPTIONS   AGE
zk-pdb   2               N/A               1                     7s
```

The non-zero value for `ALLOWED DISRUPTIONS` means that the disruption controller has seen the pods, counted the matching pods, and updated the status of the PDB.

You can get more information about the status of a PDB with this command:

```
kubectl get poddisruptionbudgets zk-pdb -o yaml
```

```
apiVersion: policy/v1
kind: PodDisruptionBudgetmetadata:  annotations:…  creationTimestamp: "2020-03-04T04:22:56Z"  generation: 1  name: zk-pdb…status:
```

### Healthiness of a Pod

The current implementation considers healthy pods, as pods that have `.status.conditions` item with `type="Ready"` and `status="True"`. These pods are tracked via `.status.currentHealthy` field in the PDB status.

## Unhealthy Pod Eviction Policy

FEATURE STATE: `Kubernetes v1.31 [stable]` (enabled by default: true)

PodDisruptionBudget guarding an application ensures that `.status.currentHealthy` number of pods does not fall below the number specified in `.status.desiredHealthy` by disallowing eviction of healthy pods. By using `.spec.unhealthyPodEvictionPolicy`, you can also define the criteria when unhealthy pods should be considered for eviction. The default behavior when no policy is specified corresponds to the `IfHealthyBudget` policy.

Policies:

`IfHealthyBudget`
: Running pods (`.status.phase="Running"`), but not yet healthy can be evicted only if the guarded application is not disrupted (`.status.currentHealthy` is at least equal to `.status.desiredHealthy`).

  This policy ensures that running pods of an already disrupted application have the best chance to become healthy. This has negative implications for draining nodes, which can be blocked by misbehaving applications that are guarded by a PDB. More specifically applications with pods in `CrashLoopBackOff` state (due to a bug or misconfiguration), or pods that are just failing to report the `Ready` condition.

`AlwaysAllow`
: Running pods (`.status.phase="Running"`), but not yet healthy are considered disrupted and can be evicted regardless of whether the criteria in a PDB is met.

  This means prospective running pods of a disrupted application might not get a chance to become healthy. By using this policy, cluster managers can easily evict misbehaving applications that are guarded by a PDB. More specifically applications with pods in `CrashLoopBackOff` state (due to a bug or misconfiguration), or pods that are just failing to report the `Ready` condition.

**Note:**

Pods in `Pending`, `Succeeded` or `Failed` phase are always considered for eviction.

## Arbitrary workloads and arbitrary selectors

You can skip this section if you only use PDBs with the built-in workload resources (Deployment, ReplicaSet, StatefulSet and ReplicationController) or with custom resources that implement a `scale` subresource, and where the PDB selector exactly matches the selector of the Pod's owning resource.

You can use a PDB with pods controlled by another resource, by an "operator", or bare pods, but with these restrictions:

- only `.spec.minAvailable` can be used, not `.spec.maxUnavailable`.
- only an integer value can be used with `.spec.minAvailable`, not a percentage.

It is not possible to use other availability configurations, because Kubernetes cannot derive a total number of pods without a supported owning resource.

You can use a selector which selects a subset or superset of the pods belonging to a workload resource. The eviction API will disallow eviction of any pod covered by multiple PDBs, so most users will want to avoid overlapping selectors. One reasonable use of overlapping PDBs is when pods are being transitioned from one PDB to another.

---

# HorizontalPodAutoscaler Walkthrough

A HorizontalPodAutoscaler (HPA for short) automatically updates a workload resource (such as a Deployment or StatefulSet), with the aim of automatically scaling the workload to match demand.

Horizontal scaling means that the response to increased load is to deploy more Pods. This is different from *vertical* scaling, which for Kubernetes would mean assigning more resources (for example: memory or CPU) to the Pods that are already running for the workload.

If the load decreases, and the number of Pods is above the configured minimum, the HorizontalPodAutoscaler instructs the workload resource (the Deployment, StatefulSet, or other similar resource) to scale back down.

This document walks you through an example of enabling HorizontalPodAutoscaler to automatically manage scale for an example web app. This example workload is Apache httpd running some PHP code.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Your Kubernetes server must be at or later than version 1.23.

To check the version, enter `kubectl version`.

If you're running an older release of Kubernetes, refer to the version of the documentation for that release (see available documentation versions).

To follow this walkthrough, you also need to use a cluster that has a Metrics Server deployed and configured. The Kubernetes Metrics Server collects resource metrics from the kubelets in your cluster, and exposes those metrics through the Kubernetes API, using an APIService to add new kinds of resource that represent metric readings.

To learn how to deploy the Metrics Server, see the metrics-server documentation.

If you are running Minikube, run the following command to enable metrics-server:

```
minikube addons enable metrics-server
```

## Run and expose php-apache server

To demonstrate a HorizontalPodAutoscaler, you will first start a Deployment that runs a container using the `hpa-example` image, and expose it as a Service using the following manifest:

`application/php-apache.yaml` Copy application/php-apache.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: php-apachespec:  selector:    matchLabels:      run: php-apache  template:    metadata:      labe
```

To do so, run the following command:

```
kubectl apply -f https://k8s.io/examples/application/php-apache.yaml
```

```
deployment.apps/php-apache created
service/php-apache created
```

## Create the HorizontalPodAutoscaler

Now that the server is running, create the autoscaler using `kubectl`. The `kubectl autoscale` subcommand, part of `kubectl`, helps you do this.

You will shortly run a command that creates a HorizontalPodAutoscaler that maintains between 1 and 10 replicas of the Pods controlled by the php-apache Deployment that you created in the first step of these instructions.

Roughly speaking, the HPA controller will increase and decrease the number of replicas (by updating the Deployment) to maintain an average CPU utilization across all Pods of 50%. The Deployment then updates the ReplicaSet - this is part of how all Deployments work in Kubernetes - and then the ReplicaSet either adds or removes Pods based on the change to its `.spec`.

Since each pod requests 200 milli-cores by `kubectl run`, this means an average CPU usage of 100 milli-cores. See Algorithm details for more details on the algorithm.

Create the HorizontalPodAutoscaler:

```
kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10
```

```
horizontalpodautoscaler.autoscaling/php-apache autoscaled
```

You can check the current status of the newly-made HorizontalPodAutoscaler, by running:

```
# You can use "hpa" or "horizontalpodautoscaler"; either name works OK.
kubectl get hpa
```

The output is similar to:

```
NAME         REFERENCE                     TARGET    MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache/scale   0% / 50%  1         10        1          18s
```

(if you see other HorizontalPodAutoscalers with different names, that means they already existed, and isn't usually a problem).

Please note that the current CPU consumption is 0% as there are no clients sending requests to the server (the `TARGET` column shows the average across all the Pods controlled by the corresponding deployment).

