# Init Containers

This page provides an overview of init containers: specialized containers that run before app containers in a Pod. Init containers can contain utilities or setup scripts not present in an app image.

You can specify init containers in the Pod specification alongside the `containers` array (which describes app containers).

In Kubernetes, a sidecar container is a container that starts before the main application container and *continues to run*. This document is about init containers: containers that run to completion during Pod initialization.

## Understanding init containers

A Pod can have multiple containers running apps within it, but it can also have one or more init containers, which are run before the app containers are started.

Init containers are exactly like regular containers, except:

- Init containers always run to completion.
- Each init container must complete successfully before the next one starts.

If a Pod's init container fails, the kubelet repeatedly restarts that init container until it succeeds. However, if the Pod has a `restartPolicy` of Never, and an init container fails during startup of that Pod, Kubernetes treats the overall Pod as failed.

To specify an init container for a Pod, add the `initContainers` field into the Pod specification, as an array of `container` items (similar to the app `containers` field and its contents). See Container in the API reference for more details.

The status of the init containers is returned in `.status.initContainerStatuses` field as an array of the container statuses (similar to the `.status.containerStatuses` field).

### Differences from regular containers

Init containers support all the fields and features of app containers, including resource limits, volumes, and security settings. However, the resource requests and limits for an init container are handled differently, as documented in Resource sharing within containers.

Regular init containers (in other words: excluding sidecar containers) do not support the `lifecycle`, `livenessProbe`, `readinessProbe`, or `startupProbe` fields. Init containers must run to completion before the Pod can be ready; sidecar containers continue running during a Pod's lifetime, and *do* support some probes. See sidecar container for further details about sidecar containers.

If you specify multiple init containers for a Pod, kubelet runs each init container sequentially. Each init container must succeed before the next can run. When all of the init containers have run to completion, kubelet initializes the application containers for the Pod and runs them as usual.

### Differences from sidecar containers

Init containers run and complete their tasks before the main application container starts. Unlike sidecar containers, init containers are not continuously running alongside the main containers.

Init containers run to completion sequentially, and the main container does not start until all the init containers have successfully completed.

init containers do not support `lifecycle`, `livenessProbe`, `readinessProbe`, or `startupProbe` whereas sidecar containers support all these probes to control their lifecycle.

Init containers share the same resources (CPU, memory, network) with the main application containers but do not interact directly with them. They can, however, use shared volumes for data exchange.

## Using init containers

Because init containers have separate images from app containers, they have some advantages for start-up related code:

- Init containers can contain utilities or custom code for setup that are not present in an app image. For example, there is no need to make an image `FROM` another image just to use a tool like `sed`, `awk`, `python`, or `dig` during setup.
- The application image builder and deployer roles can work independently without the need to jointly build a single app image.
- Init containers can run with a different view of the filesystem than app containers in the same Pod. Consequently, they can be given access to Secrets that app containers cannot access.
- Because init containers run to completion before any app containers start, init containers offer a mechanism to block or delay app container startup until a set of preconditions are met. Once preconditions are met, all of the app containers in a Pod can start in parallel.
- Init containers can securely run utilities or custom code that would otherwise make an app container image less secure. By keeping unnecessary tools separate you can limit the attack surface of your app container image.

### Examples

Here are some ideas for how to use init containers:

- Wait for a Service to be created, using a shell one-line command like:

  ```
  for i in {1..100}; do sleep 1; if nslookup myservice; then exit 0; fi; done; exit 1
  ```

- Register this Pod with a remote server from the downward API with a command like:

```
curl -X POST http://$MANAGEMENT_SERVICE_HOST:$MANAGEMENT_SERVICE_PORT/register -d 'instance=$(<POD_NAME>)&ip=$(<POD_IP>)'
```

- Wait for some time before starting the app container with a command like

  ```
  sleep 60
  ```

- Clone a Git repository into a Volume
- Place values into a configuration file and run a template tool to dynamically generate a configuration file for the main app container. For example, place the `POD_IP` value in a configuration and generate the main app configuration file using Jinja.

### Init containers in use

This example defines a simple Pod that has two init containers. The first waits for `myservice`, and the second waits for `mydb`. Once both init containers complete, the Pod runs the app container from its `spec` section.

```
apiVersion: v1
kind: Podmetadata:  name: myapp-pod  labels:    app.kubernetes.io/name: MyAppspec:  containers:  - name: myapp-container    image:
```

You can start this Pod by running:

```
kubectl apply -f myapp.yaml
```

The output is similar to this:

```
pod/myapp-pod created
```

And check on its status with:

```
kubectl get -f myapp.yaml
```

The output is similar to this:

```
NAME        READY     STATUS     RESTARTS   AGE
myapp-pod   0/1       Init:0/2   0          6m
```

or for more details:

```
kubectl describe -f myapp.yaml
```

The output is similar to this:

```
Name:          myapp-pod
Namespace:     default
[...]
Labels:        app.kubernetes.io/name=MyApp
Status:        Pending
[...]
Init Containers:
  init-myservice:
[...]
    State:         Running
[...]
  init-mydb:
[...]
    State:         Waiting
      Reason:      PodInitializing
    Ready:         False
[...]
Containers:
  myapp-container:
[...]
    State:         Waiting
      Reason:      PodInitializing
    Ready:         False
[...]
Events:
  FirstSeen    LastSeen    Count    From                      SubObjectPath                           Type          Reason
  ---------    --------    -----    ----                      -------------                           --------      ------
  16s          16s         1        {default-scheduler }                                              Normal        Scheduled
  16s          16s         1        {kubelet 172.17.4.201}    spec.initContainers{init-myservice}     Normal        Pulling
  13s          13s         1        {kubelet 172.17.4.201}    spec.initContainers{init-myservice}     Normal        Pulled
  13s          13s         1        {kubelet 172.17.4.201}    spec.initContainers{init-myservice}     Normal        Created
  13s          13s         1        {kubelet 172.17.4.201}    spec.initContainers{init-myservice}     Normal        Started
```

To see logs for the init containers in this Pod, run:

```
kubectl logs myapp-pod -c init-myservice # Inspect the first init container
kubectl logs myapp-pod -c init-mydb      # Inspect the second init container
```

At this point, those init containers will be waiting to discover Services named `mydb` and `myservice`.

Here's a configuration you can use to make those Services appear:

```
---
apiVersion: v1kind: Servicemetadata:  name: myservicespec:  ports:  - protocol: TCP    port: 80    targetPort: 9376---apiVersion:
```

To create the `mydb` and `myservice` services:

```
kubectl apply -f services.yaml
```

The output is similar to this:

```
service/myservice created
service/mydb created
```

You'll then see that those init containers complete, and that the `myapp-pod` Pod moves into the Running state:

```
kubectl get -f myapp.yaml
```

The output is similar to this:

```
NAME        READY     STATUS    RESTARTS   AGE
myapp-pod   1/1       Running   0          9m
```

This simple example should provide some inspiration for you to create your own init containers. What's next contains a link to a more detailed example.

## Detailed behavior

During Pod startup, the kubelet delays running init containers until the networking and storage are ready. Then the kubelet runs the Pod's init containers in the order they appear in the Pod's spec.

Each init container must exit successfully before the next container starts. If a container fails to start due to the runtime or exits with failure, it is retried according to the Pod `restartPolicy`. However, if the Pod `restartPolicy` is set to Always, the init containers use `restartPolicy` OnFailure.

A Pod cannot be `Ready` until all init containers have succeeded. The ports on an init container are not aggregated under a Service. A Pod that is initializing is in the `Pending` state but should have a condition `Initialized` set to false.

If the Pod restarts, or is restarted, all init containers must execute again.

Changes to the init container spec are limited to the container image field. Directly altering the `image` field of an init container does *not* restart the Pod or trigger its recreation. If the Pod has yet to start, that change may have an effect on how the Pod boots up.

For a pod template you can typically change any field for an init container; the impact of making that change depends on where the pod template is used.

Because init containers can be restarted, retried, or re-executed, init container code should be idempotent. In particular, code that writes into any `emptyDir` volume should be prepared for the possibility that an output file already exists.

Init containers have all of the fields of an app container. However, Kubernetes prohibits `readinessProbe` from being used because init containers cannot define readiness distinct from completion. This is enforced during validation.

Use `activeDeadlineSeconds` on the Pod to prevent init containers from failing forever. The active deadline includes init containers. However it is recommended to use `activeDeadlineSeconds` only if teams deploy their application as a Job, because `activeDeadlineSeconds` has an effect even after initContainer finished. The Pod which is already running correctly would be killed by `activeDeadlineSeconds` if you set.

The name of each app and init container in a Pod must be unique; a validation error is thrown for any container sharing a name with another.

### Resource sharing within containers

Given the order of execution for init, sidecar and app containers, the following rules for resource usage apply:

- The highest of any particular resource request or limit defined on all init containers is the *effective init request/limit*. If any resource has no resource limit specified this is considered as the highest limit.
- The Pod's *effective request/limit* for a resource is the higher of:
  - the sum of all app containers request/limit for a resource
  - the effective init request/limit for a resource
- Scheduling is done based on effective requests/limits, which means init containers can reserve resources for initialization that are not used during the life of the Pod.
- The QoS (quality of service) tier of the Pod's *effective QoS tier* is the QoS tier for init containers and app containers alike.

Quota and limits are applied based on the effective Pod request and limit.

### Init containers and Linux cgroups

On Linux, resource allocations for Pod level control groups (cgroups) are based on the effective Pod request and limit, the same as the scheduler.

### Pod restart reasons

A Pod can restart, causing re-execution of init containers, for the following reasons:

- The Pod infrastructure container is restarted. This is uncommon and would have to be done by someone with root access to nodes.
- All containers in a Pod are terminated while `restartPolicy` is set to Always, forcing a restart, and the init container completion record has been lost due to garbage collection.

The Pod will not be restarted when the init container image is changed, or the init container completion record has been lost due to garbage collection. This applies for Kubernetes v1.20 and later. If you are using an earlier version of Kubernetes, consult the documentation for the version you are using.

## What's next

Learn more about the following:

- Creating a Pod that has an init container.

- Debug init containers.
- Overview of kubelet and kubectl.
- Types of probes: liveness, readiness, startup probe.
- Sidecar containers.

# Deployments

A Deployment manages a set of Pods to run an application workload, usually one that doesn't maintain state.

A *Deployment* provides declarative updates for Pods and ReplicaSets.

You describe a *desired state* in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

**Note:**

Do not manage ReplicaSets owned by a Deployment. Consider opening an issue in the main Kubernetes repository if your use case is not covered below.

## Use Case

The following are typical use cases for Deployments:

- Create a Deployment to rollout a ReplicaSet. The ReplicaSet creates Pods in the background. Check the status of the rollout to see if it succeeds or not.
- Declare the new state of the Pods by updating the PodTemplateSpec of the Deployment. A new ReplicaSet is created, and the Deployment gradually scales it up while scaling down the old ReplicaSet, ensuring Pods are replaced at a controlled rate. Each new ReplicaSet updates the revision of the Deployment.
- Rollback to an earlier Deployment revision if the current state of the Deployment is not stable. Each rollback updates the revision of the Deployment.
- Scale up the Deployment to facilitate more load.
- Pause the rollout of a Deployment to apply multiple fixes to its PodTemplateSpec and then resume it to start a new rollout.
- Use the status of the Deployment as an indicator that a rollout has stuck.
- Clean up older ReplicaSets that you don't need anymore.

## Creating a Deployment

The following is an example of a Deployment. It creates a ReplicaSet to bring up three `nginx` Pods:

`controllers/nginx-deployment.yaml` Copy controllers/nginx-deployment.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: nginx-deployment  labels:    app: nginxspec:  replicas: 3  selector:    matchLabels:      app: ng
```

In this example:

- A Deployment named `nginx-deployment` is created, indicated by the `.metadata.name` field. This name will become the basis for the ReplicaSets and Pods which are created later. See Writing a Deployment Spec for more details.

- The Deployment creates a ReplicaSet that creates three replicated Pods, indicated by the `.spec.replicas` field.

- The `.spec.selector` field defines how the created ReplicaSet finds which Pods to manage. In this case, you select a label that is defined in the Pod template (`app: nginx`). However, more sophisticated selection rules are possible, as long as the Pod template itself satisfies the rule.

  **Note:**

  The `.spec.selector.matchLabels` field is a map of {key,value} pairs. A single {key,value} in the `matchLabels` map is equivalent to an element of `matchExpressions`, whose `key` field is "key", the `operator` is "In", and the `values` array contains only "value". All of the requirements, from both `matchLabels` and `matchExpressions`, must be satisfied in order to match.

- The `.spec.template` field contains the following sub-fields:

  - The Pods are labeled `app: nginx`using the `.metadata.labels` field.
  - The Pod template's specification, or `.spec` field, indicates that the Pods run one container, `nginx`, which runs the `nginx` Docker Hub image at version 1.14.2.
  - Create one container and name it `nginx` using the `.spec.containers[0].name` field.

Before you begin, make sure your Kubernetes cluster is up and running. Follow the steps given below to create the above Deployment:

1. Create the Deployment by running the following command:

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

2. Run `kubectl get deployments` to check if the Deployment was created.

   If the Deployment is still being created, the output is similar to the following:

   ```
   NAME               READY   UP-TO-DATE   AVAILABLE   AGE
   nginx-deployment   0/3     0            0           1s
   ```

   When you inspect the Deployments in your cluster, the following fields are displayed:

   - `NAME` lists the names of the Deployments in the namespace.

    - `READY` displays how many replicas of the application are available to your users. It follows the pattern ready/desired.
    - `UP-TO-DATE` displays the number of replicas that have been updated to achieve the desired state.
    - `AVAILABLE` displays how many replicas of the application are available to your users.
    - `AGE` displays the amount of time that the application has been running.

    Notice how the number of desired replicas is 3 according to `.spec.replicas` field.

3. To see the Deployment rollout status, run `kubectl rollout status deployment/nginx-deployment`.

   The output is similar to:

   ```
   Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
   deployment "nginx-deployment" successfully rolled out
   ```

4. Run the `kubectl get deployments` again a few seconds later. The output is similar to this:

   ```
   NAME               READY   UP-TO-DATE   AVAILABLE   AGE
   nginx-deployment   3/3     3            3           18s
   ```

   Notice that the Deployment has created all three replicas, and all replicas are up-to-date (they contain the latest Pod template) and available.

5. To see the ReplicaSet (`rs`) created by the Deployment, run `kubectl get rs`. The output is similar to this:

   ```
   NAME                          DESIRED   CURRENT   READY   AGE
   nginx-deployment-75675f5897   3         3         3       18s
   ```

   ReplicaSet output shows the following fields:

    - `NAME` lists the names of the ReplicaSets in the namespace.
    - `DESIRED` displays the desired number of *replicas* of the application, which you define when you create the Deployment. This is the *desired state*.
    - `CURRENT` displays how many replicas are currently running.
    - `READY` displays how many replicas of the application are available to your users.
    - `AGE` displays the amount of time that the application has been running.

   Notice that the name of the ReplicaSet is always formatted as `[DEPLOYMENT-NAME]-[HASH]`. This name will become the basis for the Pods which are created.

   The `HASH` string is the same as the `pod-template-hash` label on the ReplicaSet.

6. To see the labels automatically generated for each Pod, run `kubectl get pods --show-labels`. The output is similar to:

   ```
   NAME                                READY     STATUS    RESTARTS   AGE       LABELS
   nginx-deployment-75675f5897-7ci7o   1/1       Running   0          18s       app=nginx,pod-template-hash=75675f5897
   nginx-deployment-75675f5897-kzszj   1/1       Running   0          18s       app=nginx,pod-template-hash=75675f5897
   nginx-deployment-75675f5897-qqcnn   1/1       Running   0          18s       app=nginx,pod-template-hash=75675f5897
   ```

   The created ReplicaSet ensures that there are three `nginx` Pods.

**Note:**

You must specify an appropriate selector and Pod template labels in a Deployment (in this case, `app: nginx`).

Do not overlap labels or selectors with other controllers (including other Deployments and StatefulSets). Kubernetes doesn't stop you from overlapping, and if multiple controllers have overlapping selectors those controllers might conflict and behave unexpectedly.

### Pod-template-hash label

**Caution:**

Do not change this label.

The `pod-template-hash` label is added by the Deployment controller to every ReplicaSet that a Deployment creates or adopts.

This label ensures that child ReplicaSets of a Deployment do not overlap. It is generated by hashing the `PodTemplate` of the ReplicaSet and using the resulting hash as the label value that is added to the ReplicaSet selector, Pod template labels, and in any existing Pods that the ReplicaSet might have.

## Updating a Deployment

**Note:**

A Deployment's rollout is triggered if and only if the Deployment's Pod template (that is, `.spec.template`) is changed, for example if the labels or container images of the template are updated. Other updates, such as scaling the Deployment, do not trigger a rollout.

Follow the steps given below to update your Deployment:

1. Let's update the nginx Pods to use the `nginx:1.16.1` image instead of the `nginx:1.14.2` image.

   ```
   kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.16.1
   ```

   or use the following command:

   ```
   kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1
   ```

where `deployment/nginx-deployment` indicates the Deployment, `nginx` indicates the Container the update will take place and `nginx:1.16.1` indicates the new image and its tag.

The output is similar to:

```
deployment.apps/nginx-deployment image updated
```

Alternatively, you can `edit` the Deployment and change `.spec.template.spec.containers[0].image` from `nginx:1.14.2` to `nginx:1.16.1`:

```
kubectl edit deployment/nginx-deployment
```

The output is similar to:

```
deployment.apps/nginx-deployment edited
```

2. To see the rollout status, run:

```
kubectl rollout status deployment/nginx-deployment
```

The output is similar to this:

```
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
```

or

```
deployment "nginx-deployment" successfully rolled out
```

Get more details on your updated Deployment:

- After the rollout succeeds, you can view the Deployment by running `kubectl get deployments`. The output is similar to this:

```
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3            3           36s
```

- Run `kubectl get rs` to see that the Deployment updated the Pods by creating a new ReplicaSet and scaling it up to 3 replicas, as well as scaling down the old ReplicaSet to 0 replicas.

```
kubectl get rs
```

The output is similar to this:

```
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-1564180365   3         3         3       6s
nginx-deployment-2035384211   0         0         0       36s
```

- Running `get pods` should now show only the new Pods:

```
kubectl get pods
```

The output is similar to this:

```
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-1564180365-khku8   1/1       Running   0          14s
nginx-deployment-1564180365-nacti   1/1       Running   0          14s
nginx-deployment-1564180365-z9gth   1/1       Running   0          14s
```

Next time you want to update these Pods, you only need to update the Deployment's Pod template again.

Deployment ensures that only a certain number of Pods are down while they are being updated. By default, it ensures that at least 75% of the desired number of Pods are up (25% max unavailable).

Deployment also ensures that only a certain number of Pods are created above the desired number of Pods. By default, it ensures that at most 125% of the desired number of Pods are up (25% max surge).

For example, if you look at the above Deployment closely, you will see that it first creates a new Pod, then deletes an old Pod, and creates another new one. It does not kill old Pods until a sufficient number of new Pods have come up, and does not create new Pods until a sufficient number of old Pods have been killed. It makes sure that at least 3 Pods are available and that at max 4 Pods in total are available. In case of a Deployment with 4 replicas, the number of Pods would be between 3 and 5.

- Get details of your Deployment:

```
kubectl describe deployments
```

The output is similar to this:

```
Name:                   nginx-deployment
Namespace:              default
CreationTimestamp:      Thu, 30 Nov 2017 10:56:25 +0000
Labels:                 app=nginx
Annotations:            deployment.kubernetes.io/revision=2
Selector:               app=nginx
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
   Containers:
    nginx:
      Image:        nginx:1.16.1
      Port:         80/TCP
      Environment:  <none>
      Mounts:       <none>
    Volumes:        <none>
  Conditions:
    Type           Status  Reason
    ----           ------  ------
    Available      True    MinimumReplicasAvailable
    Progressing    True    NewReplicaSetAvailable
  OldReplicaSets:  <none>
  NewReplicaSet:   nginx-deployment-1564180365 (3/3 replicas created)
  Events:
    Type    Reason             Age   From                   Message
    ----    ------             ----  ----                   -------
    Normal  ScalingReplicaSet  2m    deployment-controller  Scaled up replica set nginx-deployment-2035384211 to 3
    Normal  ScalingReplicaSet  24s   deployment-controller  Scaled up replica set nginx-deployment-1564180365 to 1
    Normal  ScalingReplicaSet  22s   deployment-controller  Scaled down replica set nginx-deployment-2035384211 to 2
    Normal  ScalingReplicaSet  22s   deployment-controller  Scaled up replica set nginx-deployment-1564180365 to 2
    Normal  ScalingReplicaSet  19s   deployment-controller  Scaled down replica set nginx-deployment-2035384211 to 1
    Normal  ScalingReplicaSet  19s   deployment-controller  Scaled up replica set nginx-deployment-1564180365 to 3
    Normal  ScalingReplicaSet  14s   deployment-controller  Scaled down replica set nginx-deployment-2035384211 to 0
```

Here you see that when you first created the Deployment, it created a ReplicaSet (nginx-deployment-2035384211) and scaled it up to 3 replicas directly. When you updated the Deployment, it created a new ReplicaSet (nginx-deployment-1564180365) and scaled it up to 1 and waited for it to come up. Then it scaled down the old ReplicaSet to 2 and scaled up the new ReplicaSet to 2 so that at least 3 Pods were available and at most 4 Pods were created at all times. It then continued scaling up and down the new and the old ReplicaSet, with the same rolling update strategy. Finally, you'll have 3 available replicas in the new ReplicaSet, and the old ReplicaSet is scaled down to 0.

**Note:**

Kubernetes doesn't count terminating Pods when calculating the number of `availableReplicas`, which must be between `replicas - maxUnavailable` and `replicas + maxSurge`. As a result, you might notice that there are more Pods than expected during a rollout, and that the total resources consumed by the Deployment is more than `replicas + maxSurge` until the `terminationGracePeriodSeconds` of the terminating Pods expires.

### Rollover (aka multiple updates in-flight)

Each time a new Deployment is observed by the Deployment controller, a ReplicaSet is created to bring up the desired Pods. If the Deployment is updated, the existing ReplicaSet that controls Pods whose labels match `.spec.selector` but whose template does not match `.spec.template` is scaled down. Eventually, the new ReplicaSet is scaled to `.spec.replicas` and all old ReplicaSets is scaled to 0.

If you update a Deployment while an existing rollout is in progress, the Deployment creates a new ReplicaSet as per the update and start scaling that up, and rolls over the ReplicaSet that it was scaling up previously -- it will add it to its list of old ReplicaSets and start scaling it down.

For example, suppose you create a Deployment to create 5 replicas of `nginx:1.14.2`, but then update the Deployment to create 5 replicas of `nginx:1.16.1`, when only 3 replicas of `nginx:1.14.2` had been created. In that case, the Deployment immediately starts killing the 3 `nginx:1.14.2` Pods that it had created, and starts creating `nginx:1.16.1` Pods. It does not wait for the 5 replicas of `nginx:1.14.2` to be created before changing course.

### Label selector updates

It is generally discouraged to make label selector updates and it is suggested to plan your selectors up front. In any case, if you need to perform a label selector update, exercise great caution and make sure you have grasped all of the implications.

**Note:**

In API version `apps/v1`, a Deployment's label selector is immutable after it gets created.

- Selector additions require the Pod template labels in the Deployment spec to be updated with the new label too, otherwise a validation error is returned. This change is a non-overlapping one, meaning that the new selector does not select ReplicaSets and Pods created with the old selector, resulting in orphaning all old ReplicaSets and creating a new ReplicaSet.
- Selector updates changes the existing value in a selector key -- result in the same behavior as additions.
- Selector removals removes an existing key from the Deployment selector -- do not require any changes in the Pod template labels. Existing ReplicaSets are not orphaned, and a new ReplicaSet is not created, but note that the removed label still exists in any existing Pods and ReplicaSets.

## Rolling Back a Deployment

Sometimes, you may want to rollback a Deployment; for example, when the Deployment is not stable, such as crash looping. By default, all of the Deployment's rollout history is kept in the system so that you can rollback anytime you want (you can change that by modifying revision history limit).

**Note:**

A Deployment's revision is created when a Deployment's rollout is triggered. This means that the new revision is created if and only if the Deployment's Pod template (`.spec.template`) is changed, for example if you update the labels or container images of the template. Other updates, such as scaling the Deployment, do not create a Deployment revision, so that you can facilitate simultaneous manual- or auto-scaling. This means that when you roll back to an earlier revision, only the Deployment's Pod template part is rolled back.

- Suppose that you made a typo while updating the Deployment, by putting the image name as `nginx:1.161` instead of `nginx:1.16.1`:

  ```
  kubectl set image deployment/nginx-deployment nginx=nginx:1.161
  ```

  The output is similar to this:

  ```
  deployment.apps/nginx-deployment image updated
  ```

- The rollout gets stuck. You can verify it by checking the rollout status:

  ```
  kubectl rollout status deployment/nginx-deployment
  ```

  The output is similar to this:

  ```
  Waiting for rollout to finish: 1 out of 3 new replicas have been updated...
  ```

- Press Ctrl-C to stop the above rollout status watch. For more information on stuck rollouts, read more here.

- You see that the number of old replicas (adding the replica count from `nginx-deployment-1564180365` and `nginx-deployment-2035384211`) is 3, and the number of new replicas (from `nginx-deployment-3066724191`) is 1.

  ```
  kubectl get rs
  ```

  The output is similar to this:

  ```
  NAME                          DESIRED   CURRENT   READY   AGE
  nginx-deployment-1564180365   3         3         3       25s
  nginx-deployment-2035384211   0         0         0       36s
  nginx-deployment-3066724191   1         1         0       6s
  ```

- Looking at the Pods created, you see that 1 Pod created by new ReplicaSet is stuck in an image pull loop.

  ```
  kubectl get pods
  ```

  The output is similar to this:

  ```
  NAME                                READY     STATUS             RESTARTS   AGE
  nginx-deployment-1564180365-70iae   1/1       Running            0          25s
  nginx-deployment-1564180365-jbqqo   1/1       Running            0          25s
  nginx-deployment-1564180365-hysrc   1/1       Running            0          25s
  nginx-deployment-3066724191-08mng   0/1       ImagePullBackOff   0          6s
  ```

  **Note:**

  The Deployment controller stops the bad rollout automatically, and stops scaling up the new ReplicaSet. This depends on the rollingUpdate parameters (`maxUnavailable` specifically) that you have specified. Kubernetes by default sets the value to 25%.

- Get the description of the Deployment:

  ```
  kubectl describe deployment
  ```

  The output is similar to this:

  ```
  Name:           nginx-deployment
  Namespace:      default
  CreationTimestamp:  Tue, 15 Mar 2016 14:48:04 -0700
  Labels:         app=nginx
  Selector:       app=nginx
  Replicas:       3 desired | 1 updated | 4 total | 3 available | 1 unavailable
  StrategyType:       RollingUpdate
  MinReadySeconds:    0
  RollingUpdateStrategy:  25% max unavailable, 25% max surge
  Pod Template:
    Labels:  app=nginx
    Containers:
     nginx:
      Image:        nginx:1.161
      Port:         80/TCP
      Host Port:    0/TCP
      Environment:  <none>
      Mounts:       <none>
    Volumes:        <none>
  Conditions:
    Type           Status  Reason
    ----           ------  ------
    Available      True    MinimumReplicasAvailable
    Progressing    True    ReplicaSetUpdated
  OldReplicaSets:     nginx-deployment-1564180365 (3/3 replicas created)
  NewReplicaSet:      nginx-deployment-3066724191 (1/1 replicas created)
  Events:
    FirstSeen LastSeen    Count   From                    SubObjectPath   Type        Reason              Message
    --------- --------    -----   ----                    -------------   --------    ------              -------
    1m        1m          1       {deployment-controller }                Normal      ScalingReplicaSet   Scaled up replica set
    22s       22s         1       {deployment-controller }                Normal      ScalingReplicaSet   Scaled up replica set
    22s       22s         1       {deployment-controller }                Normal      ScalingReplicaSet   Scaled down replica s
    22s       22s         1       {deployment-controller }                Normal      ScalingReplicaSet   Scaled up replica set
    21s       21s         1       {deployment-controller }                Normal      ScalingReplicaSet   Scaled down replica s
    21s       21s         1       {deployment-controller }                Normal      ScalingReplicaSet   Scaled up replica set
    13s       13s         1       {deployment-controller }                Normal      ScalingReplicaSet   Scaled down replica s
    13s       13s         1       {deployment-controller }                Normal      ScalingReplicaSet   Scaled up replica set
  ```

  To fix this, you need to rollback to a previous revision of Deployment that is stable.

### Checking Rollout History of a Deployment

Follow the steps given below to check the rollout history:

1. First, check the revisions of this Deployment:

```
kubectl rollout history deployment/nginx-deployment
```

The output is similar to this:

```
deployments "nginx-deployment"
REVISION    CHANGE-CAUSE
1           <none>
2           <none>
3           <none>
```

`CHANGE-CAUSE` is copied from the Deployment annotation `kubernetes.io/change-cause` to its revisions upon creation. You can specify the`CHANGE-CAUSE` message by:

- Annotating the Deployment with `kubectl annotate deployment/nginx-deployment kubernetes.io/change-cause="image updated to 1.16.1"`
- Manually editing the manifest of the resource.
- Using tooling that sets the annotation automatically.

**Note:**

In older versions of Kubernetes, you could use the `--record` flag with kubectl commands to automatically populate the `CHANGE-CAUSE` field. This flag is deprecated and will be removed in a future release.

2. To see the details of each revision, run:

```
kubectl rollout history deployment/nginx-deployment --revision=2
```

The output is similar to this:

```
deployments "nginx-deployment" revision 2
  Labels:       app=nginx
          pod-template-hash=1159050644
  Containers:
   nginx:
    Image:      nginx:1.16.1
    Port:       80/TCP
     QoS Tier:
        cpu:      BestEffort
        memory:   BestEffort
    Environment Variables:      <none>
  No volumes.
```

### Rolling Back to a Previous Revision

Follow the steps given below to rollback the Deployment from the current version to the previous version, which is version 2.

1. Now you've decided to undo the current rollout and rollback to the previous revision:

```
kubectl rollout undo deployment/nginx-deployment
```

The output is similar to this:

```
deployment.apps/nginx-deployment rolled back
```

Alternatively, you can rollback to a specific revision by specifying it with `--to-revision`:

```
kubectl rollout undo deployment/nginx-deployment --to-revision=2
```

The output is similar to this:

```
deployment.apps/nginx-deployment rolled back
```

For more details about rollout related commands, read `kubectl rollout`.

The Deployment is now rolled back to a previous stable revision. As you can see, a `DeploymentRollback` event for rolling back to revision 2 is generated from Deployment controller.

2. Check if the rollback was successful and the Deployment is running as expected, run:

```
kubectl get deployment nginx-deployment
```

The output is similar to this:

```
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3            3           30m
```

3. Get the description of the Deployment:

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

The output is similar to this:

```
Name:                   nginx-deployment
Namespace:              default
CreationTimestamp:      Sun, 02 Sep 2018 18:17:55 -0500
Labels:                 app=nginx
Annotations:            deployment.kubernetes.io/revision=4
Selector:               app=nginx
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0 unavailable
```

```
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
   nginx:
    Image:        nginx:1.16.1
    Port:         80/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   nginx-deployment-c4747d96c (3/3 replicas created)
Events:
  Type    Reason              Age   From                   Message
  ----    ------              ----  ----                   -------
  Normal  ScalingReplicaSet   12m   deployment-controller  Scaled up replica set nginx-deployment-75675f5897 to 3
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled up replica set nginx-deployment-c4747d96c to 1
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled down replica set nginx-deployment-75675f5897 to 2
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled up replica set nginx-deployment-c4747d96c to 2
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled down replica set nginx-deployment-75675f5897 to 1
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled up replica set nginx-deployment-c4747d96c to 3
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled down replica set nginx-deployment-75675f5897 to 0
  Normal  ScalingReplicaSet   11m   deployment-controller  Scaled up replica set nginx-deployment-595696685f to 1
  Normal  DeploymentRollback  15s   deployment-controller  Rolled back deployment "nginx-deployment" to revision 2
  Normal  ScalingReplicaSet   15s   deployment-controller  Scaled down replica set nginx-deployment-595696685f to 0
```

## Scaling a Deployment

You can scale a Deployment by using the following command:

```
kubectl scale deployment/nginx-deployment --replicas=10
```

The output is similar to this:

```
deployment.apps/nginx-deployment scaled
```

Assuming horizontal Pod autoscaling is enabled in your cluster, you can set up an autoscaler for your Deployment and choose the minimum and maximum number of Pods you want to run based on the CPU utilization of your existing Pods.

```
kubectl autoscale deployment/nginx-deployment --min=10 --max=15 --cpu-percent=80
```

The output is similar to this:

```
deployment.apps/nginx-deployment scaled
```

### Proportional scaling

RollingUpdate Deployments support running multiple versions of an application at the same time. When you or an autoscaler scales a RollingUpdate Deployment that is in the middle of a rollout (either in progress or paused), the Deployment controller balances the additional replicas in the existing active ReplicaSets (ReplicaSets with Pods) in order to mitigate risk. This is called *proportional scaling*.

For example, you are running a Deployment with 10 replicas, maxSurge=3, and maxUnavailable=2.

- Ensure that the 10 replicas in your Deployment are running.

  ```
  kubectl get deploy
  ```

  The output is similar to this:

  ```
  NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
  nginx-deployment     10        10        10           10          50s
  ```

- You update to a new image which happens to be unresolvable from inside the cluster.

  ```
  kubectl set image deployment/nginx-deployment nginx=nginx:sometag
  ```

  The output is similar to this:

  ```
  deployment.apps/nginx-deployment image updated
  ```

- The image update starts a new rollout with ReplicaSet nginx-deployment-1989198191, but it's blocked due to the `maxUnavailable` requirement that you mentioned above. Check out the rollout status:

  ```
  kubectl get rs
  ```

  The output is similar to this:

  ```
  NAME                          DESIRED   CURRENT   READY     AGE
  nginx-deployment-1989198191   5         5         0         9s
  nginx-deployment-618515232    8         8         8         1m
  ```

- Then a new scaling request for the Deployment comes along. The autoscaler increments the Deployment replicas to 15. The Deployment controller needs to decide where to add these new 5 replicas. If you weren't using proportional scaling, all 5 of them would be added in the new ReplicaSet. With proportional scaling, you spread the additional replicas across all ReplicaSets. Bigger proportions go to the ReplicaSets with the most replicas and lower proportions go to ReplicaSets with less replicas. Any leftovers are added to the ReplicaSet with the most replicas. ReplicaSets with zero replicas are not scaled up.

In our example above, 3 replicas are added to the old ReplicaSet and 2 replicas are added to the new ReplicaSet. The rollout process should eventually move all replicas to the new ReplicaSet, assuming the new replicas become healthy. To confirm this, run:

```
kubectl get deploy
```

The output is similar to this:

```
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment     15        18        7            8           7m
```

The rollout status confirms how the replicas were added to each ReplicaSet.

```
kubectl get rs
```

The output is similar to this:

```
NAME                          DESIRED   CURRENT   READY     AGE
nginx-deployment-1989198191   7         7         0         7m
nginx-deployment-618515232    11        11        11        7m
```

## Pausing and Resuming a rollout of a Deployment

When you update a Deployment, or plan to, you can pause rollouts for that Deployment before you trigger one or more updates. When you're ready to apply those changes, you resume rollouts for the Deployment. This approach allows you to apply multiple fixes in between pausing and resuming without triggering unnecessary rollouts.

- For example, with a Deployment that was created:

  Get the Deployment details:

  ```
  kubectl get deploy
  ```

  The output is similar to this:

  ```
  NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
  nginx     3         3         3            3           1m
  ```

  Get the rollout status:

  ```
  kubectl get rs
  ```

  The output is similar to this:

  ```
  NAME               DESIRED   CURRENT   READY     AGE
  nginx-2142116321   3         3         3         1m
  ```

- Pause by running the following command:

  ```
  kubectl rollout pause deployment/nginx-deployment
  ```

  The output is similar to this:

  ```
  deployment.apps/nginx-deployment paused
  ```

- Then update the image of the Deployment:

  ```
  kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1
  ```

  The output is similar to this:

  ```
  deployment.apps/nginx-deployment image updated
  ```

- Notice that no new rollout started:

  ```
  kubectl rollout history deployment/nginx-deployment
  ```

  The output is similar to this:

  ```
  deployments "nginx"
  REVISION  CHANGE-CAUSE
  1   <none>
  ```

- Get the rollout status to verify that the existing ReplicaSet has not changed:

  ```
  kubectl get rs
  ```

  The output is similar to this:

  ```
  NAME               DESIRED   CURRENT   READY     AGE
  nginx-2142116321   3         3         3         2m
  ```

- You can make as many updates as you wish, for example, update the resources that will be used:

```
kubectl set resources deployment/nginx-deployment -c=nginx --limits=cpu=200m,memory=512Mi
```

The output is similar to this:

```
deployment.apps/nginx-deployment resource requirements updated
```

The initial state of the Deployment prior to pausing its rollout will continue its function, but new updates to the Deployment will not have any effect as long as the Deployment rollout is paused.

- Eventually, resume the Deployment rollout and observe a new ReplicaSet coming up with all the new updates:

  ```
  kubectl rollout resume deployment/nginx-deployment
  ```

  The output is similar to this:

  ```
  deployment.apps/nginx-deployment resumed
  ```

- Watch the status of the rollout until it's done.

  ```
  kubectl get rs --watch
  ```

  The output is similar to this:

  ```
  NAME               DESIRED   CURRENT   READY     AGE
  nginx-2142116321   2         2         2         2m
  nginx-3926361531   2         2         0         6s
  nginx-3926361531   2         2         1         18s
  nginx-2142116321   1         2         2         2m
  nginx-2142116321   1         2         2         2m
  nginx-3926361531   3         2         1         18s
  nginx-3926361531   3         2         1         18s
  nginx-2142116321   1         1         1         2m
  nginx-3926361531   3         3         1         18s
  nginx-3926361531   3         3         2         19s
  nginx-2142116321   0         1         1         2m
  nginx-2142116321   0         1         1         2m
  nginx-2142116321   0         0         0         2m
  nginx-3926361531   3         3         3         20s
  ```

- Get the status of the latest rollout:

  ```
  kubectl get rs
  ```

  The output is similar to this:

  ```
  NAME               DESIRED   CURRENT   READY     AGE
  nginx-2142116321   0         0         0         2m
  nginx-3926361531   3         3         3         28s
  ```

**Note:**

You cannot rollback a paused Deployment until you resume it.

# Deployment status

A Deployment enters various states during its lifecycle. It can be progressing while rolling out a new ReplicaSet, it can be complete, or it can fail to progress.

## Progressing Deployment

Kubernetes marks a Deployment as *progressing* when one of the following tasks is performed:

- The Deployment creates a new ReplicaSet.
- The Deployment is scaling up its newest ReplicaSet.
- The Deployment is scaling down its older ReplicaSet(s).
- New Pods become ready or available (ready for at least MinReadySeconds).

When the rollout becomes “progressing”, the Deployment controller adds a condition with the following attributes to the Deployment's `.status.conditions`:

- `type: Progressing`
- `status: "True"`
- `reason: NewReplicaSetCreated` | `reason: FoundNewReplicaSet` | `reason: ReplicaSetUpdated`

You can monitor the progress for a Deployment by using `kubectl rollout status`.

## Complete Deployment

Kubernetes marks a Deployment as *complete* when it has the following characteristics:

- All of the replicas associated with the Deployment have been updated to the latest version you've specified, meaning any updates you've requested have been completed.
- All of the replicas associated with the Deployment are available.
- No old replicas for the Deployment are running.

When the rollout becomes “complete”, the Deployment controller sets a condition with the following attributes to the Deployment's `.status.conditions`:

- `type: Progressing`
- `status: "True"`
- `reason: NewReplicaSetAvailable`

This `Progressing` condition will retain a status value of `"True"` until a new rollout is initiated. The condition holds even when availability of replicas changes (which does instead affect the `Available` condition).

You can check if a Deployment has completed by using `kubectl rollout status`. If the rollout completed successfully, `kubectl rollout status` returns a zero exit code.

```
kubectl rollout status deployment/nginx-deployment
```

The output is similar to this:

```
Waiting for rollout to finish: 2 of 3 updated replicas are available...
deployment "nginx-deployment" successfully rolled out
```

and the exit status from `kubectl rollout` is 0 (success):

```
echo $?
```

```
0
```

### Failed Deployment

Your Deployment may get stuck trying to deploy its newest ReplicaSet without ever completing. This can occur due to some of the following factors:

- Insufficient quota
- Readiness probe failures
- Image pull errors
- Insufficient permissions
- Limit ranges
- Application runtime misconfiguration

One way you can detect this condition is to specify a deadline parameter in your Deployment spec: ([`.spec.progressDeadlineSeconds`]). `.spec.progressDeadlineSeconds` denotes the number of seconds the Deployment controller waits before indicating (in the Deployment status) that the Deployment progress has stalled.

The following `kubectl` command sets the spec with `progressDeadlineSeconds` to make the controller report lack of progress of a rollout for a Deployment after 10 minutes:

```
kubectl patch deployment/nginx-deployment -p '{"spec":{"progressDeadlineSeconds":600}}'
```

The output is similar to this:

```
deployment.apps/nginx-deployment patched
```

Once the deadline has been exceeded, the Deployment controller adds a DeploymentCondition with the following attributes to the Deployment's `.status.conditions`:

- `type: Progressing`
- `status: "False"`
- `reason: ProgressDeadlineExceeded`

This condition can also fail early and is then set to status value of `"False"` due to reasons as `ReplicaSetCreateError`. Also, the deadline is not taken into account anymore once the Deployment rollout completes.

See the Kubernetes API conventions for more information on status conditions.

**Note:**

Kubernetes takes no action on a stalled Deployment other than to report a status condition with `reason: ProgressDeadlineExceeded`. Higher level orchestrators can take advantage of it and act accordingly, for example, rollback the Deployment to its previous version.

**Note:**

If you pause a Deployment rollout, Kubernetes does not check progress against your specified deadline. You can safely pause a Deployment rollout in the middle of a rollout and resume without triggering the condition for exceeding the deadline.

You may experience transient errors with your Deployments, either due to a low timeout that you have set or due to any other kind of error that can be treated as transient. For example, let's suppose you have insufficient quota. If you describe the Deployment you will notice the following section:

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

The output is similar to this:

```
<...>
Conditions:
  Type            Status  Reason
  ----            ------  ------
  Available       True    MinimumReplicasAvailable
  Progressing     True    ReplicaSetUpdated
  ReplicaFailure  True    FailedCreate
<...>
```

If you run `kubectl get deployment nginx-deployment -o yaml`, the Deployment status is similar to this:

```
status:
  availableReplicas: 2
  conditions:
  - lastTransitionTime: 2016-10-04T12:25:39Z
    lastUpdateTime: 2016-10-04T12:25:39Z
    message: Replica set "nginx-deployment-4262182780" is progressing.
    reason: ReplicaSetUpdated
    status: "True"
    type: Progressing
  - lastTransitionTime: 2016-10-04T12:25:42Z
    lastUpdateTime: 2016-10-04T12:25:42Z
    message: Deployment has minimum availability.
    reason: MinimumReplicasAvailable
    status: "True"
    type: Available
  - lastTransitionTime: 2016-10-04T12:25:39Z
    lastUpdateTime: 2016-10-04T12:25:39Z
    message: 'Error creating: pods "nginx-deployment-4262182780-" is forbidden: exceeded quota:
      object-counts, requested: pods=1, used: pods=3, limited: pods=2'
    reason: FailedCreate
    status: "True"
    type: ReplicaFailure
  observedGeneration: 3
  replicas: 2
  unavailableReplicas: 2
```

Eventually, once the Deployment progress deadline is exceeded, Kubernetes updates the status and the reason for the Progressing condition:

```
Conditions:
  Type            Status  Reason
  ----            ------  ------
  Available       True    MinimumReplicasAvailable
  Progressing     False   ProgressDeadlineExceeded
  ReplicaFailure  True    FailedCreate
```

You can address an issue of insufficient quota by scaling down your Deployment, by scaling down other controllers you may be running, or by increasing quota in your namespace. If you satisfy the quota conditions and the Deployment controller then completes the Deployment rollout, you'll see the Deployment's status update with a successful condition (`status: "True"` and `reason: NewReplicaSetAvailable`).