## Increase the load

Next, see how the autoscaler reacts to increased load. To do this, you'll start a different Pod to act as a client. The container within the client Pod runs in an infinite loop, sending queries to the php-apache service.

```
# Run this in a separate terminal
# so that the load generation continues and you can carry on with the rest of the steps
kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http
```

Now run:

```
# type Ctrl+C to end the watch when you're ready
kubectl get hpa php-apache --watch
```

Within a minute or so, you should see the higher CPU load; for example:

```
NAME         REFERENCE                     TARGET      MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache/scale   305% / 50%  1         10        1          3m
```

and then, more replicas. For example:

```
NAME         REFERENCE                     TARGET      MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache/scale   305% / 50%  1         10        7          3m
```

Here, CPU consumption has increased to 305% of the request. As a result, the Deployment was resized to 7 replicas:

```
kubectl get deployment php-apache
```

You should see the replica count matching the figure from the HorizontalPodAutoscaler

```
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
php-apache   7/7      7           7           19m
```

**Note:**

It may take a few minutes to stabilize the number of replicas. Since the amount of load is not controlled in any way it may happen that the final number of replicas will differ from this example.

## Stop generating load

To finish the example, stop sending the load.

In the terminal where you created the Pod that runs a `busybox` image, terminate the load generation by typing `<Ctrl> + C`.

Then verify the result state (after a minute or so):

```
# type Ctrl+C to end the watch when you're ready
kubectl get hpa php-apache --watch
```

The output is similar to:

```
NAME         REFERENCE                     TARGET       MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache/scale   0% / 50%     1         10        1          11m
```

and the Deployment also shows that it has scaled down:

```
kubectl get deployment php-apache
```

```
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
php-apache   1/1     1            1           27m
```

Once CPU utilization dropped to 0, the HPA automatically scaled the number of replicas back down to 1.

Autoscaling the replicas may take a few minutes.

## Autoscaling on multiple metrics and custom metrics

You can introduce additional metrics to use when autoscaling the `php-apache` Deployment by making use of the `autoscaling/v2` API version.

First, get the YAML of your HorizontalPodAutoscaler in the `autoscaling/v2` form:

```
kubectl get hpa php-apache -o yaml > /tmp/hpa-v2.yaml
```

Open the `/tmp/hpa-v2.yaml` file in an editor, and you should see YAML which looks like this:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscalermetadata:  name: php-apachespec:  scaleTargetRef:    apiVersion: apps/v1    kind: Deployment    name:
```

Notice that the `targetCPUUtilizationPercentage` field has been replaced with an array called `metrics`. The CPU utilization metric is a *resource metric*, since it is represented as a percentage of a resource specified on pod containers. Notice that you can specify other resource metrics besides CPU. By default, the only other supported resource metric is `memory`. These resources do not change names from cluster to cluster, and should always be available, as long as the `metrics.k8s.io` API is available.

You can also specify resource metrics in terms of direct values, instead of as percentages of the requested value, by using a `target.type` of `AverageValue` instead of `Utilization`, and setting the corresponding `target.averageValue` field instead of the `target.averageUtilization`.

```
  metrics:
  - type: Resource
    resource:
      name: memory
      target:
        type: AverageValue
        averageValue: 500Mi
```

There are two other types of metrics, both of which are considered *custom metrics*: pod metrics and object metrics. These metrics may have names which are cluster specific, and require a more advanced cluster monitoring setup.

The first of these alternative metric types is *pod metrics*. These metrics describe Pods, and are averaged together across Pods and compared with a target value to determine the replica count. They work much like resource metrics, except that they *only* support a `target` type of `AverageValue`.

Pod metrics are specified using a metric block like this:

```
type: Pods
pods:  metric:    name: packets-per-second  target:    type: AverageValue    averageValue: 1k
```

The second alternative metric type is *object metrics*. These metrics describe a different object in the same namespace, instead of describing Pods. The metrics are not necessarily fetched from the object; they only describe it. Object metrics support `target` types of both `Value` and `AverageValue`. With `Value`, the target is compared directly to the returned metric from the API. With `AverageValue`, the value returned from the custom metrics API is divided by the number of Pods before being compared to the target. The following example is the YAML representation of the `requests-per-second` metric.

```
type: Object
object:  metric:    name: requests-per-second  describedObject:    apiVersion: networking.k8s.io/v1    kind: Ingress    name: main
```

If you provide multiple such metric blocks, the HorizontalPodAutoscaler will consider each metric in turn. The HorizontalPodAutoscaler will calculate proposed replica counts for each metric, and then choose the one with the highest replica count.

For example, if you had your monitoring system collecting metrics about network traffic, you could update the definition above using `kubectl edit` to look like this:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscalermetadata:  name: php-apachespec:  scaleTargetRef:    apiVersion: apps/v1    kind: Deployment    name:
```

Then, your HorizontalPodAutoscaler would attempt to ensure that each pod was consuming roughly 50% of its requested CPU, serving 1000 packets per second, and that all pods behind the main-route Ingress were serving a total of 10000 requests per second.

### Autoscaling on more specific metrics

Many metrics pipelines allow you to describe metrics either by name or by a set of additional descriptors called *labels*. For all non-resource metric types (pod, object, and external, described below), you can specify an additional label selector which is passed to your metric pipeline. For instance, if you collect a metric `http_requests` with the `verb` label, you can specify the following metric block to scale only on GET requests:

```
type: Object
object:  metric:    name: http_requests    selector: {matchLabels: {verb: GET}}
```

This selector uses the same syntax as the full Kubernetes label selectors. The monitoring pipeline determines how to collapse multiple series into a single value, if the name and selector match multiple series. The selector is additive, and cannot select metrics that describe objects that are **not** the target object (the target pods in the case of the `Pods` type, and the described object in the case of the `Object` type).

### Autoscaling on metrics not related to Kubernetes objects

Applications running on Kubernetes may need to autoscale based on metrics that don't have an obvious relationship to any object in the Kubernetes cluster, such as metrics describing a hosted service with no direct correlation to Kubernetes namespaces. In Kubernetes 1.10 and later, you can address this use case with *external metrics*.

Using external metrics requires knowledge of your monitoring system; the setup is similar to that required when using custom metrics. External metrics allow you to autoscale your cluster based on any metric available in your monitoring system. Provide a `metric` block with a `name` and `selector`, as above, and use the `External` metric type instead of `Object`. If multiple time series are matched by the `metricSelector`, the sum of their values is used by the HorizontalPodAutoscaler. External metrics support both the `Value` and `AverageValue` target types, which function exactly the same as when you use the `Object` type.

For example if your application processes tasks from a hosted queue service, you could add the following section to your HorizontalPodAutoscaler manifest to specify that you need one worker per 30 outstanding tasks.

```
- type: External
  external:
    metric:
      name: queue_messages_ready
      selector:
        matchLabels:
          queue: "worker_tasks"
    target:
      type: AverageValue
      averageValue: 30
```

When possible, it's preferable to use the custom metric target types instead of external metrics, since it's easier for cluster administrators to secure the custom metrics API. The external metrics API potentially allows access to any metric, so cluster administrators should take care when exposing it.

## Appendix: Horizontal Pod Autoscaler Status Conditions

When using the `autoscaling/v2` form of the HorizontalPodAutoscaler, you will be able to see *status conditions* set by Kubernetes on the HorizontalPodAutoscaler. These status conditions indicate whether or not the HorizontalPodAutoscaler is able to scale, and whether or not it is currently restricted in any way.

The conditions appear in the `status.conditions` field. To see the conditions affecting a HorizontalPodAutoscaler, we can use `kubectl describe hpa`:

```
kubectl describe hpa cm-test
```

```
Name:                           cm-test
Namespace:                      prom
Labels:                         <none>
Annotations:                    <none>
CreationTimestamp:              Fri, 16 Jun 2017 18:09:22 +0000
Reference:                      ReplicationController/cm-test
Metrics:                        ( current / target )
  "http_requests" on pods:      66m / 500m
Min replicas:                   1
Max replicas:                   4
ReplicationController pods:     1 current / 1 desired
Conditions:
  Type                  Status  Reason                  Message
  ----                  ------  ------                  -------
  AbleToScale           True    ReadyForNewScale        the last scale time was sufficiently old as to warrant a new scale
  ScalingActive         True    ValidMetricFound        the HPA was able to successfully calculate a replica count from pods metri
  ScalingLimited        False   DesiredWithinRange      the desired replica count is within the acceptable range
Events:
```

For this HorizontalPodAutoscaler, you can see several conditions in a healthy state. The first, `AbleToScale`, indicates whether or not the HPA is able to fetch and update scales, as well as whether or not any backoff-related conditions would prevent scaling. The second, `ScalingActive`, indicates whether or not the HPA is enabled (i.e. the replica count of the target is not zero) and is able to calculate desired scales. When it is `False`, it generally indicates problems with fetching metrics. Finally, the last condition, `ScalingLimited`, indicates that the desired scale was capped by the maximum or minimum of the HorizontalPodAutoscaler. This is an indication that you may wish to raise or lower the minimum or maximum replica count constraints on your HorizontalPodAutoscaler.

## Quantities

All metrics in the HorizontalPodAutoscaler and metrics APIs are specified using a special whole-number notation known in Kubernetes as a quantity. For example, the quantity `10500m` would be written as `10.5` in decimal notation. The metrics APIs will return whole numbers without a suffix when possible, and will generally return quantities in milli-units otherwise. This means you might see your metric value fluctuate between `1` and `1500m`, or `1` and `1.5` when written in decimal notation.

## Other possible scenarios

### Creating the autoscaler declaratively

Instead of using `kubectl autoscale` command to create a HorizontalPodAutoscaler imperatively we can use the following manifest to create it declaratively:

`application/hpa/php-apache.yaml` Copy application/hpa/php-apache.yaml to clipboard

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscalermetadata:  name: php-apachespec:  scaleTargetRef:    apiVersion: apps/v1    kind: Deployment    name:
```

Then, create the autoscaler by executing the following command:

```
kubectl create -f https://k8s.io/examples/application/hpa/php-apache.yaml
```

```
horizontalpodautoscaler.autoscaling/php-apache created
```

# Run Applications

Run and manage both stateless and stateful applications.

**Run a Stateless Application Using a Deployment**

**Run a Single-Instance Stateful Application**

**Run a Replicated Stateful Application**

**Scale a StatefulSet**

**Delete a StatefulSet**

**Force Delete StatefulSet Pods**

**Horizontal Pod Autoscaling**

**HorizontalPodAutoscaler Walkthrough**

**Specifying a Disruption Budget for your Application**

**Accessing the Kubernetes API from a Pod**

# Run a Stateless Application Using a Deployment

This page shows how to run an application using a Kubernetes Deployment object.

## Objectives

- Create an nginx deployment.
- Use kubectl to list information about the deployment.
- Update the deployment.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Your Kubernetes server must be at or later than version v1.9.

To check the version, enter `kubectl version`.

## Creating and exploring an nginx deployment

You can run an application by creating a Kubernetes Deployment object, and you can describe a Deployment in a YAML file. For example, this YAML file describes a Deployment that runs the nginx:1.14.2 Docker image:

`application/deployment.yaml` Copy application/deployment.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: nginx-deploymentspec:  selector:    matchLabels:      app: nginx  replicas: 2 # tells deployment
```

1. Create a Deployment based on the YAML file:

   ```
   kubectl apply -f https://k8s.io/examples/application/deployment.yaml
   ```

2. Display information about the Deployment:

   ```
   kubectl describe deployment nginx-deployment
   ```

   The output is similar to this:

   ```
   Name:     nginx-deployment
   Namespace:    default
   CreationTimestamp:  Tue, 30 Aug 2016 18:11:37 -0700
   Labels:     app=nginx
   Annotations:    deployment.kubernetes.io/revision=1
   Selector:   app=nginx
   Replicas:   2 desired | 2 updated | 2 total | 2 available | 0 unavailable
   StrategyType:   RollingUpdate
   ```

```
MinReadySeconds:  0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
Pod Template:
  Labels:       app=nginx
  Containers:
    nginx:
    Image:              nginx:1.14.2
    Port:               80/TCP
    Environment:        <none>
    Mounts:             <none>
  Volumes:              <none>
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
OldReplicaSets:   <none>
NewReplicaSet:    nginx-deployment-1771418926 (2/2 replicas created)
No events.
```

3. List the Pods created by the deployment:

   ```
   kubectl get pods -l app=nginx
   ```

   The output is similar to this:

   ```
   NAME                               READY     STATUS    RESTARTS   AGE
   nginx-deployment-1771418926-7o5ns   1/1       Running   0          16h
   nginx-deployment-1771418926-r18az   1/1       Running   0          16h
   ```

4. Display information about a Pod:

   ```
   kubectl describe pod <pod-name>
   ```

   where `<pod-name>` is the name of one of your Pods.

## Updating the deployment

You can update the deployment by applying a new YAML file. This YAML file specifies that the deployment should be updated to use nginx 1.16.1.

[application/deployment-update.yaml](application/deployment-update.yaml) Copy application/deployment-update.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: nginx-deploymentspec:  selector:    matchLabels:      app: nginx  replicas: 2  template:    metad
```

1. Apply the new YAML file:

   ```
   kubectl apply -f https://k8s.io/examples/application/deployment-update.yaml
   ```

2. Watch the deployment create pods with new names and delete the old pods:

   ```
   kubectl get pods -l app=nginx
   ```

## Scaling the application by increasing the replica count

You can increase the number of Pods in your Deployment by applying a new YAML file. This YAML file sets `replicas` to 4, which specifies that the Deployment should have four Pods:

[application/deployment-scale.yaml](application/deployment-scale.yaml) Copy application/deployment-scale.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: nginx-deploymentspec:  selector:    matchLabels:      app: nginx  replicas: 4 # Update the replic
```

1. Apply the new YAML file:

   ```
   kubectl apply -f https://k8s.io/examples/application/deployment-scale.yaml
   ```

2. Verify that the Deployment has four Pods:

   ```
   kubectl get pods -l app=nginx
   ```

   The output is similar to this:

   ```
   NAME                               READY     STATUS    RESTARTS   AGE
   nginx-deployment-148880595-4zdqq   1/1       Running   0          25s
   nginx-deployment-148880595-6zgi1   1/1       Running   0          25s
   nginx-deployment-148880595-fxcez   1/1       Running   0          2m
   nginx-deployment-148880595-rwovn   1/1       Running   0          2m
   ```

## Deleting a deployment

Delete the deployment by name:

```
kubectl delete deployment nginx-deployment
```

## ReplicationControllers -- the Old Way

The preferred way to create a replicated application is to use a Deployment, which in turn uses a ReplicaSet. Before the Deployment and ReplicaSet were added to Kubernetes, replicated applications were configured using a ReplicationController.