```
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
```

`type: Available` with `status: "True"` means that your Deployment has minimum availability. Minimum availability is dictated by the parameters specified in the deployment strategy. `type: Progressing` with `status: "True"` means that your Deployment is either in the middle of a rollout and it is progressing or that it has successfully completed its progress and the minimum required new replicas are available (see the Reason of the condition for the particulars - in our case `reason: NewReplicaSetAvailable` means that the Deployment is complete).

You can check if a Deployment has failed to progress by using `kubectl rollout status`. `kubectl rollout status` returns a non-zero exit code if the Deployment has exceeded the progression deadline.

```
kubectl rollout status deployment/nginx-deployment
```

The output is similar to this:

```
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
error: deployment "nginx" exceeded its progress deadline
```

and the exit status from `kubectl rollout` is 1 (indicating an error):

```
echo $?
```

```
1
```

### Operating on a failed deployment

All actions that apply to a complete Deployment also apply to a failed Deployment. You can scale it up/down, roll back to a previous revision, or even pause it if you need to apply multiple tweaks in the Deployment Pod template.

## Clean up Policy

You can set `.spec.revisionHistoryLimit` field in a Deployment to specify how many old ReplicaSets for this Deployment you want to retain. The rest will be garbage-collected in the background. By default, it is 10.

**Note:**

Explicitly setting this field to 0, will result in cleaning up all the history of your Deployment thus that Deployment will not be able to roll back.

The cleanup only starts **after** a Deployment reaches a complete state. If you set `.spec.revisionHistoryLimit` to 0, any rollout nonetheless triggers creation of a new ReplicaSet before Kubernetes removes the old one.

Even with a non-zero revision history limit, you can have more ReplicaSets than the limit you configure. For example, if pods are crash looping, and there are multiple rolling updates events triggered over time, you might end up with more ReplicaSets than the `.spec.revisionHistoryLimit` because the Deployment never reaches a complete state.

## Canary Deployment

If you want to roll out releases to a subset of users or servers using the Deployment, you can create multiple Deployments, one for each release, following the canary pattern described in managing resources.

## Writing a Deployment Spec

As with all other Kubernetes configs, a Deployment needs `.apiVersion`, `.kind`, and `.metadata` fields. For general information about working with config files, see deploying applications, configuring containers, and using kubectl to manage resources documents.

When the control plane creates new Pods for a Deployment, the `.metadata.name` of the Deployment is part of the basis for naming those Pods. The name of a Deployment must be a valid DNS subdomain value, but this can produce unexpected results for the Pod hostnames. For best compatibility, the name should follow the more restrictive rules for a DNS label.

A Deployment also needs a `.spec` section.

### Pod Template

The `.spec.template` and `.spec.selector` are the only required fields of the `.spec`.

The `.spec.template` is a Pod template. It has exactly the same schema as a Pod, except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a Pod template in a Deployment must specify appropriate labels and an appropriate restart policy. For labels, make sure not to overlap with other controllers. See selector.

Only a `.spec.template.spec.restartPolicy` equal to `Always` is allowed, which is the default if not specified.

### Replicas

`.spec.replicas` is an optional field that specifies the number of desired Pods. It defaults to 1.

Should you manually scale a Deployment, example via `kubectl scale deployment deployment --replicas=X`, and then you update that Deployment based on a manifest (for example: by running `kubectl apply -f deployment.yaml`), then applying that manifest overwrites the manual scaling that you previously did.

If a HorizontalPodAutoscaler (or any similar API for horizontal scaling) is managing scaling for a Deployment, don't set `.spec.replicas`.

Instead, allow the Kubernetes control plane to manage the `.spec.replicas` field automatically.

### Selector

`.spec.selector` is a required field that specifies a label selector for the Pods targeted by this Deployment.

`.spec.selector` must match `.spec.template.metadata.labels`, or it will be rejected by the API.

In API version `apps/v1`, `.spec.selector` and `.metadata.labels` do not default to `.spec.template.metadata.labels` if not set. So they must be set explicitly. Also note that `.spec.selector` is immutable after creation of the Deployment in `apps/v1`.

A Deployment may terminate Pods whose labels match the selector if their template is different from `.spec.template` or if the total number of such Pods exceeds `.spec.replicas`. It brings up new Pods with `.spec.template` if the number of Pods is less than the desired number.

**Note:**

You should not create other Pods whose labels match this selector, either directly, by creating another Deployment, or by creating another controller such as a ReplicaSet or a ReplicationController. If you do so, the first Deployment thinks that it created these other Pods. Kubernetes does not stop you from doing this.

If you have multiple controllers that have overlapping selectors, the controllers will fight with each other and won't behave correctly.

### Strategy

`.spec.strategy` specifies the strategy used to replace old Pods by new ones. `.spec.strategy.type` can be "Recreate" or "RollingUpdate". "RollingUpdate" is the default value.

#### Recreate Deployment

All existing Pods are killed before new ones are created when `.spec.strategy.type==Recreate`.

**Note:**

This will only guarantee Pod termination previous to creation for upgrades. If you upgrade a Deployment, all Pods of the old revision will be terminated immediately. Successful removal is awaited before any Pod of the new revision is created. If you manually delete a Pod, the lifecycle is controlled by the ReplicaSet and the replacement will be created immediately (even if the old Pod is still in a Terminating state). If you need an "at most" guarantee for your Pods, you should consider using a StatefulSet.

#### Rolling Update Deployment

The Deployment updates Pods in a rolling update fashion (gradually scale down the old ReplicaSets and scale up the new one) when `.spec.strategy.type==RollingUpdate`. You can specify `maxUnavailable` and `maxSurge` to control the rolling update process.

#### Max Unavailable

`.spec.strategy.rollingUpdate.maxUnavailable` is an optional field that specifies the maximum number of Pods that can be unavailable during the update process. The value can be an absolute number (for example, 5) or a percentage of desired Pods (for example, 10%). The absolute number is calculated from percentage by rounding down. The value cannot be 0 if `.spec.strategy.rollingUpdate.maxSurge` is 0. The default value is 25%.

For example, when this value is set to 30%, the old ReplicaSet can be scaled down to 70% of desired Pods immediately when the rolling update starts. Once new Pods are ready, old ReplicaSet can be scaled down further, followed by scaling up the new ReplicaSet, ensuring that the total number of Pods available at all times during the update is at least 70% of the desired Pods.

#### Max Surge

`.spec.strategy.rollingUpdate.maxSurge` is an optional field that specifies the maximum number of Pods that can be created over the desired number of Pods. The value can be an absolute number (for example, 5) or a percentage of desired Pods (for example, 10%). The value cannot be 0 if `maxUnavailable` is 0. The absolute number is calculated from the percentage by rounding up. The default value is 25%.

For example, when this value is set to 30%, the new ReplicaSet can be scaled up immediately when the rolling update starts, such that the total number of old and new Pods does not exceed 130% of desired Pods. Once old Pods have been killed, the new ReplicaSet can be scaled up further, ensuring that the total number of Pods running at any time during the update is at most 130% of desired Pods.

Here are some Rolling Update Deployment examples that use the `maxUnavailable` and `maxSurge`:

- Max Unavailable
- Max Surge
- Hybrid

```
apiVersion: apps/v1
kind: Deploymentmetadata: name: nginx-deployment labels:   app: nginxspec: replicas: 3 selector:   matchLabels:     app: nginx tem
```

```
apiVersion: apps/v1
kind: Deploymentmetadata: name: nginx-deployment labels:   app: nginxspec: replicas: 3 selector:   matchLabels:     app: nginx tem
```

```
apiVersion: apps/v1
kind: Deploymentmetadata: name: nginx-deployment labels:   app: nginxspec: replicas: 3 selector:   matchLabels:     app: nginx tem
```

### Progress Deadline Seconds

`.spec.progressDeadlineSeconds` is an optional field that specifies the number of seconds you want to wait for your Deployment to progress before the system reports back that the Deployment has failed progressing - surfaced as a condition with `type: Progressing`, `status: "False"`. and `reason: ProgressDeadlineExceeded` in the status of the resource. The Deployment controller will keep retrying the Deployment. This defaults to 600. In the future, once automatic rollback will be implemented, the Deployment controller will roll back a Deployment as soon as it observes such a condition.

If specified, this field needs to be greater than `.spec.minReadySeconds`.

### Min Ready Seconds

`.spec.minReadySeconds` is an optional field that specifies the minimum number of seconds for which a newly created Pod should be ready without any of its containers crashing, for it to be considered available. This defaults to 0 (the Pod will be considered available as soon as it is ready). To learn more about when a Pod is considered ready, see Container Probes.

### Terminating Pods

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

You can enable this feature by setting the `DeploymentReplicaSetTerminatingReplicas` feature gate on the API server and on the kube-controller-manager

Pods that become terminating due to deletion or scale down may take a long time to terminate, and may consume additional resources during that period. As a result, the total number of all pods can temporarily exceed `.spec.replicas`. Terminating pods can be tracked using the `.status.terminatingReplicas` field of the Deployment.

### Revision History Limit

A Deployment's revision history is stored in the ReplicaSets it controls.

`.spec.revisionHistoryLimit` is an optional field that specifies the number of old ReplicaSets to retain to allow rollback. These old ReplicaSets consume resources in `etcd` and crowd the output of `kubectl get rs`. The configuration of each Deployment revision is stored in its ReplicaSets; therefore, once an old ReplicaSet is deleted, you lose the ability to rollback to that revision of Deployment. By default, 10 old ReplicaSets will be kept, however its ideal value depends on the frequency and stability of new Deployments.

More specifically, setting this field to zero means that all old ReplicaSets with 0 replicas will be cleaned up. In this case, a new Deployment rollout cannot be undone, since its revision history is cleaned up.

### Paused

`.spec.paused` is an optional boolean field for pausing and resuming a Deployment. The only difference between a paused Deployment and one that is not paused, is that any changes into the PodTemplateSpec of the paused Deployment will not trigger new rollouts as long as it is paused. A Deployment is not paused by default when it is created.

## What's next

- Learn more about Pods.
- Run a stateless application using a Deployment.
- Read the Deployment to understand the Deployment API.
- Read about PodDisruptionBudget and how you can use it to manage application availability during disruptions.
- Use kubectl to create a Deployment.

# Pod Lifecycle

This page describes the lifecycle of a Pod. Pods follow a defined lifecycle, starting in the `Pending` phase, moving through `Running` if at least one of its primary containers starts OK, and then through either the `Succeeded` or `Failed` phases depending on whether any container in the Pod terminated in failure.

Like individual application containers, Pods are considered to be relatively ephemeral (rather than durable) entities. Pods are created, assigned a unique ID (UID), and scheduled to run on nodes where they remain until termination (according to restart policy) or deletion. If a Node dies, the Pods running on (or scheduled to run on) that node are marked for deletion. The control plane marks the Pods for removal after a timeout period.

## Pod lifetime

Whilst a Pod is running, the kubelet is able to restart containers to handle some kind of faults. Within a Pod, Kubernetes tracks different container states and determines what action to take to make the Pod healthy again.

In the Kubernetes API, Pods have both a specification and an actual status. The status for a Pod object consists of a set of Pod conditions. You can also inject custom readiness information into the condition data for a Pod, if that is useful to your application.

Pods are only scheduled once in their lifetime; assigning a Pod to a specific node is called *binding*, and the process of selecting which node to use is called *scheduling*. Once a Pod has been scheduled and is bound to a node, Kubernetes tries to run that Pod on the node. The Pod runs on that node until it stops, or until the Pod is terminated; if Kubernetes isn't able to start the Pod on the selected node (for example, if the node crashes before the Pod starts), then that particular Pod never starts.

You can use Pod Scheduling Readiness to delay scheduling for a Pod until all its *scheduling gates* are removed. For example, you might want to define a set of Pods but only trigger scheduling once all the Pods have been created.

### Pods and fault recovery

If one of the containers in the Pod fails, then Kubernetes may try to restart that specific container. Read How Pods handle problems with containers to learn more.

Pods can however fail in a way that the cluster cannot recover from, and in that case Kubernetes does not attempt to heal the Pod further; instead, Kubernetes deletes the Pod and relies on other components to provide automatic healing.

If a Pod is scheduled to a node and that node then fails, the Pod is treated as unhealthy and Kubernetes eventually deletes the Pod. A Pod won't survive an eviction due to a lack of resources or Node maintenance.

Kubernetes uses a higher-level abstraction, called a controller, that handles the work of managing the relatively disposable Pod instances.

A given Pod (as defined by a UID) is never "rescheduled" to a different node; instead, that Pod can be replaced by a new, near-identical Pod. If you make a replacement Pod, it can even have same name (as in `.metadata.name`) that the old Pod had, but the replacement would have a different `.metadata.uid` from the old Pod.

Kubernetes does not guarantee that a replacement for an existing Pod would be scheduled to the same node as the old Pod that was being replaced.

### Associated lifetimes

When something is said to have the same lifetime as a Pod, such as a volume, that means that the thing exists as long as that specific Pod (with that exact UID) exists. If that Pod is deleted for any reason, and even if an identical replacement is created, the related thing (a volume, in this example) is also destroyed and created anew.

> A multi-container Pod that contains a file puller sidecar and a web server. The Pod uses an ephemeral emptyDir volume for shared storage between the containers.
>
> **Figure 1.**
>
> A multi-container Pod that contains a file puller sidecar and a web server. The Pod uses an ephemeral `emptyDir` volume for shared storage between the containers.

## Pod phase

A Pod's `status` field is a PodStatus object, which has a `phase` field.

The phase of a Pod is a simple, high-level summary of where the Pod is in its lifecycle. The phase is not intended to be a comprehensive rollup of observations of container or Pod state, nor is it intended to be a comprehensive state machine.

The number and meanings of Pod phase values are tightly guarded. Other than what is documented here, nothing should be assumed about Pods that have a given `phase` value.

Here are the possible values for `phase`:

| Value | Description |
|---|---|
| `Pending` | The Pod has been accepted by the Kubernetes cluster, but one or more of the containers has not been set up and made ready to run. This includes time a Pod spends waiting to be scheduled as well as the time spent downloading container images over the network. |
| `Running` | The Pod has been bound to a node, and all of the containers have been created. At least one container is still running, or is in the process of starting or restarting. |
| `Succeeded` | All containers in the Pod have terminated in success, and will not be restarted. |
| `Failed` | All containers in the Pod have terminated, and at least one container has terminated in failure. That is, the container either exited with non-zero status or was terminated by the system, and is not set for automatic restarting. |
| `Unknown` | For some reason the state of the Pod could not be obtained. This phase typically occurs due to an error in communicating with the node where the Pod should be running. |

**Note:**

When a pod is failing to start repeatedly, `CrashLoopBackOff` may appear in the `Status` field of some kubectl commands. Similarly, when a pod is being deleted, `Terminating` may appear in the `Status` field of some kubectl commands.

Make sure not to confuse *Status*, a kubectl display field for user intuition, with the pod's `phase`. Pod phase is an explicit part of the Kubernetes data model and of the Pod API.

```
NAMESPACE               NAME               READY   STATUS             RESTARTS   AGE
alessandras-namespace   alessandras-pod    0/1     CrashLoopBackOff   200        2d9h
```

A Pod is granted a term to terminate gracefully, which defaults to 30 seconds. You can use the flag `--force` to terminate a Pod by force.

Since Kubernetes 1.27, the kubelet transitions deleted Pods, except for static Pods and force-deleted Pods without a finalizer, to a terminal phase (`Failed` or `Succeeded` depending on the exit statuses of the pod containers) before their deletion from the API server.

If a node dies or is disconnected from the rest of the cluster, Kubernetes applies a policy for setting the `phase` of all Pods on the lost node to Failed.

## Container states

As well as the phase of the Pod overall, Kubernetes tracks the state of each container inside a Pod. You can use container lifecycle hooks to trigger events to run at certain points in a container's lifecycle.

Once the scheduler assigns a Pod to a Node, the kubelet starts creating containers for that Pod using a container runtime. There are three possible container states: `Waiting`, `Running`, and `Terminated`.

To check the state of a Pod's containers, you can use `kubectl describe pod <name-of-pod>`. The output shows the state for each container within that Pod.

Each state has a specific meaning:

### `Waiting`

If a container is not in either the `Running` or `Terminated` state, it is `Waiting`. A container in the `Waiting` state is still running the operations it requires in order to complete start up: for example, pulling the container image from a container image registry, or applying Secret data. When you use `kubectl` to query a Pod with a container that is `Waiting`, you also see a Reason field to summarize why the container is in that state.

### `Running`

The `Running` status indicates that a container is executing without issues. If there was a `postStart` hook configured, it has already executed and finished. When you use `kubectl` to query a Pod with a container that is `Running`, you also see information about when the container entered the `Running` state.

### `Terminated`

A container in the `Terminated` state began execution and then either ran to completion or failed for some reason. When you use `kubectl` to query a Pod with a container that is `Terminated`, you see a reason, an exit code, and the start and finish time for that container's period of execution.

If a container has a `preStop` hook configured, this hook runs before the container enters the `Terminated` state.

## How Pods handle problems with containers

Kubernetes manages container failures within Pods using a `restartPolicy` defined in the Pod `spec`. This policy determines how Kubernetes reacts to containers exiting due to errors or other reasons, which falls in the following sequence:

1. **Initial crash**: Kubernetes attempts an immediate restart based on the Pod `restartPolicy`.
2. **Repeated crashes**: After the initial crash Kubernetes applies an exponential backoff delay for subsequent restarts, described in `restartPolicy`. This prevents rapid, repeated restart attempts from overloading the system.
3. **CrashLoopBackOff state**: This indicates that the backoff delay mechanism is currently in effect for a given container that is in a crash loop, failing and restarting repeatedly.
4. **Backoff reset**: If a container runs successfully for a certain duration (e.g., 10 minutes), Kubernetes resets the backoff delay, treating any new crash as the first one.

In practice, a `CrashLoopBackOff` is a condition or event that might be seen as output from the `kubectl` command, while describing or listing Pods, when a container in the Pod fails to start properly and then continually tries and fails in a loop.

In other words, when a container enters the crash loop, Kubernetes applies the exponential backoff delay mentioned in the Container restart policy. This mechanism prevents a faulty container from overwhelming the system with continuous failed start attempts.

The `CrashLoopBackOff` can be caused by issues like the following:

- Application errors that cause the container to exit.
- Configuration errors, such as incorrect environment variables or missing configuration files.
- Resource constraints, where the container might not have enough memory or CPU to start properly.
- Health checks failing if the application doesn't start serving within the expected time.
- Container liveness probes or startup probes returning a `Failure` result as mentioned in the probes section.

To investigate the root cause of a `CrashLoopBackOff` issue, a user can:

1. **Check logs**: Use `kubectl logs <name-of-pod>` to check the logs of the container. This is often the most direct way to diagnose the issue causing the crashes.
2. **Inspect events**: Use `kubectl describe pod <name-of-pod>` to see events for the Pod, which can provide hints about configuration or resource issues.
3. **Review configuration**: Ensure that the Pod configuration, including environment variables and mounted volumes, is correct and that all required external resources are available.
4. **Check resource limits**: Make sure that the container has enough CPU and memory allocated. Sometimes, increasing the resources in the Pod definition can resolve the issue.
5. **Debug application**: There might exist bugs or misconfigurations in the application code. Running this container image locally or in a development environment can help diagnose application specific issues.

## Container restarts

When a container in your Pod stops, or experiences failure, Kubernetes can restart it. A restart isn't always appropriate; for example, init containers run only once, during Pod startup.

You can configure restarts as a policy that applies to all Pods, or using container-level configuration (for example: when you define a sidecar container).

### Container restarts and resilience

The Kubernetes project recommends following cloud-native principles, including resilient design that accounts for unannounced or arbitrary restarts. You can achieve this either by failing the Pod and relying on automatic replacement, or you can design for container-level resilience. Either approach helps to ensure that your overall workload remains available despite partial failure.

### Pod-level container restart policy

The `spec` of a Pod has a `restartPolicy` field with possible values Always, OnFailure, and Never. The default value is Always.

The `restartPolicy` for a Pod applies to app containers in the Pod and to regular init containers. Sidecar containers ignore the Pod-level `restartPolicy` field: in Kubernetes, a sidecar is defined as an entry inside `initContainers` that has its container-level `restartPolicy` set to `Always`. For init containers that exit with an error, the kubelet restarts the init container if the Pod level `restartPolicy` is either `OnFailure` or `Always`:

- `Always`: Automatically restarts the container after any termination.
- `OnFailure`: Only restarts the container if it exits with an error (non-zero exit status).
- `Never`: Does not automatically restart the terminated container.

When the kubelet is handling container restarts according to the configured restart policy, that only applies to restarts that make replacement containers inside the same Pod and running on the same node. After containers in a Pod exit, the kubelet restarts them with an exponential backoff delay (10s, 20s, 40s, …), that is capped at 300 seconds (5 minutes). Once a container has executed for 10 minutes without any problems, the kubelet resets the restart backoff timer for that container. Sidecar containers and Pod lifecycle explains the behaviour of `init containers` when specify `restartpolicy` field on it.

### Individual container restart policy and rules

FEATURE STATE: `Kubernetes v1.34 [alpha]` (enabled by default: false)

If your cluster has the feature gate `ContainerRestartRules` enabled, you can specify `restartPolicy` and `restartPolicyRules` on *individual containers* to override the Pod restart policy. Container restart policy and rules applies to app containers in the Pod and to regular init containers.

A Kubernetes-native sidecar container has its container-level `restartPolicy` set to `Always`, and does not support `restartPolicyRules`.

The container restarts will follow the same exponential backoff as pod restart policy described above. Supported container restart policies:

- `Always`: Automatically restarts the container after any termination.
- `OnFailure`: Only restarts the container if it exits with an error (non-zero exit status).
- `Never`: Does not automatically restart the terminated container.

Additionally, *individual containers* can specify `restartPolicyRules`. If the `restartPolicyRules` field is specified, then container `restartPolicy` **must** also be specified. The `restartPolicyRules` define a list of rules to apply on container exit. Each rule will consist of a condition and an action. The supported condition is `exitCodes`, which compares the exit code of the container with a list of given values. The supported action is `Restart`, which means the container will be restarted. The rules will be evaluated in order. On the first match, the action will be applied. If none of the rules' conditions matched, Kubernetes fallback to container's configured `restartPolicy`.

For example, a Pod with OnFailure restart policy that have a `try-once` container. This allows Pod to only restart certain containers:

```
apiVersion: v1
kind: Podmetadata:  name: on-failure-podspec:  restartPolicy: OnFailure  containers:  - name: try-once-container    # This contain
```

A Pod with Always restart policy with an init container that only execute once. If the init container fails, the Pod fails. This allows the Pod to fail if the initialization failed, but also keep running once the initialization succeeds:

```
apiVersion: v1
kind: Podmetadata:  name: fail-pod-if-init-failsspec:  restartPolicy: Always  initContainers:  - name: init-once      # This init
```

A Pod with Never restart policy with a container that ignores and restarts on specific exit codes. This is useful to differentiate between restartable errors and non-restartable errors:

```
apiVersion: v1
kind: Podmetadata:  name: restart-on-exit-codesspec:  restartPolicy: Never  containers:  - name: restart-on-exit-codes    image: d
```

Restart rules can be used for many more advanced lifecycle management scenarios. Note, restart rules are affected by the same inconsistencies as the regular restart policy. Kubelet restarts, container runtime garbage collection, intermitted connectivity issues with the control plane may cause the state loss and containers may be re-run even when you expect a container not to be restarted.

### Reduced container restart delay

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

With the alpha feature gate `ReduceDefaultCrashLoopBackOffDecay` enabled, container start retries across your cluster will be reduced to begin at 1s (instead of 10s) and increase exponentially by 2x each restart until a maximum delay of 60s (instead of 300s which is 5 minutes).

If you use this feature along with the alpha feature `KubeletCrashLoopBackOffMax` (described below), individual nodes may have different maximum delays.

### Configurable container restart delay

FEATURE STATE: `Kubernetes v1.32 [alpha]` (enabled by default: false)

With the alpha feature gate `KubeletCrashLoopBackOffMax` enabled, you can reconfigure the maximum delay between container start retries from the default of 300s (5 minutes). This configuration is set per node using kubelet configuration. In your kubelet configuration, under `crashLoopBackOff` set the `maxContainerRestartPeriod` field between `"1s"` and `"300s"`. As described above in Container restart policy, delays on that node will still start at 10s and increase exponentially by 2x each restart, but will now be capped at your configured maximum. If the `maxContainerRestartPeriod` you configure is less than the default initial value of 10s, the initial delay will instead be set to the configured maximum.

See the following kubelet configuration examples:

```
# container restart delays will start at 10s, increasing
# 2x each time they are restarted, to a maximum of 100skind: KubeletConfigurationcrashLoopBackOff:    maxContainerRestartPeriod: "
```

```
# delays between container restarts will always be 2s
kind: KubeletConfigurationcrashLoopBackOff:    maxContainerRestartPeriod: "2s"
```

If you use this feature along with the alpha feature `ReduceDefaultCrashLoopBackOffDecay` (described above), your cluster defaults for initial backoff and maximum backoff will no longer be 10s and 300s, but 1s and 60s. Per node configuration takes precedence over the defaults set by `ReduceDefaultCrashLoopBackOffDecay`, even if this would result in a node having a longer maximum backoff than other nodes in the cluster.

## Pod conditions

A Pod has a PodStatus, which has an array of PodConditions through which the Pod has or has not passed. Kubelet manages the following PodConditions:

- `PodScheduled`: the Pod has been scheduled to a node.
- `PodReadyToStartContainers`: (beta feature; enabled by default) the Pod sandbox has been successfully created and networking configured.
- `ContainersReady`: all containers in the Pod are ready.
- `Initialized`: all init containers have completed successfully.
- `Ready`: the Pod is able to serve requests and should be added to the load balancing pools of all matching Services.
- `DisruptionTarget`: the pod is about to be terminated due to a disruption (such as preemption, eviction or garbage-collection).
- `PodResizePending`: a pod resize was requested but cannot be applied. See Pod resize status.
- `PodResizeInProgress`: the pod is in the process of resizing. See Pod resize status.

| Field name | Description |
|---|---|
| `type` | Name of this Pod condition. |
| `status` | Indicates whether that condition is applicable, with possible values "`True`", "`False`", or "`Unknown`". |
| `lastProbeTime` | Timestamp of when the Pod condition was last probed. |
| `lastTransitionTime` | Timestamp for when the Pod last transitioned from one status to another. |
| `reason` | Machine-readable, UpperCamelCase text indicating the reason for the condition's last transition. |
| `message` | Human-readable message indicating details about the last status transition. |

### Pod readiness

FEATURE STATE: `Kubernetes v1.14 [stable]`

Your application can inject extra feedback or signals into PodStatus: *Pod readiness*. To use this, set `readinessGates` in the Pod's `spec` to specify a list of additional conditions that the kubelet evaluates for Pod readiness.

Readiness gates are determined by the current state of `status.condition` fields for the Pod. If Kubernetes cannot find such a condition in the `status.conditions` field of a Pod, the status of the condition is defaulted to "`False`".

Here is an example:

```
kind: Pod
...spec:  readinessGates:    - conditionType: "www.example.com/feature-1"status:  conditions:    - type: Ready
```

The Pod conditions you add must have names that meet the Kubernetes label key format.

### Status for Pod readiness

The `kubectl patch` command does not support patching object status. To set these `status.conditions` for the Pod, applications and operators should use the `PATCH` action. You can use a Kubernetes client library to write code that sets custom Pod conditions for Pod readiness.

For a Pod that uses custom conditions, that Pod is evaluated to be ready **only** when both the following statements apply:

- All containers in the Pod are ready.
- All conditions specified in `readinessGates` are `True`.

When a Pod's containers are Ready but at least one custom condition is missing or `False`, the kubelet sets the Pod's condition to `ContainersReady`.

### Pod network readiness

FEATURE STATE: `Kubernetes v1.29 [beta]`

**Note:**

During its early development, this condition was named `PodHasNetwork`.

After a Pod gets scheduled on a node, it needs to be admitted by the kubelet and to have any required storage volumes mounted. Once these phases are complete, the kubelet works with a container runtime (using Container Runtime Interface (CRI)) to set up a runtime sandbox and configure networking for the Pod. If the `PodReadyToStartContainersCondition` feature gate is enabled (it is enabled by default for Kubernetes 1.34), the `PodReadyToStartContainers` condition will be added to the `status.conditions` field of a Pod.

The `PodReadyToStartContainers` condition is set to `False` by the Kubelet when it detects a Pod does not have a runtime sandbox with networking configured. This occurs in the following scenarios:

- Early in the lifecycle of the Pod, when the kubelet has not yet begun to set up a sandbox for the Pod using the container runtime.
- Later in the lifecycle of the Pod, when the Pod sandbox has been destroyed due to either:
  - the node rebooting, without the Pod getting evicted
  - for container runtimes that use virtual machines for isolation, the Pod sandbox virtual machine rebooting, which then requires creating a new sandbox and fresh container network configuration.

The `PodReadyToStartContainers` condition is set to `True` by the kubelet after the successful completion of sandbox creation and network configuration for the Pod by the runtime plugin. The kubelet can start pulling container images and create containers after `PodReadyToStartContainers` condition has been set to `True`.

For a Pod with init containers, the kubelet sets the `Initialized` condition to `True` after the init containers have successfully completed (which happens after successful sandbox creation and network configuration by the runtime plugin). For a Pod without init containers, the kubelet sets the `Initialized` condition to `True` before sandbox creation and network configuration starts.

## Container probes

A *probe* is a diagnostic performed periodically by the kubelet on a container. To perform a diagnostic, the kubelet either executes code within the container, or makes a network request.

### Check mechanisms

There are four different ways to check a container using a probe. Each probe must define exactly one of these four mechanisms:

`exec`
: Executes a specified command inside the container. The diagnostic is considered successful if the command exits with a status code of 0.

`grpc`
: Performs a remote procedure call using gRPC. The target should implement gRPC health checks. The diagnostic is considered successful if the `status` of the response is `SERVING`.

`httpGet`
: Performs an HTTP `GET` request against the Pod's IP address on a specified port and path. The diagnostic is considered successful if the response has a status code greater than or equal to 200 and less than 400.

`tcpSocket`
: Performs a TCP check against the Pod's IP address on a specified port. The diagnostic is considered successful if the port is open. If the remote system (the container) closes the connection immediately after it opens, this counts as healthy.

**Caution:**

Unlike the other mechanisms, `exec` probe's implementation involves the creation/forking of multiple processes each time when executed. As a result, in case of the clusters having higher pod densities, lower intervals of `initialDelaySeconds`, `periodSeconds`, configuring any probe with exec mechanism might introduce an overhead on the cpu usage of the node. In such scenarios, consider using the alternative probe mechanisms to avoid the overhead.

### Probe outcome

Each probe has one of three results:

`Success`
: The container passed the diagnostic.

`Failure`
: The container failed the diagnostic.

`Unknown`
: The diagnostic failed (no action should be taken, and the kubelet will make further checks).

### Types of probe

The kubelet can optionally perform and react to three kinds of probes on running containers:

`livenessProbe`
: Indicates whether the container is running. If the liveness probe fails, the kubelet kills the container, and the container is subjected to its restart policy. If a container does not provide a liveness probe, the default state is `Success`.

`readinessProbe`
: Indicates whether the container is ready to respond to requests. If the readiness probe fails, the EndpointSlice controller removes the Pod's IP address from the EndpointSlices of all Services that match the Pod. The default state of readiness before the initial delay is `Failure`. If a container does not provide a readiness probe, the default state is `Success`.

`startupProbe`
: Indicates whether the application within the container is started. All other probes are disabled if a startup probe is provided, until it succeeds. If the startup probe fails, the kubelet kills the container, and the container is subjected to its restart policy. If a container does not provide a startup probe, the default state is `Success`.

For more information about how to set up a liveness, readiness, or startup probe, see Configure Liveness, Readiness and Startup Probes.

**When should you use a liveness probe?**

If the process in your container is able to crash on its own whenever it encounters an issue or becomes unhealthy, you do not necessarily need a liveness probe; the kubelet will automatically perform the correct action in accordance with the Pod's `restartPolicy`.

If you'd like your container to be killed and restarted if a probe fails, then specify a liveness probe, and specify a `restartPolicy` of Always or OnFailure.

**When should you use a readiness probe?**

If you'd like to start sending traffic to a Pod only when a probe succeeds, specify a readiness probe. In this case, the readiness probe might be the same as the liveness probe, but the existence of the readiness probe in the spec means that the Pod will start without receiving any traffic and only start receiving traffic after the probe starts succeeding.

If you want your container to be able to take itself down for maintenance, you can specify a readiness probe that checks an endpoint specific to readiness that is different from the liveness probe.

If your app has a strict dependency on back-end services, you can implement both a liveness and a readiness probe. The liveness probe passes when the app itself is healthy, but the readiness probe additionally checks that each required back-end service is available. This helps you avoid directing traffic to Pods that can only respond with error messages.

If your container needs to work on loading large data, configuration files, or migrations during startup, you can use a startup probe. However, if you want to detect the difference between an app that has failed and an app that is still processing its startup data, you might prefer a readiness probe.

**Note:**

If you want to be able to drain requests when the Pod is deleted, you do not necessarily need a readiness probe; when the Pod is deleted, the corresponding endpoint in the `EndpointSlice` will update its conditions: the endpoint `ready` condition will be set to `false`, so load balancers will not use the Pod for regular traffic. See Pod termination for more information about how the kubelet handles Pod deletion.

**When should you use a startup probe?**

Startup probes are useful for Pods that have containers that take a long time to come into service. Rather than set a long liveness interval, you can configure a separate configuration for probing the container as it starts up, allowing a time longer than the liveness interval would allow.

If your container usually starts in more than \( initialDelaySeconds + failureThreshold \times periodSeconds \), you should specify a startup probe that checks the same endpoint as the liveness probe. The default for `periodSeconds` is 10s. You should then set its `failureThreshold` high enough to allow the container to start, without changing the default values of the liveness probe. This helps to protect against deadlocks.

# Termination of Pods

Because Pods represent processes running on nodes in the cluster, it is important to allow those processes to gracefully terminate when they are no longer needed (rather than being abruptly stopped with a `KILL` signal and having no chance to clean up).

The design aim is for you to be able to request deletion and know when processes terminate, but also be able to ensure that deletes eventually complete. When you request deletion of a Pod, the cluster records and tracks the intended grace period before the Pod is allowed to be forcefully killed. With that forceful shutdown tracking in place, the kubelet attempts graceful shutdown.

Typically, with this graceful termination of the pod, kubelet makes requests to the container runtime to attempt to stop the containers in the pod by first sending a TERM (aka. SIGTERM) signal, with a grace period timeout, to the main process in each container. The requests to stop the containers are processed by the container runtime asynchronously. There is no guarantee to the order of processing for these requests. Many container runtimes respect the `STOPSIGNAL` value defined in the container image and, if different, send the container image configured STOPSIGNAL instead of TERM. Once the grace period has expired, the KILL signal is sent to any remaining processes, and the Pod is then deleted from the API Server. If the kubelet or the container runtime's management service is restarted while waiting for processes to terminate, the cluster retries from the start including the full original grace period.

## Stop Signals

The stop signal used to kill the container can be defined in the container image with the `STOPSIGNAL` instruction. If no stop signal is defined in the image, the default signal of the container runtime (SIGTERM for both containerd and CRI-O) would be used to kill the container.

## Defining custom stop signals

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

If the `ContainerStopSignals` feature gate is enabled, you can configure a custom stop signal for your containers from the container Lifecycle. We require the Pod's `spec.os.name` field to be present as a requirement for defining stop signals in the container lifecycle. The list of signals that are valid depends on the OS the Pod is scheduled to. For Pods scheduled to Windows nodes, we only support SIGTERM and SIGKILL as valid signals.

Here is an example Pod spec defining a custom stop signal:

```yaml
spec:
  os:
    name: linux
  containers:
    - name: my-container
      image: container-image:latest
      lifecycle:
        stopSignal: SIGUSR1
```

If a stop signal is defined in the lifecycle, this will override the signal defined in the container image. If no stop signal is defined in the container spec, the container would fall back to the default behavior.

### Pod Termination Flow

Pod termination flow, illustrated with an example:

1. You use the `kubectl` tool to manually delete a specific Pod, with the default grace period (30 seconds).

2. The Pod in the API server is updated with the time beyond which the Pod is considered "dead" along with the grace period. If you use `kubectl describe` to check the Pod you're deleting, that Pod shows up as "Terminating". On the node where the Pod is running: as soon as the kubelet sees that a Pod has been marked as terminating (a graceful shutdown duration has been set), the kubelet begins the local Pod shutdown process.

    1. If one of the Pod's containers has defined a `preStop` hook and the `terminationGracePeriodSeconds` in the Pod spec is not set to 0, the kubelet runs that hook inside of the container. The default `terminationGracePeriodSeconds` setting is 30 seconds.

        If the `preStop` hook is still running after the grace period expires, the kubelet requests a small, one-off grace period extension of 2 seconds.

        **Note:**

        If the `preStop` hook needs longer to complete than the default grace period allows, you must modify `terminationGracePeriodSeconds` to suit this.

    2. The kubelet triggers the container runtime to send a TERM signal to process 1 inside each container.

        There is special ordering if the Pod has any sidecar containers defined. Otherwise, the containers in the Pod receive the TERM signal at different times and in an arbitrary order. If the order of shutdowns matters, consider using a `preStop` hook to synchronize (or switch to using sidecar containers).

3. At the same time as the kubelet is starting graceful shutdown of the Pod, the control plane evaluates whether to remove that shutting-down Pod from EndpointSlice objects, where those objects represent a Service with a configured selector. ReplicaSets and other workload resources no longer treat the shutting-down Pod as a valid, in-service replica.

    Pods that shut down slowly should not continue to serve regular traffic and should start terminating and finish processing open connections. Some applications need to go beyond finishing open connections and need more graceful termination, for example, session draining and completion.

    Any endpoints that represent the terminating Pods are not immediately removed from EndpointSlices, and a status indicating terminating state is exposed from the EndpointSlice API. Terminating endpoints always have their `ready` status as `false` (for backward compatibility with versions before 1.26), so load balancers will not use it for regular traffic.

    If traffic draining on terminating Pod is needed, the actual readiness can be checked as a condition `serving`. You can find more details on how to implement connections draining in the tutorial Pods And Endpoints Termination Flow

4. The kubelet ensures the Pod is shut down and terminated

    1. When the grace period expires, if there is still any container running in the Pod, the kubelet triggers forcible shutdown. The container runtime sends `SIGKILL` to any processes still running in any container in the Pod. The kubelet also cleans up a hidden `pause` container if that container runtime uses one.
    2. The kubelet transitions the Pod into a terminal phase (`Failed` or `Succeeded` depending on the end state of its containers).
    3. The kubelet triggers forcible removal of the Pod object from the API server, by setting grace period to 0 (immediate deletion).
    4. The API server deletes the Pod's API object, which is then no longer visible from any client.

### Forced Pod termination

**Caution:**

Forced deletions can be potentially disruptive for some workloads and their Pods.

By default, all deletes are graceful within 30 seconds. The `kubectl delete` command supports the `--grace-period=<seconds>` option which allows you to override the default and specify your own value.

Setting the grace period to `0` forcibly and immediately deletes the Pod from the API server. If the Pod was still running on a node, that forcible deletion triggers the kubelet to begin immediate cleanup.

Using kubectl, You must specify an additional flag `--force` along with `--grace-period=0` in order to perform force deletions.

When a force deletion is performed, the API server does not wait for confirmation from the kubelet that the Pod has been terminated on the node it was running on. It removes the Pod in the API immediately so a new Pod can be created with the same name. On the node, Pods that are set to terminate immediately will still be given a small grace period before being force killed.

**Caution:**

Immediate deletion does not wait for confirmation that the running resource has been terminated. The resource may continue to run on the cluster indefinitely.

If you need to force-delete Pods that are part of a StatefulSet, refer to the task documentation for deleting Pods from a StatefulSet.

### Pod shutdown and sidecar containers

If your Pod includes one or more sidecar containers (init containers with an Always restart policy), the kubelet will delay sending the TERM signal to these sidecar containers until the last main container has fully terminated. The sidecar containers will be terminated in the reverse order they are defined in the Pod spec. This ensures that sidecar containers continue serving the other containers in the Pod until they are no longer needed.

This means that slow termination of a main container will also delay the termination of the sidecar containers. If the grace period expires before the termination process is complete, the Pod may enter forced termination. In this case, all remaining containers in the Pod will be terminated simultaneously with a short grace period.

Similarly, if the Pod has a `preStop` hook that exceeds the termination grace period, emergency termination may occur. In general, if you have used `preStop` hooks to control the termination order without sidecar containers, you can now remove them and allow the kubelet to manage sidecar termination automatically.

### Garbage collection of Pods

For failed Pods, the API objects remain in the cluster's API until a human or controller process explicitly removes them.

The Pod garbage collector (PodGC), which is a controller in the control plane, cleans up terminated Pods (with a phase of `Succeeded` or `Failed`), when the number of Pods exceeds the configured threshold (determined by `terminated-pod-gc-threshold` in the kube-controller-manager). This avoids a resource leak as Pods are created and terminated over time.

Additionally, PodGC cleans up any Pods which satisfy any of the following conditions:

1. are orphan Pods - bound to a node which no longer exists,
2. are unscheduled terminating Pods,
3. are terminating Pods, bound to a non-ready node tainted with `node.kubernetes.io/out-of-service`.

Along with cleaning up the Pods, PodGC will also mark them as failed if they are in a non-terminal phase. Also, PodGC adds a Pod disruption condition when cleaning up an orphan Pod. See Pod disruption conditions for more details.

## What's next

- Get hands-on experience attaching handlers to container lifecycle events.
- Get hands-on experience configuring Liveness, Readiness and Startup Probes.
- Learn more about container lifecycle hooks.
- Learn more about sidecar containers.
- For detailed information about Pod and container status in the API, see the API reference documentation covering `status` for Pod.

# ReplicationController

Legacy API for managing workloads that can scale horizontally. Superseded by the Deployment and ReplicaSet APIs.

**Note:**

A `Deployment` that configures a `ReplicaSet` is now the recommended way to set up replication.

A *ReplicationController* ensures that a specified number of pod replicas are running at any one time. In other words, a ReplicationController makes sure that a pod or a homogeneous set of pods is always up and available.

## How a ReplicationController works

If there are too many pods, the ReplicationController terminates the extra pods. If there are too few, the ReplicationController starts more pods. Unlike manually created pods, the pods maintained by a ReplicationController are automatically replaced if they fail, are deleted, or are terminated. For example, your pods are re-created on a node after disruptive maintenance such as a kernel upgrade. For this reason, you should use a ReplicationController even if your application requires only a single pod. A ReplicationController is similar to a process supervisor, but instead of supervising individual processes on a single node, the ReplicationController supervises multiple pods across multiple nodes.

ReplicationController is often abbreviated to "rc" in discussion, and as a shortcut in kubectl commands.

A simple case is to create one ReplicationController object to reliably run one instance of a Pod indefinitely. A more complex use case is to run several identical replicas of a replicated service, such as web servers.

## Running an example ReplicationController

This example ReplicationController config runs three copies of the nginx web server.

controllers/replication.yaml Copy controllers/replication.yaml to clipboard