## What's next

- Learn more about Deployment objects.

# Delete a StatefulSet

This task shows you how to delete a StatefulSet.

## Before you begin

- This task assumes you have an application running on your cluster represented by a StatefulSet.

## Deleting a StatefulSet

You can delete a StatefulSet in the same way you delete other resources in Kubernetes: use the `kubectl delete` command, and specify the StatefulSet either by file or by name.

```
kubectl delete -f <file.yaml>
```

```
kubectl delete statefulsets <statefulset-name>
```

You may need to delete the associated headless service separately after the StatefulSet itself is deleted.

```
kubectl delete service <service-name>
```

When deleting a StatefulSet through `kubectl`, the StatefulSet scales down to 0. All Pods that are part of this workload are also deleted. If you want to delete only the StatefulSet and not the Pods, use `--cascade=orphan`. For example:

```
kubectl delete -f <file.yaml> --cascade=orphan
```

By passing `--cascade=orphan` to `kubectl delete`, the Pods managed by the StatefulSet are left behind even after the StatefulSet object itself is deleted. If the pods have a label `app.kubernetes.io/name=MyApp`, you can then delete them as follows:

```
kubectl delete pods -l app.kubernetes.io/name=MyApp
```

### Persistent Volumes

Deleting the Pods in a StatefulSet will not delete the associated volumes. This is to ensure that you have the chance to copy data off the volume before deleting it. Deleting the PVC after the pods have terminated might trigger deletion of the backing Persistent Volumes depending on the storage class and reclaim policy. You should never assume ability to access a volume after claim deletion.

**Note:**

Use caution when deleting a PVC, as it may lead to data loss.

### Complete deletion of a StatefulSet

To delete everything in a StatefulSet, including the associated pods, you can run a series of commands similar to the following:

```
grace=$(kubectl get pods <stateful-set-pod> --template '{{.spec.terminationGracePeriodSeconds}}')
kubectl delete statefulset -l app.kubernetes.io/name=MyApp
sleep $grace
kubectl delete pvc -l app.kubernetes.io/name=MyApp
```

In the example above, the Pods have the label `app.kubernetes.io/name=MyApp`; substitute your own label as appropriate.

### Force deletion of StatefulSet pods

If you find that some pods in your StatefulSet are stuck in the 'Terminating' or 'Unknown' states for an extended period of time, you may need to manually intervene to forcefully delete the pods from the apiserver. This is a potentially dangerous task. Refer to Force Delete StatefulSet Pods for details.

## What's next

Learn more about force deleting StatefulSet Pods.

# Horizontal Pod Autoscaling

In Kubernetes, a *HorizontalPodAutoscaler* automatically updates a workload resource (such as a Deployment or StatefulSet), with the aim of automatically scaling the workload to match demand.

Horizontal scaling means that the response to increased load is to deploy more Pods. This is different from *vertical* scaling, which for Kubernetes would mean assigning more resources (for example: memory or CPU) to the Pods that are already running for the workload.

If the load decreases, and the number of Pods is above the configured minimum, the HorizontalPodAutoscaler instructs the workload resource (the Deployment, StatefulSet, or other similar resource) to scale back down.

Horizontal pod autoscaling does not apply to objects that can't be scaled (for example: a DaemonSet.)

The HorizontalPodAutoscaler is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. The horizontal pod autoscaling controller, running within the Kubernetes control plane, periodically adjusts the desired scale of its target (for example, a Deployment) to match observed metrics such as average CPU utilization, average memory utilization, or any other custom metric you specify.

There is walkthrough example of using horizontal pod autoscaling.

## How does a HorizontalPodAutoscaler work?

graph BT hpa[HorizontalPodAutoscaler] --> scale[Scale] subgraph rc[RC / Deployment] scale end scale -.-> pod1[Pod 1] scale -.-> pod2[Pod 2] scale -.-> pod3[Pod N] classDef hpa fill:#D5A6BD,stroke:#1E1E1D,stroke-width:1px,color:#1E1E1D; classDef rc fill:#F9CB9C,stroke:#1E1E1D,stroke-width:1px,color:#1E1E1D; classDef scale fill:#B6D7A8,stroke:#1E1E1D,stroke-width:1px,color:#1E1E1D; classDef pod fill:#9FC5E8,stroke:#1E1E1D,stroke-width:1px,color:#1E1E1D; class hpa hpa; class rc rc; class scale scale; class pod1,pod2,pod3 pod

Figure 1. HorizontalPodAutoscaler controls the scale of a Deployment and its ReplicaSet

Kubernetes implements horizontal pod autoscaling as a control loop that runs intermittently (it is not a continuous process). The interval is set by the `--horizontal-pod-autoscaler-sync-period` parameter to the `kube-controller-manager` (and the default interval is 15 seconds).

Once during each period, the controller manager queries the resource utilization against the metrics specified in each HorizontalPodAutoscaler definition. The controller manager finds the target resource defined by the `scaleTargetRef`, then selects the pods based on the target resource's `.spec.selector` labels, and obtains the metrics from either the resource metrics API (for per-pod resource metrics), or the custom metrics API (for all other metrics).

- For per-pod resource metrics (like CPU), the controller fetches the metrics from the resource metrics API for each Pod targeted by the HorizontalPodAutoscaler. Then, if a target utilization value is set, the controller calculates the utilization value as a percentage of the equivalent resource request on the containers in each Pod. If a target raw value is set, the raw metric values are used directly. The controller then takes the mean of the utilization or the raw value (depending on the type of target specified) across all targeted Pods, and produces a ratio used to scale the number of desired replicas.

  Please note that if some of the Pod's containers do not have the relevant resource request set, CPU utilization for the Pod will not be defined and the autoscaler will not take any action for that metric. See the algorithm details section below for more information about how the autoscaling algorithm works.

- For per-pod custom metrics, the controller functions similarly to per-pod resource metrics, except that it works with raw values, not utilization values.

- For object metrics and external metrics, a single metric is fetched, which describes the object in question. This metric is compared to the target value, to produce a ratio as above. In the `autoscaling/v2` API version, this value can optionally be divided by the number of Pods before the comparison is made.

The common use for HorizontalPodAutoscaler is to configure it to fetch metrics from aggregated APIs (`metrics.k8s.io`, `custom.metrics.k8s.io`, or `external.metrics.k8s.io`). The `metrics.k8s.io` API is usually provided by an add-on named Metrics Server, which needs to be launched separately. For more information about resource metrics, see Metrics Server.

Support for metrics APIs explains the stability guarantees and support status for these different APIs.

The HorizontalPodAutoscaler controller accesses corresponding workload resources that support scaling (such as Deployments and StatefulSet). These resources each have a subresource named `scale`, an interface that allows you to dynamically set the number of replicas and examine each of their current states. For general information about subresources in the Kubernetes API, see Kubernetes API Concepts.

### Algorithm details

From the most basic perspective, the HorizontalPodAutoscaler controller operates on the ratio between desired metric value and current metric value:

$$\begin{equation*} desiredReplicas = ceil\left\lceil currentReplicas \times \frac{currentMetricValue}{desiredMetricValue} \right\rceil \end{equation*}$$

For example, if the current metric value is `200m`, and the desired value is `100m`, the number of replicas will be doubled, since \( { 200.0 \div 100.0 } = 2.0 \). If the current value is instead `50m`, you'll halve the number of replicas, since \( { 50.0 \div 100.0 } = 0.5 \). The control plane skips any scaling action if the ratio is sufficiently close to 1.0 (within a configurable tolerance, 0.1 by default).

When a `targetAverageValue` or `targetAverageUtilization` is specified, the `currentMetricValue` is computed by taking the average of the given metric across all Pods in the HorizontalPodAutoscaler's scale target.

Before checking the tolerance and deciding on the final values, the control plane also considers whether any metrics are missing, and how many Pods are `Ready`. All Pods with a deletion timestamp set (objects with a deletion timestamp are in the process of being shut down / removed) are ignored, and all failed Pods are discarded.

If a particular Pod is missing metrics, it is set aside for later; Pods with missing metrics will be used to adjust the final scaling amount.

When scaling on CPU, if any pod has yet to become ready (it's still initializing, or possibly is unhealthy) *or* the most recent metric point for the pod was before it became ready, that pod is set aside as well.

Due to technical constraints, the HorizontalPodAutoscaler controller cannot exactly determine the first time a pod becomes ready when determining whether to set aside certain CPU metrics. Instead, it considers a Pod "not yet ready" if it's unready and transitioned to ready within a short, configurable window of time since it started. This value is configured with the `--horizontal-pod-autoscaler-initial-readiness-delay` command line option, and its default is 30 seconds. Once a pod has become ready, it considers any transition to ready to be the first if it occurred within a longer, configurable time since it started. This value is configured with the `--horizontal-pod-autoscaler-cpu-initialization-period` command line option, and its default is 5 minutes.

The \( currentMetricValue \over desiredMetricValue \) base scale ratio is then calculated, using the remaining pods not set aside or discarded from above.

If there were any missing metrics, the control plane recomputes the average more conservatively, assuming those pods were consuming 100% of the desired value in case of a scale down, and 0% in case of a scale up. This dampens the magnitude of any potential scale.

Furthermore, if any not-yet-ready pods were present, and the workload would have scaled up without factoring in missing metrics or not-yet-ready pods, the controller conservatively assumes that the not-yet-ready pods are consuming 0% of the desired metric, further dampening the magnitude of a scale up.

After factoring in the not-yet-ready pods and missing metrics, the controller recalculates the usage ratio. If the new ratio reverses the scale direction, or is within the tolerance, the controller doesn't take any scaling action. In other cases, the new ratio is used to decide any change to the number of Pods.

Note that the *original* value for the average utilization is reported back via the HorizontalPodAutoscaler status, without factoring in the not-yet-ready pods or missing metrics, even when the new usage ratio is used.

If multiple metrics are specified in a HorizontalPodAutoscaler, this calculation is done for each metric, and then the largest of the desired replica counts is chosen. If any of these metrics cannot be converted into a desired replica count (e.g. due to an error fetching the metrics from the metrics APIs) and a scale down is suggested by the metrics which can be fetched, scaling is skipped. This means that the HPA is still capable of scaling up if one or more metrics give a `desiredReplicas` greater than the current value.

Finally, right before HPA scales the target, the scale recommendation is recorded. The controller considers all recommendations within a configurable window choosing the highest recommendation from within that window. You can configure this value using the `--horizontal-pod-autoscaler-downscale-stabilization` command line option, which defaults to 5 minutes. This means that scaledowns will occur gradually, smoothing out the impact of rapidly fluctuating metric values.

## Pod readiness and autoscaling metrics

The HorizontalPodAutoscaler (HPA) controller includes two command line options that influence how CPU metrics are collected from Pods during startup:

1. `--horizontal-pod-autoscaler-cpu-initialization-period` (default: 5 minutes)

This defines the time window after a Pod starts during which its **CPU usage is ignored** unless: - The Pod is in a `Ready` state **and** - The metric sample was taken entirely during the period it was `Ready`.

This command line option helps **exclude misleading high CPU usage** from initializing Pods (for example: Java apps warming up) in HPA scaling decisions.

1. `--horizontal-pod-autoscaler-initial-readiness-delay` (default: 30 seconds)

This defines a short delay period after a Pod starts during which the HPA controller treats Pods that are currently `Unready` as still initializing, **even if they have previously transitioned to `Ready` briefly**.

It is designed to: - Avoid including Pods that rapidly fluctuate between `Ready` and `Unready` during startup. - Ensure stability in the initial readiness signal before HPA considers their metrics valid.

You can only set these command line options cluster-wide.

### Key behaviors for pod readiness

- If a Pod is `Ready` and remains `Ready`, it can be counted as contributing metrics even within the delay.
- If a Pod rapidly toggles between `Ready` and `Unready`, metrics are ignored until it's considered stably `Ready`.

### Good practice for pod readiness

- Configure a `startupProbe` that doesn't pass until the high CPU usage has passed, or
- Ensure your `readinessProbe` only reports `Ready` **after** the CPU spike subsides, using `initialDelaySeconds`.

And ideally also set `--horizontal-pod-autoscaler-cpu-initialization-period` to **cover the startup duration**.

## API object

The HorizontalPodAutoscaler is an API kind in the Kubernetes `autoscaling` API group. The current stable version can be found in the `autoscaling/v2` API version which includes support for scaling on memory and custom metrics. The new fields introduced in `autoscaling/v2` are preserved as annotations when working with `autoscaling/v1`.

When you create a HorizontalPodAutoscaler API object, make sure the name specified is a valid DNS subdomain name. More details about the API object can be found at HorizontalPodAutoscaler Object.

## Stability of workload scale

When managing the scale of a group of replicas using the HorizontalPodAutoscaler, it is possible that the number of replicas keeps fluctuating frequently due to the dynamic nature of the metrics evaluated. This is sometimes referred to as *thrashing*, or *flapping*. It's similar to the concept of *hysteresis* in cybernetics.