```
apiVersion: v1
kind: ReplicationControllermetadata:  name: nginxspec:  replicas: 3  selector:    app: nginx  template:    metadata:      name: ng
```

Run the example job by downloading the example file and then running this command:

```
kubectl apply -f https://k8s.io/examples/controllers/replication.yaml
```

The output is similar to this:

```
replicationcontroller/nginx created
```

Check on the status of the ReplicationController using this command:

```
kubectl describe replicationcontrollers/nginx
```

The output is similar to this:

```
Name:        nginx
Namespace:   default
Selector:    app=nginx
Labels:      app=nginx
Annotations:    <none>
Replicas:    3 current / 3 desired
Pods Status: 0 Running / 3 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:       app=nginx
  Containers:
   nginx:
    Image:              nginx
    Port:               80/TCP
    Environment:        <none>
    Mounts:             <none>
  Volumes:              <none>
Events:
  FirstSeen       LastSeen     Count    From                          SubobjectPath    Type      Reason              Message
  ---------       --------     -----    ----                          -------------    ----      ------              -------
  20s             20s          1        {replication-controller }                      Normal    SuccessfulCreate    Created pod: ng
  20s             20s          1        {replication-controller }                      Normal    SuccessfulCreate    Created pod: ng
  20s             20s          1        {replication-controller }                      Normal    SuccessfulCreate    Created pod: ng
```

Here, three pods are created, but none is running yet, perhaps because the image is being pulled. A little later, the same command may show:

```
Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed
```

To list all the pods that belong to the ReplicationController in a machine readable form, you can use a command like this:

```
pods=$(kubectl get pods --selector=app=nginx --output=jsonpath={.items..metadata.name})
echo $pods
```

The output is similar to this:

```
nginx-3ntk0 nginx-4ok8v nginx-qrm3m
```

Here, the selector is the same as the selector for the ReplicationController (seen in the `kubectl describe` output), and in a different form in `replication.yaml`. The `--output=jsonpath` option specifies an expression with the name from each pod in the returned list.

## Writing a ReplicationController Manifest

As with all other Kubernetes config, a ReplicationController needs `apiVersion`, `kind`, and `metadata` fields.

When the control plane creates new Pods for a ReplicationController, the `.metadata.name` of the ReplicationController is part of the basis for naming those Pods. The name of a ReplicationController must be a valid DNS subdomain value, but this can produce unexpected results for the Pod hostnames. For best compatibility, the name should follow the more restrictive rules for a DNS label.

For general information about working with configuration files, see object management.

A ReplicationController also needs a `.spec` section.

### Pod Template

The `.spec.template` is the only required field of the `.spec`.

The `.spec.template` is a pod template. It has exactly the same schema as a Pod, except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a pod template in a ReplicationController must specify appropriate labels and an appropriate restart policy. For labels, make sure not to overlap with other controllers. See pod selector.

Only a `.spec.template.spec.restartPolicy` equal to `Always` is allowed, which is the default if not specified.

For local container restarts, ReplicationControllers delegate to an agent on the node, for example the Kubelet.

### Labels on the ReplicationController

The ReplicationController can itself have labels (`.metadata.labels`). Typically, you would set these the same as the `.spec.template.metadata.labels`; if `.metadata.labels` is not specified then it defaults to `.spec.template.metadata.labels`. However, they are allowed to be different, and the `.metadata.labels` do not affect the behavior of the ReplicationController.

### Pod Selector

The `.spec.selector` field is a label selector. A ReplicationController manages all the pods with labels that match the selector. It does not distinguish between pods that it created or deleted and pods that another person or process created or deleted. This allows the ReplicationController to be replaced without affecting the running pods.

If specified, the `.spec.template.metadata.labels` must be equal to the `.spec.selector`, or it will be rejected by the API. If `.spec.selector` is unspecified, it will be defaulted to `.spec.template.metadata.labels`.

Also you should not normally create any pods whose labels match this selector, either directly, with another ReplicationController, or with another controller such as Job. If you do so, the ReplicationController thinks that it created the other pods. Kubernetes does not stop you from doing this.

If you do end up with multiple controllers that have overlapping selectors, you will have to manage the deletion yourself (see below).

### Multiple Replicas

You can specify how many pods should run concurrently by setting `.spec.replicas` to the number of pods you would like to have running concurrently. The number running at any time may be higher or lower, such as if the replicas were just increased or decreased, or if a pod is gracefully shutdown, and a replacement starts early.

If you do not specify `.spec.replicas`, then it defaults to 1.

## Working with ReplicationControllers

### Deleting a ReplicationController and its Pods

To delete a ReplicationController and all its pods, use `kubectl delete`. Kubectl will scale the ReplicationController to zero and wait for it to delete each pod before deleting the ReplicationController itself. If this kubectl command is interrupted, it can be restarted.

When using the REST API or client library, you need to do the steps explicitly (scale replicas to 0, wait for pod deletions, then delete the ReplicationController).

### Deleting only a ReplicationController

You can delete a ReplicationController without affecting any of its pods.

Using kubectl, specify the `--cascade=orphan` option to `kubectl delete`.

When using the REST API or client library, you can delete the ReplicationController object.

Once the original is deleted, you can create a new ReplicationController to replace it. As long as the old and new `.spec.selector` are the same, then the new one will adopt the old pods. However, it will not make any effort to make existing pods match a new, different pod template. To update pods to a new spec in a controlled way, use a rolling update.

### Isolating pods from a ReplicationController

Pods may be removed from a ReplicationController's target set by changing their labels. This technique may be used to remove pods from service for debugging and data recovery. Pods that are removed in this way will be replaced automatically (assuming that the number of replicas is not also changed).

## Common usage patterns

### Rescheduling

As mentioned above, whether you have 1 pod you want to keep running, or 1000, a ReplicationController will ensure that the specified number of pods exists, even in the event of node failure or pod termination (for example, due to an action by another control agent).

### Scaling

The ReplicationController enables scaling the number of replicas up or down, either manually or by an auto-scaling control agent, by updating the `replicas` field.

### Rolling updates

The ReplicationController is designed to facilitate rolling updates to a service by replacing pods one-by-one.

As explained in #1353, the recommended approach is to create a new ReplicationController with 1 replica, scale the new (+1) and old (-1) controllers one by one, and then delete the old controller after it reaches 0 replicas. This predictably updates the set of pods regardless of unexpected failures.

Ideally, the rolling update controller would take application readiness into account, and would ensure that a sufficient number of pods were productively serving at any given time.

The two ReplicationControllers would need to create pods with at least one differentiating label, such as the image tag of the primary container of the pod, since it is typically image updates that motivate rolling updates.

### Multiple release tracks

In addition to running multiple releases of an application while a rolling update is in progress, it's common to run multiple releases for an extended period of time, or even continuously, using multiple release tracks. The tracks would be differentiated by labels.

For instance, a service might target all pods with `tier in (frontend), environment in (prod)`. Now say you have 10 replicated pods that make up this tier. But you want to be able to 'canary' a new version of this component. You could set up a ReplicationController with `replicas` set to 9 for the bulk of the replicas, with labels `tier=frontend, environment=prod, track=stable`, and another ReplicationController with `replicas` set to 1 for the canary, with labels `tier=frontend, environment=prod, track=canary`. Now the service is covering both the canary and non-canary pods. But you can mess with the ReplicationControllers separately to test things out, monitor the results, etc.

### Using ReplicationControllers with Services

Multiple ReplicationControllers can sit behind a single service, so that, for example, some traffic goes to the old version, and some goes to the new version.

A ReplicationController will never terminate on its own, but it isn't expected to be as long-lived as services. Services may be composed of pods controlled by multiple ReplicationControllers, and it is expected that many ReplicationControllers may be created and destroyed over the lifetime of a service (for instance, to perform an update of pods that run the service). Both services themselves and their clients should remain oblivious to the ReplicationControllers that maintain the pods of the services.

## Writing programs for Replication

Pods created by a ReplicationController are intended to be fungible and semantically identical, though their configurations may become heterogeneous over time. This is an obvious fit for replicated stateless servers, but ReplicationControllers can also be used to maintain availability of master-elected, sharded, and worker-pool applications. Such applications should use dynamic work assignment mechanisms, such as the RabbitMQ work queues, as opposed to static/one-time customization of the configuration of each pod, which is considered an anti-pattern. Any pod customization performed, such as vertical auto-sizing of resources (for example, cpu or memory), should be performed by another online controller process, not unlike the ReplicationController itself.

## Responsibilities of the ReplicationController

The ReplicationController ensures that the desired number of pods matches its label selector and are operational. Currently, only terminated pods are excluded from its count. In the future, readiness and other information available from the system may be taken into account, we may add more controls over the replacement policy, and we plan to emit events that could be used by external clients to implement arbitrarily sophisticated replacement and/or scale-down policies.

The ReplicationController is forever constrained to this narrow responsibility. It itself will not perform readiness nor liveness probes. Rather than performing auto-scaling, it is intended to be controlled by an external auto-scaler (as discussed in #492), which would change its `replicas` field. We will not add scheduling policies (for example, spreading) to the ReplicationController. Nor should it verify that the pods controlled match the currently specified template, as that would obstruct auto-sizing and other automated processes. Similarly, completion deadlines, ordering dependencies, configuration expansion, and other features belong elsewhere. We even plan to factor out the mechanism for bulk pod creation (#170).

The ReplicationController is intended to be a composable building-block primitive. We expect higher-level APIs and/or tools to be built on top of it and other complementary primitives for user convenience in the future. The "macro" operations currently supported by kubectl (run, scale) are proof-of-concept examples of this. For instance, we could imagine something like Asgard managing ReplicationControllers, auto-scalers, services, scheduling policies, canaries, etc.

## API Object

Replication controller is a top-level resource in the Kubernetes REST API. More details about the API object can be found at: ReplicationController API object.

## Alternatives to ReplicationController

### ReplicaSet

`ReplicaSet` is the next-generation ReplicationController that supports the new set-based label selector. It's mainly used by Deployment as a mechanism to orchestrate pod creation, deletion and updates. Note that we recommend using Deployments instead of directly using Replica Sets, unless you require custom update orchestration or don't require updates at all.

### Deployment (Recommended)

`Deployment` is a higher-level API object that updates its underlying Replica Sets and their Pods. Deployments are recommended if you want the rolling update functionality, because they are declarative, server-side, and have additional features.

### Bare Pods

Unlike in the case where a user directly created pods, a ReplicationController replaces pods that are deleted or terminated for any reason, such as in the case of node failure or disruptive node maintenance, such as a kernel upgrade. For this reason, we recommend that you use a ReplicationController even if your application requires only a single pod. Think of it similarly to a process supervisor, only it supervises multiple pods across multiple nodes instead of individual processes on a single node. A ReplicationController delegates local container restarts to some agent on the node, such as the kubelet.

### Job

Use a `Job` instead of a ReplicationController for pods that are expected to terminate on their own (that is, batch jobs).

### DaemonSet

Use a `DaemonSet` instead of a ReplicationController for pods that provide a machine-level function, such as machine monitoring or machine logging. These pods have a lifetime that is tied to a machine lifetime: the pod needs to be running on the machine before other pods start, and are safe to terminate when the machine is otherwise ready to be rebooted/shutdown.

## What's next

- Learn about Pods.
- Learn about Deployment, the replacement for ReplicationController.
- `ReplicationController` is part of the Kubernetes REST API. Read the ReplicationController object definition to understand the API for replication controllers.

# Jobs

Jobs represent one-off tasks that run to completion and then stop.

A Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete. Deleting a Job will clean up the Pods it created. Suspending a Job will delete its active Pods until the Job is resumed again.

A simple case is to create one Job object in order to reliably run one Pod to completion. The Job object will start a new Pod if the first Pod fails or is deleted (for example due to a node hardware failure or a node reboot).

You can also use a Job to run multiple Pods in parallel.

If you want to run a Job (either a single task, or several in parallel) on a schedule, see CronJob.

## Running an example Job

Here is an example Job config. It computes π to 2000 places and prints it out. It takes around 10s to complete.

`controllers/job.yaml` Copy controllers/job.yaml to clipboard

```
apiVersion: batch/v1
kind: Jobmetadata:  name: pispec:  template:    spec:      containers:      - name: pi        image: perl:5.34.0        command: [
```

You can run the example with this command:

```
kubectl apply -f https://kubernetes.io/examples/controllers/job.yaml
```

The output is similar to this:

```
job.batch/pi created
```

Check on the status of the Job with `kubectl`:

- kubectl describe job pi
- kubectl get job pi -o yaml

```
Name:           pi
Namespace:      default
Selector:       batch.kubernetes.io/controller-uid=c9948307-e56d-4b5d-8302-ae2d7b7da67c
Labels:         batch.kubernetes.io/controller-uid=c9948307-e56d-4b5d-8302-ae2d7b7da67c
                batch.kubernetes.io/job-name=pi
                ...
Annotations:    batch.kubernetes.io/job-tracking: ""
Parallelism:    1
Completions:    1
Start Time:     Mon, 02 Dec 2019 15:20:11 +0200
Completed At:   Mon, 02 Dec 2019 15:21:16 +0200
Duration:       65s
Pods Statuses:  0 Running / 1 Succeeded / 0 Failed
Pod Template:
  Labels:  batch.kubernetes.io/controller-uid=c9948307-e56d-4b5d-8302-ae2d7b7da67c
           batch.kubernetes.io/job-name=pi
  Containers:
   pi:
    Image:      perl:5.34.0
    Port:       <none>
    Host Port:  <none>
    Command:
      perl
      -Mbignum=bpi
      -wle
      print bpi(2000)
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From            Message
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  21s   job-controller  Created pod: pi-xf9p4
  Normal  Completed         18s   job-controller  Job completed
```

```
apiVersion: batch/v1
```

```
kind: Job
metadata:
  annotations: batch.kubernetes.io/job-tracking: ""
             ...
  creationTimestamp: "2022-11-10T17:53:53Z"
  generation: 1
  labels:
    batch.kubernetes.io/controller-uid: 863452e6-270d-420e-9b94-53a54146c223
    batch.kubernetes.io/job-name: pi
  name: pi
  namespace: default
  resourceVersion: "4751"
  uid: 204fb678-040b-497f-9266-35ffa8716d14
spec:
  backoffLimit: 4
  completionMode: NonIndexed
  completions: 1
  parallelism: 1
  selector:
    matchLabels:
      batch.kubernetes.io/controller-uid: 863452e6-270d-420e-9b94-53a54146c223
  suspend: false
  template:
    metadata:
      creationTimestamp: null
      labels:
        batch.kubernetes.io/controller-uid: 863452e6-270d-420e-9b94-53a54146c223
        batch.kubernetes.io/job-name: pi
    spec:
      containers:
      - command:
        - perl
        - -Mbignum=bpi
        - -wle
        - print bpi(2000)
        image: perl:5.34.0
        imagePullPolicy: IfNotPresent
        name: pi
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      restartPolicy: Never
      schedulerName: default-scheduler
      securityContext: {}
      terminationGracePeriodSeconds: 30
status:
  active: 1
  ready: 0
  startTime: "2022-11-10T17:53:57Z"
  uncountedTerminatedPods: {}
```

To view completed Pods of a Job, use `kubectl get pods`.

To list all the Pods that belong to a Job in a machine readable form, you can use a command like this:

```
pods=$(kubectl get pods --selector=batch.kubernetes.io/job-name=pi --output=jsonpath='{.items[*].metadata.name}')
echo $pods
```

The output is similar to this:

```
pi-5rwd7
```

Here, the selector is the same as the selector for the Job. The `--output=jsonpath` option specifies an expression with the name from each Pod in the returned list.

View the standard output of one of the pods:

```
kubectl logs $pods
```

Another way to view the logs of a Job:

```
kubectl logs jobs/pi
```

The output is similar to this:

```
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460
```

## Writing a Job spec

As with all other Kubernetes config, a Job needs `apiVersion`, `kind`, and `metadata` fields.

When the control plane creates new Pods for a Job, the `.metadata.name` of the Job is part of the basis for naming those Pods. The name of a Job must be a valid DNS subdomain value, but this can produce unexpected results for the Pod hostnames. For best compatibility, the name should follow the more restrictive rules for a DNS label. Even when the name is a DNS subdomain, the name must be no longer than 63 characters.

A Job also needs a `.spec` section.

### Job Labels

Job labels will have `batch.kubernetes.io/` prefix for `job-name` and `controller-uid`.

### Pod Template

The `.spec.template` is the only required field of the `.spec`.

The `.spec.template` is a pod template. It has exactly the same schema as a Pod, except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a pod template in a Job must specify appropriate labels (see pod selector) and an appropriate restart policy.

Only a `RestartPolicy` equal to `Never` or `OnFailure` is allowed.

### Pod selector

The `.spec.selector` field is optional. In almost all cases you should not specify it. See section specifying your own pod selector.

### Parallel execution for Jobs

There are three main types of task suitable to run as a Job:

1. Non-parallel Jobs
   - normally, only one Pod is started, unless the Pod fails.
   - the Job is complete as soon as its Pod terminates successfully.
2. Parallel Jobs with a *fixed completion count*:
   - specify a non-zero positive value for `.spec.completions`.
   - the Job represents the overall task, and is complete when there are `.spec.completions` successful Pods.
   - when using `.spec.completionMode="Indexed"`, each Pod gets a different index in the range 0 to `.spec.completions-1`.
3. Parallel Jobs with a *work queue*:
   - do not specify `.spec.completions`, default to `.spec.parallelism`.
   - the Pods must coordinate amongst themselves or an external service to determine what each should work on. For example, a Pod might fetch a batch of up to N items from the work queue.
   - each Pod is independently capable of determining whether or not all its peers are done, and thus that the entire Job is done.
   - when *any* Pod from the Job terminates with success, no new Pods are created.
   - once at least one Pod has terminated with success and all Pods are terminated, then the Job is completed with success.
   - once any Pod has exited with success, no other Pod should still be doing any work for this task or writing any output. They should all be in the process of exiting.

For a *non-parallel* Job, you can leave both `.spec.completions` and `.spec.parallelism` unset. When both are unset, both are defaulted to 1.

For a *fixed completion count* Job, you should set `.spec.completions` to the number of completions needed. You can set `.spec.parallelism`, or leave it unset and it will default to 1.

For a *work queue* Job, you must leave `.spec.completions` unset, and set `.spec.parallelism` to a non-negative integer.

For more information about how to make use of the different types of job, see the job patterns section.

#### Controlling parallelism

The requested parallelism (`.spec.parallelism`) can be set to any non-negative value. If it is unspecified, it defaults to 1. If it is specified as 0, then the Job is effectively paused until it is increased.

Actual parallelism (number of pods running at any instant) may be more or less than requested parallelism, for a variety of reasons:

- For *fixed completion count* Jobs, the actual number of pods running in parallel will not exceed the number of remaining completions. Higher values of `.spec.parallelism` are effectively ignored.
- For *work queue* Jobs, no new Pods are started after any Pod has succeeded -- remaining Pods are allowed to complete, however.
- If the Job Controller has not had time to react.
- If the Job controller failed to create Pods for any reason (lack of `ResourceQuota`, lack of permission, etc.), then there may be fewer pods than requested.
- The Job controller may throttle new Pod creation due to excessive previous pod failures in the same Job.
- When a Pod is gracefully shut down, it takes time to stop.

### Completion mode

FEATURE STATE: `Kubernetes v1.24 [stable]`

Jobs with *fixed completion count* - that is, jobs that have non null `.spec.completions` - can have a completion mode that is specified in `.spec.completionMode`:

- `NonIndexed` (default): the Job is considered complete when there have been `.spec.completions` successfully completed Pods. In other words, each Pod completion is homologous to each other. Note that Jobs that have null `.spec.completions` are implicitly `NonIndexed`.

- `Indexed`: the Pods of a Job get an associated completion index from 0 to `.spec.completions-1`. The index is available through four mechanisms:

  - The Pod annotation `batch.kubernetes.io/job-completion-index`.
  - The Pod label `batch.kubernetes.io/job-completion-index` (for v1.28 and later). Note the feature gate `PodIndexLabel` must be enabled to use this label, and it is enabled by default.
  - As part of the Pod hostname, following the pattern `$(job-name)-$(index)`. When you use an Indexed Job in combination with a Service, Pods within the Job can use the deterministic hostnames to address each other via DNS. For more information about how to configure this, see Job with Pod-to-Pod Communication.
  - From the containerized task, in the environment variable `JOB_COMPLETION_INDEX`.

The Job is considered complete when there is one successfully completed Pod for each index. For more information about how to use this mode, see Indexed Job for Parallel Processing with Static Work Assignment.

**Note:**

Although rare, more than one Pod could be started for the same index (due to various reasons such as node failures, kubelet restarts, or Pod evictions). In this case, only the first Pod that completes successfully will count towards the completion count and update the status of the Job. The other Pods that are running or completed for the same index will be deleted by the Job controller once they are detected.

## Handling Pod and container failures

A container in a Pod may fail for a number of reasons, such as because the process in it exited with a non-zero exit code, or the container was killed for exceeding a memory limit, etc. If this happens, and the `.spec.template.spec.restartPolicy = "OnFailure"`, then the Pod stays on the node, but the container is re-run. Therefore, your program needs to handle the case when it is restarted locally, or else specify `.spec.template.spec.restartPolicy = "Never"`. See pod lifecycle for more information on `restartPolicy`.

An entire Pod can also fail, for a number of reasons, such as when the pod is kicked off the node (node is upgraded, rebooted, deleted, etc.), or if a container of the Pod fails and the `.spec.template.spec.restartPolicy = "Never"`. When a Pod fails, then the Job controller starts a new Pod. This means that your application needs to handle the case when it is restarted in a new pod. In particular, it needs to handle temporary files, locks, incomplete output and the like caused by previous runs.

By default, each pod failure is counted towards the `.spec.backoffLimit` limit, see pod backoff failure policy. However, you can customize handling of pod failures by setting the Job's pod failure policy.

Additionally, you can choose to count the pod failures independently for each index of an Indexed Job by setting the `.spec.backoffLimitPerIndex` field (for more information, see backoff limit per index).

Note that even if you specify `.spec.parallelism = 1` and `.spec.completions = 1` and `.spec.template.spec.restartPolicy = "Never"`, the same program may sometimes be started twice.

If you do specify `.spec.parallelism` and `.spec.completions` both greater than 1, then there may be multiple pods running at once. Therefore, your pods must also be tolerant of concurrency.

If you specify the `.spec.podFailurePolicy` field, the Job controller does not consider a terminating Pod (a pod that has a `.metadata.deletionTimestamp` field set) as a failure until that Pod is terminal (its `.status.phase` is `Failed` or `Succeeded`). However, the Job controller creates a replacement Pod as soon as the termination becomes apparent. Once the pod terminates, the Job controller evaluates `.backoffLimit` and `.podFailurePolicy` for the relevant Job, taking this now-terminated Pod into consideration.

If either of these requirements is not satisfied, the Job controller counts a terminating Pod as an immediate failure, even if that Pod later terminates with `phase: "Succeeded"`.

### Pod backoff failure policy

There are situations where you want to fail a Job after some amount of retries due to a logical error in configuration etc. To do so, set `.spec.backoffLimit` to specify the number of retries before considering a Job as failed.

The `.spec.backoffLimit` is set by default to 6, unless the backoff limit per index (only Indexed Job) is specified. When `.spec.backoffLimitPerIndex` is specified, then `.spec.backoffLimit` defaults to 2147483647 (MaxInt32).

Failed Pods associated with the Job are recreated by the Job controller with an exponential back-off delay (10s, 20s, 40s ...) capped at six minutes.

The number of retries is calculated in two ways:

- The number of Pods with `.status.phase = "Failed"`.
- When using `restartPolicy = "OnFailure"`, the number of retries in all the containers of Pods with `.status.phase` equal to `Pending` or `Running`.

If either of the calculations reaches the `.spec.backoffLimit`, the Job is considered failed.

**Note:**

If your Job has `restartPolicy = "OnFailure"`, keep in mind that your Pod running the job will be terminated once the job backoff limit has been reached. This can make debugging the Job's executable more difficult. We suggest setting `restartPolicy = "Never"` when debugging the Job or using a logging system to ensure output from failed Jobs is not lost inadvertently.

### Backoff limit per index

FEATURE STATE: `Kubernetes v1.33 [stable]` (enabled by default: true)

When you run an indexed Job, you can choose to handle retries for pod failures independently for each index. To do so, set the `.spec.backoffLimitPerIndex` to specify the maximal number of pod failures per index.

When the per-index backoff limit is exceeded for an index, Kubernetes considers the index as failed and adds it to the `.status.failedIndexes` field. The succeeded indexes, those with a successfully executed pods, are recorded in the `.status.completedIndexes` field, regardless of whether you set the `backoffLimitPerIndex` field.

Note that a failing index does not interrupt execution of other indexes. Once all indexes finish for a Job where you specified a backoff limit per index, if at least one of those indexes did fail, the Job controller marks the overall Job as failed, by setting the Failed condition in the status. The Job gets marked as failed even if some, potentially nearly all, of the indexes were processed successfully.

You can additionally limit the maximal number of indexes marked failed by setting the `.spec.maxFailedIndexes` field. When the number of failed indexes exceeds the `maxFailedIndexes` field, the Job controller triggers termination of all remaining running Pods for that Job. Once all pods are terminated, the entire Job is marked failed by the Job controller, by setting the Failed condition in the Job status.

Here is an example manifest for a Job that defines a `backoffLimitPerIndex`:

[/controllers/job-backoff-limit-per-index-example.yaml](/controllers/job-backoff-limit-per-index-example.yaml) Copy /controllers/job-backoff-limit-per-index-example.yaml to clipboard