## Autoscaling during rolling update

Kubernetes lets you perform a rolling update on a Deployment. In that case, the Deployment manages the underlying ReplicaSets for you. When you configure autoscaling for a Deployment, you bind a HorizontalPodAutoscaler to a single Deployment. The HorizontalPodAutoscaler manages the `replicas` field of the Deployment. The deployment controller is responsible for setting the `replicas` of the underlying ReplicaSets so that they add up to a suitable number during the rollout and also afterwards.

If you perform a rolling update of a StatefulSet that has an autoscaled number of replicas, the StatefulSet directly manages its set of Pods (there is no intermediate resource similar to ReplicaSet).

## Support for resource metrics

Any HPA target can be scaled based on the resource usage of the pods in the scaling target. When defining the pod specification the resource requests like `cpu` and `memory` should be specified. This is used to determine the resource utilization and used by the HPA controller to scale the target up or down. To use resource utilization based scaling specify a metric source like this:

```
type: Resource
resource:  name: cpu  target:    type: Utilization    averageUtilization: 60
```

With this metric the HPA controller will keep the average utilization of the pods in the scaling target at 60%. Utilization is the ratio between the current usage of resource to the requested resources of the pod. See Algorithm for more details about how the utilization is calculated and averaged.

**Note:**

Since the resource usages of all the containers are summed up the total pod utilization may not accurately represent the individual container resource usage. This could lead to situations where a single container might be running with high usage and the HPA will not scale out because the overall pod usage is still within acceptable limits.

### Container resource metrics

FEATURE STATE: `Kubernetes v1.30 [stable]` (enabled by default: true)

The HorizontalPodAutoscaler API also supports a container metric source where the HPA can track the resource usage of individual containers across a set of Pods, in order to scale the target resource. This lets you configure scaling thresholds for the containers that matter most in a particular Pod. For example, if you have a web application and a sidecar container that provides logging, you can scale based on the resource use of the web application, ignoring the sidecar container and its resource use.

If you revise the target resource to have a new Pod specification with a different set of containers, you should revise the HPA spec if that newly added container should also be used for scaling. If the specified container in the metric source is not present or only present in a subset of the pods then those pods are ignored and the recommendation is recalculated. See Algorithm for more details about the calculation. To use container resources for autoscaling define a metric source as follows:

```
type: ContainerResource
containerResource:  name: cpu  container: application  target:    type: Utilization    averageUtilization: 60
```

In the above example the HPA controller scales the target such that the average utilization of the cpu in the `application` container of all the pods is 60%.

**Note:**

If you change the name of a container that a HorizontalPodAutoscaler is tracking, you can make that change in a specific order to ensure scaling remains available and effective whilst the change is being applied. Before you update the resource that defines the container (such as a Deployment), you should update the associated HPA to track both the new and old container names. This way, the HPA is able to calculate a scaling recommendation throughout the update process.

Once you have rolled out the container name change to the workload resource, tidy up by removing the old container name from the HPA specification.

## Scaling on custom metrics

FEATURE STATE: `Kubernetes v1.23 [stable]`

(the `autoscaling/v2beta2` API version previously provided this ability as a beta feature)

Provided that you use the `autoscaling/v2` API version, you can configure a HorizontalPodAutoscaler to scale based on a custom metric (that is not built in to Kubernetes or any Kubernetes component). The HorizontalPodAutoscaler controller then queries for these custom metrics from the Kubernetes API.

See Support for metrics APIs for the requirements.

## Scaling on multiple metrics

FEATURE STATE: `Kubernetes v1.23 [stable]`

(the `autoscaling/v2beta2` API version previously provided this ability as a beta feature)

Provided that you use the `autoscaling/v2` API version, you can specify multiple metrics for a HorizontalPodAutoscaler to scale on. Then, the HorizontalPodAutoscaler controller evaluates each metric, and proposes a new scale based on that metric. The HorizontalPodAutoscaler takes the maximum scale recommended for each metric and sets the workload to that size (provided that this isn't larger than the overall maximum that you configured).

## Support for metrics APIs

By default, the HorizontalPodAutoscaler controller retrieves metrics from a series of APIs. In order for it to access these APIs, cluster administrators must ensure that:

- The API aggregation layer is enabled.
- The corresponding APIs are registered:

- For resource metrics, this is the `metrics.k8s.io` API, generally provided by metrics-server. It can be launched as a cluster add-on.
- For custom metrics, this is the `custom.metrics.k8s.io` API. It's provided by "adapter" API servers provided by metrics solution vendors. Check with your metrics pipeline to see if there is a Kubernetes metrics adapter available.
- For external metrics, this is the `external.metrics.k8s.io` API. It may be provided by the custom metrics adapters provided above.

For more information on these different metrics paths and how they differ please see the relevant design proposals for the HPA V2, custom.metrics.k8s.io and external.metrics.k8s.io.

For examples of how to use them see the walkthrough for using custom metrics and the walkthrough for using external metrics.

## Configurable scaling behavior

FEATURE STATE: `Kubernetes v1.23 [stable]`

(the `autoscaling/v2beta2` API version previously provided this ability as a beta feature)

If you use the `v2` HorizontalPodAutoscaler API, you can use the `behavior` field (see the API reference) to configure separate scale-up and scale-down behaviors. You specify these behaviors by setting `scaleUp` and / or `scaleDown` under the `behavior` field.

Scaling policies let you control the rate of change of replicas while scaling. Also two settings can be used to prevent flapping: you can specify a *stabilization window* for smoothing replica counts, and a tolerance to ignore minor metric fluctuations below a specified threshold.

### Scaling policies

One or more scaling policies can be specified in the `behavior` section of the spec. When multiple policies are specified the policy which allows the highest amount of change is the policy which is selected by default. The following example shows this behavior while scaling down:

```yaml
behavior:
  scaleDown:
    policies:
    - type: Pods
      value: 4
      periodSeconds: 60
    - type: Percent
      value: 10
      periodSeconds: 60
```

`periodSeconds` indicates the length of time in the past for which the policy must hold true. The maximum value that you can set for `periodSeconds` is 1800 (half an hour). The first policy *(Pods)* allows at most 4 replicas to be scaled down in one minute. The second policy *(Percent)* allows at most 10% of the current replicas to be scaled down in one minute.

Since by default the policy which allows the highest amount of change is selected, the second policy will only be used when the number of pod replicas is more than 40. With 40 or less replicas, the first policy will be applied. For instance if there are 80 replicas and the target has to be scaled down to 10 replicas then during the first step 8 replicas will be reduced. In the next iteration when the number of replicas is 72, 10% of the pods is 7.2 but the number is rounded up to 8. On each loop of the autoscaler controller the number of pods to be change is re-calculated based on the number of current replicas. When the number of replicas falls below 40 the first policy *(Pods)* is applied and 4 replicas will be reduced at a time.

The policy selection can be changed by specifying the `selectPolicy` field for a scaling direction. By setting the value to `Min` which would select the policy which allows the smallest change in the replica count. Setting the value to `Disabled` completely disables scaling in that direction.

### Stabilization window

The stabilization window is used to restrict the flapping of replica count when the metrics used for scaling keep fluctuating. The autoscaling algorithm uses this window to infer a previous desired state and avoid unwanted changes to workload scale.

For example, in the following example snippet, a stabilization window is specified for `scaleDown`.

```yaml
behavior:
  scaleDown:
    stabilizationWindowSeconds: 300
```

When the metrics indicate that the target should be scaled down the algorithm looks into previously computed desired states, and uses the highest value from the specified interval. In the above example, all desired states from the past 5 minutes will be considered.

This approximates a rolling maximum, and avoids having the scaling algorithm frequently remove Pods only to trigger recreating an equivalent Pod just moments later.

### Tolerance

FEATURE STATE: `Kubernetes v1.33 [alpha]` (enabled by default: false)

The `tolerance` field configures a threshold for metric variations, preventing the autoscaler from scaling for changes below that value.

This tolerance is defined as the amount of variation around the desired metric value under which no scaling will occur. For example, consider a HorizontalPodAutoscaler configured with a target memory consumption of 100MiB and a scale-up tolerance of 5%:

```yaml
behavior:
  scaleUp:
    tolerance: 0.05 # 5% tolerance for scale up
```

With this configuration, the HPA algorithm will only consider scaling up if the memory consumption is higher than 105MiB (that is: 5% above the target).

If you don't set this field, the HPA applies the default cluster-wide tolerance of 10%. This default can be updated for both scale-up and scale-down using the kube-controller-manager `--horizontal-pod-autoscaler-tolerance` command line argument. (You can't use the Kubernetes API to configure this default value.)

### Default behavior

To use the custom scaling not all fields have to be specified. Only values which need to be customized can be specified. These custom values are merged with default values. The default values match the existing behavior in the HPA algorithm.

```
behavior:
  scaleDown:
    stabilizationWindowSeconds: 300
    policies:
    - type: Percent
      value: 100
      periodSeconds: 15
  scaleUp:
    stabilizationWindowSeconds: 0
    policies:
    - type: Percent
      value: 100
      periodSeconds: 15
    - type: Pods
      value: 4
      periodSeconds: 15
    selectPolicy: Max
```

For scaling down the stabilization window is *300* seconds (or the value of the `--horizontal-pod-autoscaler-downscale-stabilization` command line option, if provided). There is only a single policy for scaling down which allows a 100% of the currently running replicas to be removed which means the scaling target can be scaled down to the minimum allowed replicas. For scaling up there is no stabilization window. When the metrics indicate that the target should be scaled up the target is scaled up immediately. There are 2 policies where 4 pods or a 100% of the currently running replicas may at most be added every 15 seconds till the HPA reaches its steady state.

### Example: change downscale stabilization window

To provide a custom downscale stabilization window of 1 minute, the following behavior would be added to the HPA:

```
behavior:
  scaleDown:
    stabilizationWindowSeconds: 60
```

### Example: limit scale down rate

To limit the rate at which pods are removed by the HPA to 10% per minute, the following behavior would be added to the HPA:

```
behavior:
  scaleDown:
    policies:
    - type: Percent
      value: 10
      periodSeconds: 60
```

To ensure that no more than 5 Pods are removed per minute, you can add a second scale-down policy with a fixed size of 5, and set `selectPolicy` to minimum. Setting `selectPolicy` to `Min` means that the autoscaler chooses the policy that affects the smallest number of Pods:

```
behavior:
  scaleDown:
    policies:
    - type: Percent
      value: 10
      periodSeconds: 60
    - type: Pods
      value: 5
      periodSeconds: 60
    selectPolicy: Min
```

### Example: disable scale down

The `selectPolicy` value of `Disabled` turns off scaling the given direction. So to prevent downscaling the following policy would be used:

```
behavior:
  scaleDown:
    selectPolicy: Disabled
```

## Support for HorizontalPodAutoscaler in kubectl

HorizontalPodAutoscaler, like every API resource, is supported in a standard way by `kubectl`. You can create a new autoscaler using `kubectl create` command. You can list autoscalers by `kubectl get hpa` or get detailed description by `kubectl describe hpa`. Finally, you can delete an autoscaler using `kubectl delete hpa`.

In addition, there is a special `kubectl autoscale` command for creating a HorizontalPodAutoscaler object. For instance, executing `kubectl autoscale rs foo --min=2 --max=5 --cpu-percent=80` will create an autoscaler for ReplicaSet *foo*, with target CPU utilization set to `80%` and the number of replicas between 2 and 5.

## Implicit maintenance-mode deactivation

You can implicitly deactivate the HPA for a target without the need to change the HPA configuration itself. If the target's desired replica count is set to 0, and the HPA's minimum replica count is greater than 0, the HPA stops adjusting the target (and sets the `ScalingActive` Condition on itself to `false`) until you reactivate it by manually adjusting the target's desired replica count or HPA's minimum replica count.

### Migrating Deployments and StatefulSets to horizontal autoscaling

When an HPA is enabled, it is recommended that the value of `spec.replicas` of the Deployment and / or StatefulSet be removed from their manifest(s). If this isn't done, any time a change to that object is applied, for example via `kubectl apply -f deployment.yaml`, this will instruct Kubernetes to scale the current number of Pods to the value of the `spec.replicas` key. This may not be desired and could be troublesome when an HPA is active, resulting in thrashing or flapping behavior.

Keep in mind that the removal of `spec.replicas` may incur a one-time degradation of Pod counts as the default value of this key is 1 (reference Deployment Replicas). Upon the update, all Pods except 1 will begin their termination procedures. Any deployment application afterwards will behave as normal and respect a rolling update configuration as desired. You can avoid this degradation by choosing one of the following two methods based on how you are modifying your deployments:

- Client Side Apply (this is the default)
- Server Side Apply

1. `kubectl apply edit-last-applied deployment/<deployment_name>`
2. In the editor, remove `spec.replicas`. When you save and exit the editor, `kubectl` applies the update. No changes to Pod counts happen at this step.
3. You can now remove `spec.replicas` from the manifest. If you use source code management, also commit your changes or take whatever other steps for revising the source code are appropriate for how you track updates.
4. From here on out you can run `kubectl apply -f deployment.yaml`

When using the Server-Side Apply you can follow the transferring ownership guidelines, which cover this exact use case.

## What's next

If you configure autoscaling in your cluster, you may also want to consider using node autoscaling to ensure you are running the right number of nodes.

For more information on HorizontalPodAutoscaler:

- Read a walkthrough example for horizontal pod autoscaling.
- Read documentation for `kubectl autoscale`.
- If you would like to write your own custom metrics adapter, check out the boilerplate to get started.
- Read the API reference for HorizontalPodAutoscaler.

---

# Force Delete StatefulSet Pods

This page shows how to delete Pods which are part of a stateful set, and explains the considerations to keep in mind when doing so.

## Before you begin

- This is a fairly advanced task and has the potential to violate some of the properties inherent to StatefulSet.
- Before proceeding, make yourself familiar with the considerations enumerated below.

## StatefulSet considerations

In normal operation of a StatefulSet, there is **never** a need to force delete a StatefulSet Pod. The StatefulSet controller is responsible for creating, scaling and deleting members of the StatefulSet. It tries to ensure that the specified number of Pods from ordinal 0 through N-1 are alive and ready. StatefulSet ensures that, at any time, there is at most one Pod with a given identity running in a cluster. This is referred to as *at most one* semantics provided by a StatefulSet.

Manual force deletion should be undertaken with caution, as it has the potential to violate the at most one semantics inherent to StatefulSet. StatefulSets may be used to run distributed and clustered applications which have a need for a stable network identity and stable storage. These applications often have configuration which relies on an ensemble of a fixed number of members with fixed identities. Having multiple members with the same identity can be disastrous and may lead to data loss (e.g. split brain scenario in quorum-based systems).

## Delete Pods

You can perform a graceful pod deletion with the following command:

```
kubectl delete pods <pod>
```

For the above to lead to graceful termination, the Pod **must not** specify a `pod.Spec.TerminationGracePeriodSeconds` of 0. The practice of setting a `pod.Spec.TerminationGracePeriodSeconds` of 0 seconds is unsafe and strongly discouraged for StatefulSet Pods. Graceful deletion is safe and will ensure that the Pod shuts down gracefully before the kubelet deletes the name from the apiserver.

A Pod is not deleted automatically when a node is unreachable. The Pods running on an unreachable Node enter the 'Terminating' or 'Unknown' state after a timeout. Pods may also enter these states when the user attempts graceful deletion of a Pod on an unreachable Node. The only ways in which a Pod in such a state can be removed from the apiserver are as follows:

- The Node object is deleted (either by you, or by the Node Controller).
- The kubelet on the unresponsive Node starts responding, kills the Pod and removes the entry from the apiserver.
- Force deletion of the Pod by the user.

The recommended best practice is to use the first or second approach. If a Node is confirmed to be dead (e.g. permanently disconnected from the network, powered down, etc), then delete the Node object. If the Node is suffering from a network partition, then try to resolve this or wait for it to resolve. When the partition heals, the kubelet will complete the deletion of the Pod and free up its name in the apiserver.

Normally, the system completes the deletion once the Pod is no longer running on a Node, or the Node is deleted by an administrator. You may override this by force deleting the Pod.

### Force Deletion

Force deletions **do not** wait for confirmation from the kubelet that the Pod has been terminated. Irrespective of whether a force deletion is successful in killing a Pod, it will immediately free up the name from the apiserver. This would let the StatefulSet controller create a replacement Pod with that same identity; this can lead to the duplication of a still-running Pod, and if said Pod can still communicate with the other members of the StatefulSet, will violate the at most one semantics that StatefulSet is designed to guarantee.

When you force delete a StatefulSet pod, you are asserting that the Pod in question will never again make contact with other Pods in the StatefulSet and its name can be safely freed up for a replacement to be created.

If you want to delete a Pod forcibly using kubectl version >= 1.5, do the following:

```
kubectl delete pods <pod> --grace-period=0 --force
```

If you're using any version of kubectl <= 1.4, you should omit the `--force` option and use:

```
kubectl delete pods <pod> --grace-period=0
```

If even after these commands the pod is stuck on `Unknown` state, use the following command to remove the pod from the cluster:

```
kubectl patch pod <pod> -p '{"metadata":{"finalizers":null}}'
```

Always perform force deletion of StatefulSet Pods carefully and with complete knowledge of the risks involved.

## What's next

Learn more about debugging a StatefulSet.

---

# Accessing the Kubernetes API from a Pod

This guide demonstrates how to access the Kubernetes API from within a pod.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

## Accessing the API from within a Pod

When accessing the API from within a Pod, locating and authenticating to the API server are slightly different to the external client case.

The easiest way to use the Kubernetes API from a Pod is to use one of the official client libraries. These libraries can automatically discover the API server and authenticate.

### Using Official Client Libraries

From within a Pod, the recommended ways to connect to the Kubernetes API are:

- For a Go client, use the official Go client library. The `rest.InClusterConfig()` function handles API host discovery and authentication automatically. See an example here.

- For a Python client, use the official Python client library. The `config.load_incluster_config()` function handles API host discovery and authentication automatically. See an example here.

- There are a number of other libraries available, please refer to the Client Libraries page.

In each case, the service account credentials of the Pod are used to communicate securely with the API server.

### Directly accessing the REST API

While running in a Pod, your container can create an HTTPS URL for the Kubernetes API server by fetching the `KUBERNETES_SERVICE_HOST` and `KUBERNETES_SERVICE_PORT_HTTPS` environment variables. The API server's in-cluster address is also published to a Service named `kubernetes` in the `default` namespace so that pods may reference `kubernetes.default.svc` as a DNS name for the local API server.

**Note:**

Kubernetes does not guarantee that the API server has a valid certificate for the hostname `kubernetes.default.svc`; however, the control plane **is** expected to present a valid certificate for the hostname or IP address that `$KUBERNETES_SERVICE_HOST` represents.

The recommended way to authenticate to the API server is with a service account credential. By default, a Pod is associated with a service account, and a credential (token) for that service account is placed into the filesystem tree of each container in that Pod, at `/var/run/secrets/kubernetes.io/serviceaccount/token`.

If available, a certificate bundle is placed into the filesystem tree of each container at `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt`, and should be used to verify the serving certificate of the API server.

Finally, the default namespace to be used for namespaced API operations is placed in a file at `/var/run/secrets/kubernetes.io/serviceaccount/namespace` in each container.

### Using kubectl proxy

If you would like to query the API without an official client library, you can run `kubectl proxy` as the command of a new sidecar container in the Pod. This way, `kubectl proxy` will authenticate to the API and expose it on the `localhost` interface of the Pod, so that other containers in the Pod can use it directly.

### Without using a proxy

It is possible to avoid using the kubectl proxy by passing the authentication token directly to the API server. The internal certificate secures the connection.

```
# Point to the internal API server hostname
APISERVER=https://kubernetes.default.svc

# Path to ServiceAccount token
SERVICEACCOUNT=/var/run/secrets/kubernetes.io/serviceaccount

# Read this Pod's namespace
NAMESPACE=$(cat ${SERVICEACCOUNT}/namespace)

# Read the ServiceAccount bearer token
TOKEN=$(cat ${SERVICEACCOUNT}/token)

# Reference the internal certificate authority (CA)
CACERT=${SERVICEACCOUNT}/ca.crt

# Explore the API with TOKEN
curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" -X GET ${APISERVER}/api
```

The output will be similar to this:

```
{
  "kind": "APIVersions",
  "versions": ["v1"],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.1.149:443"
    }
  ]
}
```