```
apiVersion: batch/v1
kind: Jobmetadata:  name: job-backoff-limit-per-index-examplespec:  completions: 10  parallelism: 3  completionMode: Indexed  # re
```

In the example above, the Job controller allows for one restart for each of the indexes. When the total number of failed indexes exceeds 5, then the entire Job is terminated.

Once the job is finished, the Job status looks as follows:

```
kubectl get -o yaml job job-backoff-limit-per-index-example
```

```
  status:
    completedIndexes: 1,3,5,7,9
    failedIndexes: 0,2,4,6,8
    succeeded: 5          # 1 succeeded pod for each of 5 succeeded indexes
    failed: 10            # 2 failed pods (1 retry) for each of 5 failed indexes
    conditions:
    - message: Job has failed indexes
      reason: FailedIndexes
      status: "True"
      type: FailureTarget
    - message: Job has failed indexes
      reason: FailedIndexes
      status: "True"
      type: Failed
```

The Job controller adds the `FailureTarget` Job condition to trigger Job termination and cleanup. When all of the Job Pods are terminated, the Job controller adds the `Failed` condition with the same values for `reason` and `message` as the `FailureTarget` Job condition. For details, see Termination of Job Pods.

Additionally, you may want to use the per-index backoff along with a pod failure policy. When using per-index backoff, there is a new `FailIndex` action available which allows you to avoid unnecessary retries within an index.

## Pod failure policy

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

A Pod failure policy, defined with the `.spec.podFailurePolicy` field, enables your cluster to handle Pod failures based on the container exit codes and the Pod conditions.

In some situations, you may want to have a better control when handling Pod failures than the control provided by the Pod backoff failure policy, which is based on the Job's `.spec.backoffLimit`. These are some examples of use cases:

- To optimize costs of running workloads by avoiding unnecessary Pod restarts, you can terminate a Job as soon as one of its Pods fails with an exit code indicating a software bug.
- To guarantee that your Job finishes even if there are disruptions, you can ignore Pod failures caused by disruptions (such as preemption, API-initiated eviction or taint-based eviction) so that they don't count towards the `.spec.backoffLimit` limit of retries.

You can configure a Pod failure policy, in the `.spec.podFailurePolicy` field, to meet the above use cases. This policy can handle Pod failures based on the container exit codes and the Pod conditions.

Here is a manifest for a Job that defines a `podFailurePolicy`:

[/controllers/job-pod-failure-policy-example.yaml](/controllers/job-pod-failure-policy-example.yaml) Copy /controllers/job-pod-failure-policy-example.yaml to clipboard

```
apiVersion: batch/v1
kind: Jobmetadata:  name: job-pod-failure-policy-examplespec:  completions: 12  parallelism: 3  template:    spec:      restartPol
```

In the example above, the first rule of the Pod failure policy specifies that the Job should be marked failed if the `main` container fails with the 42 exit code. The following are the rules for the `main` container specifically:

- an exit code of 0 means that the container succeeded
- an exit code of 42 means that the **entire Job** failed
- any other exit code represents that the container failed, and hence the entire Pod. The Pod will be re-created if the total number of restarts is below `backoffLimit`. If the `backoffLimit` is reached the **entire Job** failed.

**Note:**

Because the Pod template specifies a `restartPolicy: Never`, the kubelet does not restart the `main` container in that particular Pod.

The second rule of the Pod failure policy, specifying the `Ignore` action for failed Pods with condition `DisruptionTarget` excludes Pod disruptions from being counted towards the `.spec.backoffLimit` limit of retries.

**Note:**

If the Job failed, either by the Pod failure policy or Pod backoff failure policy, and the Job is running multiple Pods, Kubernetes terminates all the Pods in that Job that are still Pending or Running.

These are some requirements and semantics of the API:

- if you want to use a `.spec.podFailurePolicy` field for a Job, you must also define that Job's pod template with `.spec.restartPolicy` set to `Never`.
- the Pod failure policy rules you specify under `spec.podFailurePolicy.rules` are evaluated in order. Once a rule matches a Pod failure, the remaining rules are ignored. When no rule matches the Pod failure, the default handling applies.
- you may want to restrict a rule to a specific container by specifying its name in`spec.podFailurePolicy.rules[*].onExitCodes.containerName`. When not specified the rule applies to all containers. When specified, it should match one the container or `initContainer` names in the Pod template.
- you may specify the action taken when a Pod failure policy is matched by `spec.podFailurePolicy.rules[*].action`. Possible values are:
    - `FailJob`: use to indicate that the Pod's job should be marked as Failed and all running Pods should be terminated.
    - `Ignore`: use to indicate that the counter towards the `.spec.backoffLimit` should not be incremented and a replacement Pod should be created.
    - `Count`: use to indicate that the Pod should be handled in the default way. The counter towards the `.spec.backoffLimit` should be incremented.
    - `FailIndex`: use this action along with backoff limit per index to avoid unnecessary retries within the index of a failed pod.

**Note:**

When you use a `podFailurePolicy`, the job controller only matches Pods in the `Failed` phase. Pods with a deletion timestamp that are not in a terminal phase (`Failed` or `Succeeded`) are considered still terminating. This implies that terminating pods retain a tracking finalizer until they reach a terminal phase. Since Kubernetes 1.27, Kubelet transitions deleted pods to a terminal phase (see: Pod Phase). This ensures that deleted pods have their finalizers removed by the Job controller.

**Note:**

Starting with Kubernetes v1.28, when Pod failure policy is used, the Job controller recreates terminating Pods only once these Pods reach the terminal `Failed` phase. This behavior is similar to `podReplacementPolicy: Failed`. For more information, see Pod replacement policy.

When you use the `podFailurePolicy`, and the Job fails due to the pod matching the rule with the `FailJob` action, then the Job controller triggers the Job termination process by adding the `FailureTarget` condition. For more details, see Job termination and cleanup.

## Success policy

When creating an Indexed Job, you can define when a Job can be declared as succeeded using a `.spec.successPolicy`, based on the pods that succeeded.

By default, a Job succeeds when the number of succeeded Pods equals `.spec.completions`. These are some situations where you might want additional control for declaring a Job succeeded:

- When running simulations with different parameters, you might not need all the simulations to succeed for the overall Job to be successful.
- When following a leader-worker pattern, only the success of the leader determines the success or failure of a Job. Examples of this are frameworks like MPI and PyTorch etc.

You can configure a success policy, in the `.spec.successPolicy` field, to meet the above use cases. This policy can handle Job success based on the succeeded pods. After the Job meets the success policy, the job controller terminates the lingering Pods. A success policy is defined by rules. Each rule can take one of the following forms:

- When you specify the `succeededIndexes` only, once all indexes specified in the `succeededIndexes` succeed, the job controller marks the Job as succeeded. The `succeededIndexes` must be a list of intervals between 0 and `.spec.completions-1`.
- When you specify the `succeededCount` only, once the number of succeeded indexes reaches the `succeededCount`, the job controller marks the Job as succeeded.
- When you specify both `succeededIndexes` and `succeededCount`, once the number of succeeded indexes from the subset of indexes specified in the `succeededIndexes` reaches the `succeededCount`, the job controller marks the Job as succeeded.

Note that when you specify multiple rules in the `.spec.successPolicy.rules`, the job controller evaluates the rules in order. Once the Job meets a rule, the job controller ignores remaining rules.

Here is a manifest for a Job with `successPolicy`:

`/controllers/job-success-policy.yaml` Copy /controllers/job-success-policy.yaml to clipboard

```
apiVersion: batch/v1
kind: Jobmetadata:  name: job-successspec:  parallelism: 10  completions: 10  completionMode: Indexed # Required for the success p
```

In the example above, both `succeededIndexes` and `succeededCount` have been specified. Therefore, the job controller will mark the Job as succeeded and terminate the lingering Pods when either of the specified indexes, 0, 2, or 3, succeed. The Job that meets the success policy gets the `SuccessCriteriaMet` condition with a `SuccessPolicy` reason. After the removal of the lingering Pods is issued, the Job gets the `Complete` condition.

Note that the `succeededIndexes` is represented as intervals separated by a hyphen. The number are listed in represented by the first and last element of the series, separated by a hyphen.

**Note:**

When you specify both a success policy and some terminating policies such as `.spec.backoffLimit` and `.spec.podFailurePolicy`, once the Job meets either policy, the job controller respects the terminating policy and ignores the success policy.

## Job termination and cleanup

When a Job completes, no more Pods are created, but the Pods are usually not deleted either. Keeping them around allows you to still view the logs of completed pods to check for errors, warnings, or other diagnostic output. The job object also remains after it is completed so that you can view its status. It is up to the user to delete old jobs after noting their status. Delete the job with `kubectl` (e.g. `kubectl delete jobs/pi` or `kubectl delete -f ./job.yaml`). When you delete the job using `kubectl`, all the pods it created are deleted too.

By default, a Job will run uninterrupted unless a Pod fails (`restartPolicy=Never`) or a Container exits in error (`restartPolicy=OnFailure`), at which point the Job defers to the `.spec.backoffLimit` described above. Once `.spec.backoffLimit` has been reached the Job will be marked as failed and any running Pods will be terminated.

Another way to terminate a Job is by setting an active deadline. Do this by setting the `.spec.activeDeadlineSeconds` field of the Job to a number of seconds. The `activeDeadlineSeconds` applies to the duration of the job, no matter how many Pods are created. Once a Job reaches `activeDeadlineSeconds`, all of its running Pods are terminated and the Job status will become `type: Failed` with `reason: DeadlineExceeded`.

Note that a Job's `.spec.activeDeadlineSeconds` takes precedence over its `.spec.backoffLimit`. Therefore, a Job that is retrying one or more failed Pods will not deploy additional Pods once it reaches the time limit specified by `activeDeadlineSeconds`, even if the `backoffLimit` is not yet reached.

Example:

```
apiVersion: batch/v1
kind: Jobmetadata:  name: pi-with-timeoutspec:  backoffLimit: 5  activeDeadlineSeconds: 100  template:    spec:      containers:
```

Note that both the Job spec and the Pod template spec within the Job have an `activeDeadlineSeconds` field. Ensure that you set this field at the proper level.

Keep in mind that the `restartPolicy` applies to the Pod, and not to the Job itself: there is no automatic Job restart once the Job status is `type: Failed`. That is, the Job termination mechanisms activated with `.spec.activeDeadlineSeconds` and `.spec.backoffLimit` result in a permanent Job failure that requires manual intervention to resolve.

### Terminal Job conditions

A Job has two possible terminal states, each of which has a corresponding Job condition:

- Succeeded: Job condition `Complete`
- Failed: Job condition `Failed`

Jobs fail for the following reasons:

- The number of Pod failures exceeded the specified `.spec.backoffLimit` in the Job specification. For details, see Pod backoff failure policy.
- The Job runtime exceeded the specified `.spec.activeDeadlineSeconds`
- An indexed Job that used `.spec.backoffLimitPerIndex` has failed indexes. For details, see Backoff limit per index.
- The number of failed indexes in the Job exceeded the specified `spec.maxFailedIndexes`. For details, see Backoff limit per index
- A failed Pod matches a rule in `.spec.podFailurePolicy` that has the `FailJob` action. For details about how Pod failure policy rules might affect failure evaluation, see Pod failure policy.

Jobs succeed for the following reasons:

- The number of succeeded Pods reached the specified `.spec.completions`
- The criteria specified in `.spec.successPolicy` are met. For details, see Success policy.

In Kubernetes v1.31 and later the Job controller delays the addition of the terminal conditions,`Failed` or `Complete`, until all of the Job Pods are terminated.

In Kubernetes v1.30 and earlier, the Job controller added the `Complete` or the `Failed` Job terminal conditions as soon as the Job termination process was triggered and all Pod finalizers were removed. However, some Pods would still be running or terminating at the moment that the terminal condition was added.

In Kubernetes v1.31 and later, the controller only adds the Job terminal conditions *after* all of the Pods are terminated. You can control this behavior by using the `JobManagedBy` and the `JobPodReplacementPolicy` (both enabled by default) feature gates.

### Termination of Job pods

The Job controller adds the `FailureTarget` condition or the `SuccessCriteriaMet` condition to the Job to trigger Pod termination after a Job meets either the success or failure criteria.

Factors like `terminationGracePeriodSeconds` might increase the amount of time from the moment that the Job controller adds the `FailureTarget` condition or the `SuccessCriteriaMet` condition to the moment that all of the Job Pods terminate and the Job controller adds a terminal condition (`Failed` or `Complete`).

You can use the `FailureTarget` or the `SuccessCriteriaMet` condition to evaluate whether the Job has failed or succeeded without having to wait for the controller to add a terminal condition.

For example, you might want to decide when to create a replacement Job that replaces a failed Job. If you replace the failed Job when the `FailureTarget` condition appears, your replacement Job runs sooner, but could result in Pods from the failed and the replacement Job running at the same time, using extra compute resources.

Alternatively, if your cluster has limited resource capacity, you could choose to wait until the `Failed` condition appears on the Job, which would delay your replacement Job but would ensure that you conserve resources by waiting until all of the failed Pods are removed.

## Clean up finished jobs automatically

Finished Jobs are usually no longer needed in the system. Keeping them around in the system will put pressure on the API server. If the Jobs are managed directly by a higher level controller, such as CronJobs, the Jobs can be cleaned up by CronJobs based on the specified capacity-based cleanup policy.

### TTL mechanism for finished Jobs

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

Another way to clean up finished Jobs (either `Complete` or `Failed`) automatically is to use a TTL mechanism provided by a TTL controller for finished resources, by specifying the `.spec.ttlSecondsAfterFinished` field of the Job.

When the TTL controller cleans up the Job, it will delete the Job cascadingly, i.e. delete its dependent objects, such as Pods, together with the Job. Note that when the Job is deleted, its lifecycle guarantees, such as finalizers, will be honored.

For example:

```
apiVersion: batch/v1
kind: Jobmetadata:  name: pi-with-ttlspec:  ttlSecondsAfterFinished: 100  template:    spec:      containers:      - name: pi
```

The Job `pi-with-ttl` will be eligible to be automatically deleted, `100` seconds after it finishes.

If the field is set to `0`, the Job will be eligible to be automatically deleted immediately after it finishes. If the field is unset, this Job won't be cleaned up by the TTL controller after it finishes.

**Note:**

It is recommended to set `ttlSecondsAfterFinished` field because unmanaged jobs (Jobs that you created directly, and not indirectly through other workload APIs such as CronJob) have a default deletion policy of `orphanDependents` causing Pods created by an unmanaged Job to be left around after that Job is fully deleted. Even though the control plane eventually garbage collects the Pods from a deleted Job after they either fail or complete, sometimes those lingering pods may cause cluster performance degradation or in worst case cause the cluster to go offline due to this degradation.

You can use LimitRanges and ResourceQuotas to place a cap on the amount of resources that a particular namespace can consume.

# Job patterns

The Job object can be used to process a set of independent but related *work items*. These might be emails to be sent, frames to be rendered, files to be transcoded, ranges of keys in a NoSQL database to scan, and so on.

In a complex system, there may be multiple different sets of work items. Here we are just considering one set of work items that the user wants to manage together — a *batch job*.

There are several different patterns for parallel computation, each with strengths and weaknesses. The tradeoffs are:

- One Job object for each work item, versus a single Job object for all work items. One Job per work item creates some overhead for the user and for the system to manage large numbers of Job objects. A single Job for all work items is better for large numbers of items.
- Number of Pods created equals number of work items, versus each Pod can process multiple work items. When the number of Pods equals the number of work items, the Pods typically requires less modification to existing code and containers. Having each Pod process multiple work items is better for large numbers of items.
- Several approaches use a work queue. This requires running a queue service, and modifications to the existing program or container to make it use the work queue. Other approaches are easier to adapt to an existing containerised application.
- When the Job is associated with a headless Service, you can enable the Pods within a Job to communicate with each other to collaborate in a computation.

The tradeoffs are summarized here, with columns 2 to 4 corresponding to the above tradeoffs. The pattern names are also links to examples and more detailed description.

| Pattern | Single Job object | Fewer pods than work items? | Use app unmodified? |
|---|---|---|---|
| Queue with Pod Per Work Item | ✓ | | sometimes |
| Queue with Variable Pod Count | ✓ | ✓ | |
| Indexed Job with Static Work Assignment | ✓ | | ✓ |
| Job with Pod-to-Pod Communication | ✓ | sometimes | sometimes |
| Job Template Expansion | | | ✓ |

When you specify completions with `.spec.completions`, each Pod created by the Job controller has an identical `spec`. This means that all pods for a task will have the same command line and the same image, the same volumes, and (almost) the same environment variables. These patterns are different ways to arrange for pods to work on different things.

This table shows the required settings for `.spec.parallelism` and `.spec.completions` for each of the patterns. Here, `W` is the number of work items.

| Pattern | `.spec.completions` | `.spec.parallelism` |
|---|---|---|
| Queue with Pod Per Work Item | W | any |
| Queue with Variable Pod Count | null | any |
| Indexed Job with Static Work Assignment | W | any |
| Job with Pod-to-Pod Communication | W | W |
| Job Template Expansion | 1 | should be 1 |

# Advanced usage

## Suspending a Job

FEATURE STATE: `Kubernetes v1.24 [stable]`

When a Job is created, the Job controller will immediately begin creating Pods to satisfy the Job's requirements and will continue to do so until the Job is complete. However, you may want to temporarily suspend a Job's execution and resume it later, or start Jobs in suspended state and have a custom controller decide later when to start them.

To suspend a Job, you can update the `.spec.suspend` field of the Job to true; later, when you want to resume it again, update it to false. Creating a Job with `.spec.suspend` set to true will create it in the suspended state.

When a Job is resumed from suspension, its `.status.startTime` field will be reset to the current time. This means that the `.spec.activeDeadlineSeconds` timer will be stopped and reset when a Job is suspended and resumed.

When you suspend a Job, any running Pods that don't have a status of `Completed` will be terminated with a SIGTERM signal. The Pod's graceful termination period will be honored and your Pod must handle this signal in this period. This may involve saving progress for later or undoing changes. Pods terminated this way will not count towards the Job's `completions` count.

An example Job definition in the suspended state can be like so:

```
kubectl get job myjob -o yaml
```

```
apiVersion: batch/v1
kind: Jobmetadata:  name: myjobspec:  suspend: true  parallelism: 1  completions: 5  template:    spec:      ...
```

You can also toggle Job suspension by patching the Job using the command line.

Suspend an active Job:

```
kubectl patch job/myjob --type=strategic --patch '{"spec":{"suspend":true}}'
```

Resume a suspended Job:

```
kubectl patch job/myjob --type=strategic --patch '{"spec":{"suspend":false}}'
```

The Job's status can be used to determine if a Job is suspended or has been suspended in the past:

```
kubectl get jobs/myjob -o yaml
```

```
apiVersion: batch/v1
kind: Job# .metadata and .spec omittedstatus:  conditions:  - lastProbeTime: "2021-02-05T13:14:33Z"    lastTransitionTime: "2021-0
```

The Job condition of type "Suspended" with status "True" means the Job is suspended; the `lastTransitionTime` field can be used to determine how long the Job has been suspended for. If the status of that condition is "False", then the Job was previously suspended and is now running. If such a condition does not exist in the Job's status, the Job has never been stopped.

Events are also created when the Job is suspended and resumed:

```
kubectl describe jobs/myjob
```

```
Name:           myjob
...
Events:
  Type    Reason            Age   From            Message
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  12m   job-controller  Created pod: myjob-hlrpl
  Normal  SuccessfulDelete  11m   job-controller  Deleted pod: myjob-hlrpl
  Normal  Suspended         11m   job-controller  Job suspended
  Normal  SuccessfulCreate  3s    job-controller  Created pod: myjob-jvb44
  Normal  Resumed           3s    job-controller  Job resumed
```

The last four events, particularly the "Suspended" and "Resumed" events, are directly a result of toggling the `.spec.suspend` field. In the time between these two events, we see that no Pods were created, but Pod creation restarted as soon as the Job was resumed.

### Mutable Scheduling Directives

FEATURE STATE: `Kubernetes v1.27 [stable]`

In most cases, a parallel job will want the pods to run with constraints, like all in the same zone, or all either on GPU model x or y but not a mix of both.

The suspend field is the first step towards achieving those semantics. Suspend allows a custom queue controller to decide when a job should start; However, once a job is unsuspended, a custom queue controller has no influence on where the pods of a job will actually land.

This feature allows updating a Job's scheduling directives before it starts, which gives custom queue controllers the ability to influence pod placement while at the same time offloading actual pod-to-node assignment to kube-scheduler. This is allowed only for suspended Jobs that have never been unsuspended before.

The fields in a Job's pod template that can be updated are node affinity, node selector, tolerations, labels, annotations and scheduling gates.

### Specifying your own Pod selector

Normally, when you create a Job object, you do not specify `.spec.selector`. The system defaulting logic adds this field when the Job is created. It picks a selector value that will not overlap with any other jobs.

However, in some cases, you might need to override this automatically set selector. To do this, you can specify the `.spec.selector` of the Job.

Be very careful when doing this. If you specify a label selector which is not unique to the pods of that Job, and which matches unrelated Pods, then pods of the unrelated job may be deleted, or this Job may count other Pods as completing it, or one or both Jobs may refuse to create Pods or run to completion. If a non-unique selector is chosen, then other controllers (e.g. ReplicationController) and their Pods may behave in unpredictable ways too. Kubernetes will not stop you from making a mistake when specifying `.spec.selector`.

Here is an example of a case when you might want to use this feature.

Say Job `old` is already running. You want existing Pods to keep running, but you want the rest of the Pods it creates to use a different pod template and for the Job to have a new name. You cannot update the Job because these fields are not updatable. Therefore, you delete Job `old` but *leave its pods running*, using `kubectl delete jobs/old --cascade=orphan`. Before deleting it, you make a note of what selector it uses:

```
kubectl get job old -o yaml
```

The output is similar to this:

```
kind: Job
metadata:   name: old   ...spec:   selector:    matchLabels:      batch.kubernetes.io/controller-uid: a8f3d00d-c6d2-11e5-9f87-42010af
```

Then you create a new Job with name `new` and you explicitly specify the same selector. Since the existing Pods have label `batch.kubernetes.io/controller-uid=a8f3d00d-c6d2-11e5-9f87-42010af00002`, they are controlled by Job `new` as well.

You need to specify `manualSelector: true` in the new Job since you are not using the selector that the system normally generates for you automatically.

```
kind: Job
metadata:   name: new   ...spec:   manualSelector: true   selector:    matchLabels:      batch.kubernetes.io/controller-uid: a8f3d00d-
```

The new Job itself will have a different uid from `a8f3d00d-c6d2-11e5-9f87-42010af00002`. Setting `manualSelector: true` tells the system that you know what you are doing and to allow this mismatch.

### Job tracking with finalizers

FEATURE STATE: `Kubernetes v1.26 [stable]`

The control plane keeps track of the Pods that belong to any Job and notices if any such Pod is removed from the API server. To do that, the Job controller creates Pods with the finalizer `batch.kubernetes.io/job-tracking`. The controller removes the finalizer only after the Pod has been accounted for in the Job status, allowing the Pod to be removed by other controllers or users.

**Note:**

See My pod stays terminating if you observe that pods from a Job are stuck with the tracking finalizer.

### Elastic Indexed Jobs

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

You can scale Indexed Jobs up or down by mutating both `.spec.parallelism` and `.spec.completions` together such that `.spec.parallelism == .spec.completions`. When scaling down, Kubernetes removes the Pods with higher indexes.

Use cases for elastic Indexed Jobs include batch workloads which require scaling an indexed Job, such as MPI, Horovod, Ray, and PyTorch training jobs.

### Delayed creation of replacement pods

FEATURE STATE: `Kubernetes v1.34 [stable]` (enabled by default: true)

By default, the Job controller recreates Pods as soon they either fail or are terminating (have a deletion timestamp). This means that, at a given time, when some of the Pods are terminating, the number of running Pods for a Job can be greater than `parallelism` or greater than one Pod per index (if you are using an Indexed Job).

You may choose to create replacement Pods only when the terminating Pod is fully terminal (has `status.phase: Failed`). To do this, set the `.spec.podReplacementPolicy: Failed`. The default replacement policy depends on whether the Job has a `podFailurePolicy` set. With no Pod failure policy defined for a Job, omitting the `podReplacementPolicy` field selects the `TerminatingOrFailed` replacement policy: the control plane creates replacement Pods immediately upon Pod deletion (as soon as the control plane sees that a Pod for this Job has `deletionTimestamp` set). For Jobs with a Pod failure policy set, the default `podReplacementPolicy` is `Failed`, and no other value is permitted. See Pod failure policy to learn more about Pod failure policies for Jobs.

```
kind: Job
metadata:   name: new   ...spec:   podReplacementPolicy: Failed   ...
```

Provided your cluster has the feature gate enabled, you can inspect the `.status.terminating` field of a Job. The value of the field is the number of Pods owned by the Job that are currently terminating.

```
kubectl get jobs/myjob -o yaml
```

```
apiVersion: batch/v1
kind: Job# .metadata and .spec omittedstatus:  terminating: 3 # three Pods are terminating and have not yet reached the Failed pha
```

### Delegation of managing a Job object to external controller

FEATURE STATE: `Kubernetes v1.32 [beta]` (enabled by default: true)

**Note:**

You can only set the `managedBy` field on Jobs if you enable the `JobManagedBy` feature gate (enabled by default).

This feature allows you to disable the built-in Job controller, for a specific Job, and delegate reconciliation of the Job to an external controller.

You indicate the controller that reconciles the Job by setting a custom value for the `spec.managedBy` field - any value other than `kubernetes.io/job-controller`. The value of the field is immutable.

**Note:**

When using this feature, make sure the controller indicated by the field is installed, otherwise the Job may not be reconciled at all.

**Note:**

When developing an external Job controller be aware that your controller needs to operate in a fashion conformant with the definitions of the API spec and status fields of the Job object.

Please review these in detail in the Job API. We also recommend that you run the e2e conformance tests for the Job object to verify your implementation.

Finally, when developing an external Job controller make sure it does not use the `batch.kubernetes.io/job-tracking` finalizer, reserved for the built-in controller.

**Warning:**

If you are considering to disable the `JobManagedBy` feature gate, or to downgrade the cluster to a version without the feature gate enabled, check if there are jobs with a custom value of the `spec.managedBy` field. If there are such jobs, there is a risk that they might be reconciled by two controllers after the operation: the built-in Job controller and the external controller indicated by the field value.

## Alternatives

### Bare Pods

When the node that a Pod is running on reboots or fails, the pod is terminated and will not be restarted. However, a Job will create new Pods to replace terminated ones. For this reason, we recommend that you use a Job rather than a bare Pod, even if your application requires only a single Pod.

### Replication Controller

Jobs are complementary to Replication Controllers. A Replication Controller manages Pods which are not expected to terminate (e.g. web servers), and a Job manages Pods that are expected to terminate (e.g. batch tasks).

As discussed in Pod Lifecycle, `Job` is *only* appropriate for pods with `RestartPolicy` equal to `OnFailure` or `Never`.

**Note:**

If `RestartPolicy` is not set, the default value is `Always`.

### Single Job starts controller Pod

Another pattern is for a single Job to create a Pod which then creates other Pods, acting as a sort of custom controller for those Pods. This allows the most flexibility, but may be somewhat complicated to get started with and offers less integration with Kubernetes.

An advantage of this approach is that the overall process gets the completion guarantee of a Job object, but maintains complete control over what Pods are created and how work is assigned to them.

## What's next

- Learn about Pods.
- Read about different ways of running Jobs:
  - Coarse Parallel Processing Using a Work Queue
  - Fine Parallel Processing Using a Work Queue
  - Use an indexed Job for parallel processing with static work assignment
  - Create multiple Jobs based on a template: Parallel Processing using Expansions
- Follow the links within Clean up finished jobs automatically to learn more about how your cluster can clean up completed and / or failed tasks.
- `Job` is part of the Kubernetes REST API. Read the Job object definition to understand the API for jobs.
- Read about `CronJob`, which you can use to define a series of Jobs that will run based on a schedule, similar to the UNIX tool `cron`.
- Practice how to configure handling of retriable and non-retriable pod failures using `podFailurePolicy`, based on the step-by-step examples.

# Disruptions

This guide is for application owners who want to build highly available applications, and thus need to understand what types of disruptions can happen to Pods.

It is also for cluster administrators who want to perform automated cluster actions, like upgrading and autoscaling clusters.

## Voluntary and involuntary disruptions

Pods do not disappear until someone (a person or a controller) destroys them, or there is an unavoidable hardware or system software error.

We call these unavoidable cases *involuntary disruptions* to an application. Examples are:

- a hardware failure of the physical machine backing the node
- cluster administrator deletes VM (instance) by mistake
- cloud provider or hypervisor failure makes VM disappear
- a kernel panic
- the node disappears from the cluster due to cluster network partition
- eviction of a pod due to the node being out-of-resources.

Except for the out-of-resources condition, all these conditions should be familiar to most users; they are not specific to Kubernetes.

We call other cases *voluntary disruptions*. These include both actions initiated by the application owner and those initiated by a Cluster Administrator. Typical application owner actions include:

- deleting the deployment or other controller that manages the pod
- updating a deployment's pod template causing a restart
- directly deleting a pod (e.g. by accident)

Cluster administrator actions include:

- Draining a node for repair or upgrade.
- Draining a node from a cluster to scale the cluster down (learn about Node Autoscaling).
- Removing a pod from a node to permit something else to fit on that node.

These actions might be taken directly by the cluster administrator, or by automation run by the cluster administrator, or by your cluster hosting provider.

Ask your cluster administrator or consult your cloud provider or distribution documentation to determine if any sources of voluntary disruptions are enabled for your cluster. If none are enabled, you can skip creating Pod Disruption Budgets.

**Caution:**

Not all voluntary disruptions are constrained by Pod Disruption Budgets. For example, deleting deployments or pods bypasses Pod Disruption Budgets.

## Dealing with disruptions

Here are some ways to mitigate involuntary disruptions:

- Ensure your pod requests the resources it needs.
- Replicate your application if you need higher availability. (Learn about running replicated stateless and stateful applications.)
- For even higher availability when running replicated applications, spread applications across racks (using anti-affinity) or across zones (if using a multi-zone cluster.)

The frequency of voluntary disruptions varies. On a basic Kubernetes cluster, there are no automated voluntary disruptions (only user-triggered ones). However, your cluster administrator or hosting provider may run some additional services which cause voluntary disruptions. For example, rolling out node software updates can cause voluntary disruptions. Also, some implementations of cluster (node) autoscaling may cause voluntary disruptions to defragment and compact nodes. Your cluster administrator or hosting provider should have documented what level of voluntary disruptions, if any, to expect. Certain configuration options, such as using PriorityClasses in your pod spec can also cause voluntary (and involuntary) disruptions.

## Pod disruption budgets

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

Kubernetes offers features to help you run highly available applications even when you introduce frequent voluntary disruptions.

As an application owner, you can create a PodDisruptionBudget (PDB) for each application. A PDB limits the number of Pods of a replicated application that are down simultaneously from voluntary disruptions. For example, a quorum-based application would like to ensure that the number of replicas running is never brought below the number needed for a quorum. A web front end might want to ensure that the number of replicas serving load never falls below a certain percentage of the total.

Cluster managers and hosting providers should use tools which respect PodDisruptionBudgets by calling the Eviction API instead of directly deleting pods or deployments.

For example, the `kubectl drain` subcommand lets you mark a node as going out of service. When you run `kubectl drain`, the tool tries to evict all of the Pods on the Node you're taking out of service. The eviction request that `kubectl` submits on your behalf may be temporarily rejected, so the tool periodically retries all failed requests until all Pods on the target node are terminated, or until a configurable timeout is reached.

A PDB specifies the number of replicas that an application can tolerate having, relative to how many it is intended to have. For example, a Deployment which has a `.spec.replicas: 5` is supposed to have 5 pods at any given time. If its PDB allows for there to be 4 at a time, then the Eviction API will allow voluntary disruption of one (but not two) pods at a time.

The group of pods that comprise the application is specified using a label selector, the same as the one used by the application's controller (deployment, stateful-set, etc).

The "intended" number of pods is computed from the `.spec.replicas` of the workload resource that is managing those pods. The control plane discovers the owning workload resource by examining the `.metadata.ownerReferences` of the Pod.

Involuntary disruptions cannot be prevented by PDBs; however they do count against the budget.

Pods which are deleted or unavailable due to a rolling upgrade to an application do count against the disruption budget, but workload resources (such as Deployment and StatefulSet) are not limited by PDBs when doing rolling upgrades. Instead, the handling of failures during application updates is configured in the spec for the specific workload resource.

It is recommended to set `AlwaysAllow` Unhealthy Pod Eviction Policy to your PodDisruptionBudgets to support eviction of misbehaving applications during a node drain. The default behavior is to wait for the application pods to become healthy before the drain can proceed.

When a pod is evicted using the eviction API, it is gracefully terminated, honoring the `terminationGracePeriodSeconds` setting in its PodSpec.

## PodDisruptionBudget example

Consider a cluster with 3 nodes, `node-1` through `node-3`. The cluster is running several applications. One of them has 3 replicas initially called `pod-a`, `pod-b`, and `pod-c`. Another, unrelated pod without a PDB, called `pod-x`, is also shown. Initially, the pods are laid out as follows:

| **node-1** | **node-2** | **node-3** |
|---|---|---|
| pod-a *available* | pod-b *available* | pod-c *available* |
| pod-x *available* | | |

All 3 pods are part of a deployment, and they collectively have a PDB which requires there be at least 2 of the 3 pods to be available at all times.

For example, assume the cluster administrator wants to reboot into a new kernel version to fix a bug in the kernel. The cluster administrator first tries to drain `node-1` using the `kubectl drain` command. That tool tries to evict `pod-a` and `pod-x`. This succeeds immediately. Both pods go into the `terminating` state at the same time. This puts the cluster in this state:

| **node-1** ***draining*** | **node-2** | **node-3** |
|---|---|---|
| pod-a *terminating* | pod-b *available* | pod-c *available* |
| pod-x *terminating* | | |

The deployment notices that one of the pods is terminating, so it creates a replacement called `pod-d`. Since `node-1` is cordoned, it lands on another node. Something has also created `pod-y` as a replacement for `pod-x`.

(Note: for a StatefulSet, `pod-a`, which would be called something like `pod-0`, would need to terminate completely before its replacement, which is also called `pod-0` but has a different UID, could be created. Otherwise, the example applies to a StatefulSet as well.)

Now the cluster is in this state:

| **node-1** ***draining*** | **node-2** | **node-3** |
|---|---|---|
| pod-a *terminating* | pod-b *available* | pod-c *available* |
| pod-x *terminating* | pod-d *starting* | pod-y |

At some point, the pods terminate, and the cluster looks like this:

| **node-1** ***drained*** | **node-2** | **node-3** |
|---|---|---|
| | pod-b *available* | pod-c *available* |
| | pod-d *starting* | pod-y |

At this point, if an impatient cluster administrator tries to drain `node-2` or `node-3`, the drain command will block, because there are only 2 available pods for the deployment, and its PDB requires at least 2. After some time passes, `pod-d` becomes available.

The cluster state now looks like this:

| **node-1** ***drained*** | **node-2** | **node-3** |
|---|---|---|
| | pod-b *available* | pod-c *available* |
| | pod-d *available* | pod-y |

Now, the cluster administrator tries to drain `node-2`. The drain command will try to evict the two pods in some order, say `pod-b` first and then `pod-d`. It will succeed at evicting `pod-b`. But, when it tries to evict `pod-d`, it will be refused because that would leave only one pod available for the deployment.

The deployment creates a replacement for `pod-b` called `pod-e`. Because there are not enough resources in the cluster to schedule `pod-e` the drain will again block. The cluster may end up in this state:

| **node-1** ***drained*** | **node-2** | **node-3** | ***no node*** |
|---|---|---|---|
| | pod-b *terminating* | pod-c *available* | pod-e *pending* |
| | pod-d *available* | pod-y | |

At this point, the cluster administrator needs to add a node back to the cluster to proceed with the upgrade.

You can see how Kubernetes varies the rate at which disruptions can happen, according to:

- how many replicas an application needs
- how long it takes to gracefully shutdown an instance
- how long it takes a new instance to start up
- the type of controller
- the cluster's resource capacity

## Pod disruption conditions

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

A dedicated Pod `DisruptionTarget` condition is added to indicate that the Pod is about to be deleted due to a disruption. The `reason` field of the condition additionally indicates one of the following reasons for the Pod termination:

`PreemptionByScheduler`
: Pod is due to be preempted by a scheduler in order to accommodate a new Pod with a higher priority. For more information, see Pod priority preemption.

`DeletionByTaintManager`
: Pod is due to be deleted by Taint Manager (which is part of the node lifecycle controller within `kube-controller-manager`) due to a `NoExecute` taint that the Pod does not tolerate; see taint-based evictions.

`EvictionByEvictionAPI`
: Pod has been marked for eviction using the Kubernetes API .

`DeletionByPodGC`
: Pod, that is bound to a no longer existing Node, is due to be deleted by Pod garbage collection.

`TerminationByKubelet`
: Pod has been terminated by the kubelet, because of either node pressure eviction, the graceful node shutdown, or preemption for system critical pods.

In all other disruption scenarios, like eviction due to exceeding Pod container limits, Pods don't receive the `DisruptionTarget` condition because the disruptions were probably caused by the Pod and would reoccur on retry.

**Note:**

A Pod disruption might be interrupted. The control plane might re-attempt to continue the disruption of the same Pod, but it is not guaranteed. As a result, the `DisruptionTarget` condition might be added to a Pod, but that Pod might then not actually be deleted. In such a situation, after some time, the Pod disruption condition will be cleared.

Along with cleaning up the pods, the Pod garbage collector (PodGC) will also mark them as failed if they are in a non-terminal phase (see also Pod garbage collection).

When using a Job (or CronJob), you may want to use these Pod disruption conditions as part of your Job's Pod failure policy.

## Separating Cluster Owner and Application Owner Roles

Often, it is useful to think of the Cluster Manager and Application Owner as separate roles with limited knowledge of each other. This separation of responsibilities may make sense in these scenarios:

- when there are many application teams sharing a Kubernetes cluster, and there is natural specialization of roles
- when third-party tools or services are used to automate cluster management

Pod Disruption Budgets support this separation of roles by providing an interface between the roles.

If you do not have such a separation of responsibilities in your organization, you may not need to use Pod Disruption Budgets.

## How to perform Disruptive Actions on your Cluster

If you are a Cluster Administrator, and you need to perform a disruptive action on all the nodes in your cluster, such as a node or system software upgrade, here are some options:

- Accept downtime during the upgrade.
- Failover to another complete replica cluster.
  - No downtime, but may be costly both for the duplicated nodes and for human effort to orchestrate the switchover.
- Write disruption tolerant applications and use PDBs.
  - No downtime.
  - Minimal resource duplication.
  - Allows more automation of cluster administration.
  - Writing disruption-tolerant applications is tricky, but the work to tolerate voluntary disruptions largely overlaps with work to support autoscaling and tolerating involuntary disruptions.

## What's next

- Follow steps to protect your application by configuring a Pod Disruption Budget.

- Learn more about draining nodes

- Learn about updating a deployment including steps to maintain its availability during the rollout.

# User Namespaces

FEATURE STATE: `Kubernetes v1.30 [beta]`

This page explains how user namespaces are used in Kubernetes pods. A user namespace isolates the user running inside the container from the one in the host.

A process running as root in a container can run as a different (non-root) user in the host; in other words, the process has full privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace.

You can use this feature to reduce the damage a compromised container can do to the host or other pods in the same node. There are several security vulnerabilities rated either **HIGH** or **CRITICAL** that were not exploitable when user namespaces is active. It is expected user namespace will mitigate some future vulnerabilities too.

## Before you begin

**Note:** This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

This is a Linux-only feature and support is needed in Linux for idmap mounts on the filesystems used. This means:

- On the node, the filesystem you use for `/var/lib/kubelet/pods/`, or the custom directory you configure for this, needs idmap mount support.
- All the filesystems used in the pod's volumes must support idmap mounts.

In practice this means you need at least Linux 6.3, as tmpfs started supporting idmap mounts in that version. This is usually needed as several Kubernetes features use tmpfs (the service account token that is mounted by default uses a tmpfs, Secrets use a tmpfs, etc.)

Some popular filesystems that support idmap mounts in Linux 6.3 are: btrfs, ext4, xfs, fat, tmpfs, overlayfs.

In addition, the container runtime and its underlying OCI runtime must support user namespaces. The following OCI runtimes offer support:

- crun version 1.9 or greater (it's recommend version 1.13+).
- runc version 1.2 or greater

**Note:**

Some OCI runtimes do not include the support needed for using user namespaces in Linux pods. If you use a managed Kubernetes, or have downloaded it from packages and set it up, it's possible that nodes in your cluster use a runtime that doesn't include this support.

To use user namespaces with Kubernetes, you also need to use a CRI container runtime to use this feature with Kubernetes pods:

- containerd: version 2.0 (and later) supports user namespaces for containers.
- CRI-O: version 1.25 (and later) supports user namespaces for containers.

You can see the status of user namespaces support in cri-dockerd tracked in an issue on GitHub.

## Introduction

User namespaces is a Linux feature that allows to map users in the container to different users in the host. Furthermore, the capabilities granted to a pod in a user namespace are valid only in the namespace and void outside of it.

A pod can opt-in to use user namespaces by setting the `pod.spec.hostUsers` field to `false`.

The kubelet will pick host UIDs/GIDs a pod is mapped to, and will do so in a way to guarantee that no two pods on the same node use the same mapping.

The `runAsUser`, `runAsGroup`, `fsGroup`, etc. fields in the `pod.spec` always refer to the user inside the container. These users will be used for volume mounts (specified in `pod.spec.volumes`) and therefore the host UID/GID will not have any effect on writes/reads from volumes the pod can mount. In other words, the inodes created/read in volumes mounted by the pod will be the same as if the pod wasn't using user namespaces.

This way, a pod can easily enable and disable user namespaces (without affecting its volume's file ownerships) and can also share volumes with pods without user namespaces by just setting the appropriate users inside the container (`RunAsUser`, `RunAsGroup`, `fsGroup`, etc.). This applies to any volume the pod can mount, including `hostPath` (if the pod is allowed to mount `hostPath` volumes).

By default, the valid UIDs/GIDs when this feature is enabled is the range 0-65535. This applies to files and processes (`runAsUser`, `runAsGroup`, etc.).

Files using a UID/GID outside this range will be seen as belonging to the overflow ID, usually 65534 (configured in `/proc/sys/kernel/overflowuid` and `/proc/sys/kernel/overflowgid`). However, it is not possible to modify those files, even by running as the 65534 user/group.

If the range 0-65535 is extended with a configuration knob, the aforementioned restrictions apply to the extended range.

Most applications that need to run as root but don't access other host namespaces or resources, should continue to run fine without any changes needed if user namespaces is activated.

## Understanding user namespaces for pods

Several container runtimes with their default configuration (like Docker Engine, containerd, CRI-O) use Linux namespaces for isolation. Other technologies exist and can be used with those runtimes too (e.g. Kata Containers uses VMs instead of Linux namespaces). This page is applicable for container runtimes using Linux namespaces for isolation.

When creating a pod, by default, several new namespaces are used for isolation: a network namespace to isolate the network of the container, a PID namespace to isolate the view of processes, etc. If a user namespace is used, this will isolate the users in the container from the users in the node.

This means containers can run as root and be mapped to a non-root user on the host. Inside the container the process will think it is running as root (and therefore tools like `apt`, `yum`, etc. work fine), while in reality the process doesn't have privileges on the host. You can verify this, for example, if you check which user the container process is running by executing `ps aux` from the host. The user `ps` shows is not the same as the user you see if you execute inside the container the command `id`.

This abstraction limits what can happen, for example, if the container manages to escape to the host. Given that the container is running as a non-privileged user on the host, it is limited what it can do to the host.

Furthermore, as users on each pod will be mapped to different non-overlapping users in the host, it is limited what they can do to other pods too.

Capabilities granted to a pod are also limited to the pod user namespace and mostly invalid out of it, some are even completely void. Here are two examples:

- `CAP_SYS_MODULE` does not have any effect if granted to a pod using user namespaces, the pod isn't able to load kernel modules.
- `CAP_SYS_ADMIN` is limited to the pod's user namespace and invalid outside of it.

Without using a user namespace a container running as root, in the case of a container breakout, has root privileges on the node. And if some capability were granted to the container, the capabilities are valid on the host too. None of this is true when we use user namespaces.

If you want to know more details about what changes when user namespaces are in use, see `man 7 user_namespaces`.

## Set up a node to support user namespaces

By default, the kubelet assigns pods UIDs/GIDs above the range 0-65535, based on the assumption that the host's files and processes use UIDs/GIDs within this range, which is standard for most Linux distributions. This approach prevents any overlap between the UIDs/GIDs of the host and those of the pods.

Avoiding the overlap is important to mitigate the impact of vulnerabilities such as CVE-2021-25741, where a pod can potentially read arbitrary files in the host. If the UIDs/GIDs of the pod and the host don't overlap, it is limited what a pod would be able to do: the pod UID/GID won't match the host's file owner/group.

The kubelet can use a custom range for user IDs and group IDs for pods. To configure a custom range, the node needs to have:

- A user `kubelet` in the system (you cannot use any other username here)
- The binary `getsubids` installed (part of shadow-utils) and in the `PATH` for the kubelet binary.
- A configuration of subordinate UIDs/GIDs for the `kubelet` user (see `man 5 subuid` and `man 5 subgid`).

This setting only gathers the UID/GID range configuration and does not change the user executing the `kubelet`.

You must follow some constraints for the subordinate ID range that you assign to the `kubelet` user:

- The subordinate user ID, that starts the UID range for Pods, **must** be a multiple of 65536 and must also be greater than or equal to 65536. In other words, you cannot use any ID from the range 0-65535 for Pods; the kubelet imposes this restriction to make it difficult to create an accidentally insecure configuration.

- The subordinate ID count must be a multiple of 65536

- The subordinate ID count must be at least `65536 x <maxPods>` where `<maxPods>` is the maximum number of pods that can run on the node.

- You must assign the same range for both user IDs and for group IDs, It doesn't matter if other users have user ID ranges that don't align with the group ID ranges.

- None of the assigned ranges should overlap with any other assignment.

- The subordinate configuration must be only one line. In other words, you can't have multiple ranges.

For example, you could define `/etc/subuid` and `/etc/subgid` to both have these entries for the `kubelet` user:

```
# The format is
#   name:firstID:count of IDs
# where
# - firstID is 65536 (the minimum value possible)
# - count of IDs is 110 * 65536
#   (110 is the default limit for number of pods on the node)

kubelet:65536:7208960
```

## ID count for each of Pods

Starting with Kubernetes v1.33, the ID count for each of Pods can be set in `KubeletConfiguration`.

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfigurationuserNamespaces:  idsPerPod: 1048576
```

The value of `idsPerPod` (uint32) must be a multiple of 65536. The default value is 65536. This value only applies to containers created after the kubelet was started with this `KubeletConfiguration`. Running containers are not affected by this config.

In Kubernetes prior to v1.33, the ID count for each of Pods was hard-coded to 65536.

## Integration with Pod security admission checks

FEATURE STATE: `Kubernetes v1.29 [alpha]`

For Linux Pods that enable user namespaces, Kubernetes relaxes the application of Pod Security Standards in a controlled way. This behavior can be controlled by the feature gate `UserNamespacesPodSecurityStandards`, which allows an early opt-in for end users. Admins have to ensure that user namespaces are enabled by all nodes within the cluster if using the feature gate.

If you enable the associated feature gate and create a Pod that uses user namespaces, the following fields won't be constrained even in contexts that enforce the *Baseline* or *Restricted* pod security standard. This behavior does not present a security concern because `root` inside a Pod with user namespaces actually refers to the user inside the container, that is never mapped to a privileged user on the host. Here's the list of fields that are **not** checks for Pods in those circumstances:

- `spec.securityContext.runAsNonRoot`
- `spec.containers[*].securityContext.runAsNonRoot`
- `spec.initContainers[*].securityContext.runAsNonRoot`
- `spec.ephemeralContainers[*].securityContext.runAsNonRoot`
- `spec.securityContext.runAsUser`
- `spec.containers[*].securityContext.runAsUser`
- `spec.initContainers[*].securityContext.runAsUser`
- `spec.ephemeralContainers[*].securityContext.runAsUser`

## Limitations

When using a user namespace for the pod, it is disallowed to use other host namespaces. In particular, if you set `hostUsers: false` then you are not allowed to set any of:

- `hostNetwork: true`
- `hostIPC: true`
- `hostPID: true`

No container can use `volumeDevices` (raw block volumes, like /dev/sda) either. This includes all the container arrays in the pod spec:

- `containers`
- `initContainers`
- `ephemeralContainers`

## Metrics and observability

The kubelet exports two prometheus metrics specific to user-namespaces:

- `started_user_namespaced_pods_total`: a counter that tracks the number of user namespaced pods that are attempted to be created.
- `started_user_namespaced_pods_errors_total`: a counter that tracks the number of errors creating user namespaced pods.

## What's next

- Take a look at Use a User Namespace With a Pod

---

# Windows in Kubernetes

Kubernetes supports nodes that run Microsoft Windows.

Kubernetes supports worker nodes running either Linux or Microsoft Windows.

□ This item links to a third party project or product that is not part of Kubernetes itself. More information

The CNCF and its parent the Linux Foundation take a vendor-neutral approach towards compatibility. It is possible to join your Windows server as a worker node to a Kubernetes cluster.

You can install and set up kubectl on Windows no matter what operating system you use within your cluster.

If you are using Windows nodes, you can read:

- Networking On Windows
- Windows Storage In Kubernetes
- Resource Management for Windows Nodes
- Configure RunAsUserName for Windows Pods and Containers
- Create A Windows HostProcess Pod
- Configure Group Managed Service Accounts for Windows Pods and Containers
- Security For Windows Nodes
- Windows Debugging Tips
- Guide for Scheduling Windows Containers in Kubernetes

or, for an overview, read:

- Windows containers in Kubernetes
- Guide for Running Windows Containers in Kubernetes

---

# Pods

*Pods* are the smallest deployable units of computing that you can create and manage in Kubernetes.

A *Pod* (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host.

As well as application containers, a Pod can contain init containers that run during Pod startup. You can also inject ephemeral containers for debugging a running Pod.

## What is a Pod?

**Note:**

You need to install a container runtime into each node in the cluster so that Pods can run there.

The shared context of a Pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation - the same things that isolate a container. Within a Pod's context, the individual applications may have further sub-isolations applied.

A Pod is similar to a set of containers with shared namespaces and shared filesystem volumes.

Pods in a Kubernetes cluster are used in two main ways:

- **Pods that run a single container**. The "one-container-per-Pod" model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container; Kubernetes manages Pods rather than managing the containers directly.
- **Pods that run multiple containers that need to work together**. A Pod can encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers form a single cohesive unit.

  Grouping multiple co-located and co-managed containers in a single Pod is a relatively advanced use case. You should use this pattern only in specific instances in which your containers are tightly coupled.

  You don't need to run multiple containers to provide replication (for resilience or capacity); if you need multiple replicas, see Workload management.

# Using Pods

The following is an example of a Pod which consists of a container running the image `nginx:1.14.2`.

`pods/simple-pod.yaml` Copy pods/simple-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: nginxspec:  containers:  - name: nginx    image: nginx:1.14.2    ports:    - containerPort: 80
```

To create the Pod shown above, run the following command:

```
kubectl apply -f https://k8s.io/examples/pods/simple-pod.yaml
```

Pods are generally not created directly and are created using workload resources. See Working with Pods for more information on how Pods are used with workload resources.

### Workload resources for managing pods

Usually you don't need to create Pods directly, even singleton Pods. Instead, create them using workload resources such as Deployment or Job. If your Pods need to track state, consider the StatefulSet resource.

Each Pod is meant to run a single instance of a given application. If you want to scale your application horizontally (to provide more overall resources by running more instances), you should use multiple Pods, one for each instance. In Kubernetes, this is typically referred to as *replication*. Replicated Pods are usually created and managed as a group by a workload resource and its controller.

See Pods and controllers for more information on how Kubernetes uses workload resources, and their controllers, to implement application scaling and auto-healing.

Pods natively provide two kinds of shared resources for their constituent containers: networking and storage.

# Working with Pods

You'll rarely create individual Pods directly in Kubernetes—even singleton Pods. This is because Pods are designed as relatively ephemeral, disposable entities. When a Pod gets created (directly by you, or indirectly by a controller), the new Pod is scheduled to run on a Node in your cluster. The Pod remains on that node until the Pod finishes execution, the Pod object is deleted, the Pod is *evicted* for lack of resources, or the node fails.

**Note:**

Restarting a container in a Pod should not be confused with restarting a Pod. A Pod is not a process, but an environment for running container(s). A Pod persists until it is deleted.

The name of a Pod must be a valid DNS subdomain value, but this can produce unexpected results for the Pod hostname. For best compatibility, the name should follow the more restrictive rules for a DNS label.

### Pod OS

FEATURE STATE: `Kubernetes v1.25 [stable]`

You should set the `.spec.os.name` field to either `windows` or `linux` to indicate the OS on which you want the pod to run. These two are the only operating systems supported for now by Kubernetes. In the future, this list may be expanded.

In Kubernetes v1.34, the value of `.spec.os.name` does not affect how the kube-scheduler picks a node for the Pod to run on. In any cluster where there is more than one operating system for running nodes, you should set the kubernetes.io/os label correctly on each node, and define pods with a `nodeSelector` based on the operating system label. The kube-scheduler assigns your pod to a node based on other criteria and may or may not succeed in picking a suitable node placement where the node OS is right for the containers in that Pod. The Pod security standards also use this field to avoid enforcing policies that aren't relevant to the operating system.

### Pods and controllers

You can use workload resources to create and manage multiple Pods for you. A controller for the resource handles replication and rollout and automatic healing in case of Pod failure. For example, if a Node fails, a controller notices that Pods on that Node have stopped working and creates a replacement Pod. The scheduler places the replacement Pod onto a healthy Node.

Here are some examples of workload resources that manage one or more Pods:

- Deployment
- StatefulSet
- DaemonSet

### Pod templates

Controllers for workload resources create Pods from a *pod template* and manage those Pods on your behalf.

PodTemplates are specifications for creating Pods, and are included in workload resources such as Deployments, Jobs, and DaemonSets.

Each controller for a workload resource uses the `PodTemplate` inside the workload object to make actual Pods. The `PodTemplate` is part of the desired state of whatever workload resource you used to run your app.

When you create a Pod, you can include environment variables in the Pod template for the containers that run in the Pod.

The sample below is a manifest for a simple Job with a `template` that starts one container. The container in that Pod prints a message then pauses.

```
apiVersion: batch/v1
kind: Jobmetadata:  name: hellospec:  template:    # This is the pod template    spec:      containers:      - name: hello
```

Modifying the pod template or switching to a new pod template has no direct effect on the Pods that already exist. If you change the pod template for a workload resource, that resource needs to create replacement Pods that use the updated template.

For example, the StatefulSet controller ensures that the running Pods match the current pod template for each StatefulSet object. If you edit the StatefulSet to change its pod template, the StatefulSet starts to create new Pods based on the updated template. Eventually, all of the old Pods are replaced with new Pods, and the update is complete.

Each workload resource implements its own rules for handling changes to the Pod template. If you want to read more about StatefulSet specifically, read Update strategy in the StatefulSet Basics tutorial.

On Nodes, the kubelet does not directly observe or manage any of the details around pod templates and updates; those details are abstracted away. That abstraction and separation of concerns simplifies system semantics, and makes it feasible to extend the cluster's behavior without changing existing code.

## Pod update and replacement

As mentioned in the previous section, when the Pod template for a workload resource is changed, the controller creates new Pods based on the updated template instead of updating or patching the existing Pods.

Kubernetes doesn't prevent you from managing Pods directly. It is possible to update some fields of a running Pod, in place. However, Pod update operations like `patch`, and `replace` have some limitations:

- Most of the metadata about a Pod is immutable. For example, you cannot change the `namespace`, `name`, `uid`, or `creationTimestamp` fields.
- If the `metadata.deletionTimestamp` is set, no new entry can be added to the `metadata.finalizers` list.
- Pod updates may not change fields other than `spec.containers[*].image`, `spec.initContainers[*].image`, `spec.activeDeadlineSeconds`, `spec.terminationGracePeriodSeconds`, `spec.tolerations` or `spec.schedulingGates`. For `spec.tolerations`, you can only add new entries.
- When updating the `spec.activeDeadlineSeconds` field, two types of updates are allowed:

    1. setting the unassigned field to a positive number;
    2. updating the field from a positive number to a smaller, non-negative number.

### Pod subresources

The above update rules apply to regular pod updates, but other pod fields can be updated through *subresources*.

- **Resize:** The `resize` subresource allows container resources (`spec.containers[*].resources`) to be updated. See Resize Container Resources for more details.
- **Ephemeral Containers:** The `ephemeralContainers` subresource allows ephemeral containers to be added to a Pod. See Ephemeral Containers for more details.
- **Status:** The `status` subresource allows the pod status to be updated. This is typically only used by the Kubelet and other system controllers.
- **Binding:** The `binding` subresource allows setting the pod's `spec.nodeName` via a `Binding` request. This is typically only used by the scheduler.

### Pod generation

- The `metadata.generation` field is unique. It will be automatically set by the system such that new pods have a `metadata.generation` of 1, and every update to mutable fields in the pod's spec will increment the `metadata.generation` by 1.

FEATURE STATE: `Kubernetes v1.34 [beta]` (enabled by default: true)

- `observedGeneration` is a field that is captured in the `status` section of the Pod object. If the feature gate `PodObservedGenerationTracking` is set, the Kubelet will set `status.observedGeneration` to track the pod state to the current pod status. The pod's `status.observedGeneration` will reflect the `metadata.generation` of the pod at the point that the pod status is being reported.

**Note:**

The `status.observedGeneration` field is managed by the kubelet and external controllers should **not** modify this field.

Different status fields may either be associated with the `metadata.generation` of the current sync loop, or with the `metadata.generation` of the previous sync loop. The key distinction is whether a change in the `spec` is reflected directly in the `status` or is an indirect result of a running process.

#### Direct Status Updates

For status fields where the allocated spec is directly reflected, the `observedGeneration` will be associated with the current `metadata.generation` (Generation N).

This behavior applies to:

- **Resize Status**: The status of a resource resize operation.
- **Allocated Resources**: The resources allocated to the Pod after a resize.
- **Ephemeral Containers**: When a new ephemeral container is added, and it is in `Waiting` state.

#### Indirect Status Updates

For status fields that are an indirect result of running the spec, the `observedGeneration` will be associated with the `metadata.generation` of the previous sync loop (Generation N-1).

This behavior applies to:

- **Container Image**: The `ContainerStatus.ImageID` reflects the image from the previous generation until the new image is pulled and the container is updated.
- **Actual Resources**: During an in-progress resize, the actual resources in use still belong to the previous generation's request.
- **Container state**: During an in-progress resize, with require restart policy reflects the previous generation's request.
- **activeDeadlineSeconds** & **terminationGracePeriodSeconds** & **deletionTimestamp**: The effects of these fields on the Pod's status are a result of the previously observed specification.

## Resource sharing and communication

Pods enable data sharing and communication among their constituent containers.

### Storage in Pods

A Pod can specify a set of shared storage volumes. All containers in the Pod can access the shared volumes, allowing those containers to share data. Volumes also allow persistent data in a Pod to survive in case one of the containers within needs to be restarted. See Storage for more information on how Kubernetes implements shared storage and makes it available to Pods.

### Pod networking

Each Pod is assigned a unique IP address for each address family. Every container in a Pod shares the network namespace, including the IP address and network ports. Inside a Pod (and **only** then), the containers that belong to the Pod can communicate with one another using `localhost`. When containers in a Pod communicate with entities *outside the Pod*, they must coordinate how they use the shared network resources (such as ports). Within a Pod, containers share an IP address and port space, and can find each other via `localhost`. The containers in a Pod can also communicate with each other using standard inter-process communications like SystemV semaphores or POSIX shared memory. Containers in different Pods have distinct IP addresses and can not communicate by OS-level IPC without special configuration. Containers that want to interact with a container running in a different Pod can use IP networking to communicate.

Containers within the Pod see the system hostname as being the same as the configured `name` for the Pod. There's more about this in the networking section.

## Pod security settings

To set security constraints on Pods and containers, you use the `securityContext` field in the Pod specification. This field gives you granular control over what a Pod or individual containers can do. For example:

- Drop specific Linux capabilities to avoid the impact of a CVE.
- Force all processes in the Pod to run as a non-root user or as a specific user or group ID.
- Set a specific seccomp profile.
- Set Windows security options, such as whether containers run as HostProcess.

#### Caution:

You can also use the Pod securityContext to enable *privileged mode* in Linux containers. Privileged mode overrides many of the other security settings in the securityContext. Avoid using this setting unless you can't grant the equivalent permissions by using other fields in the securityContext. In Kubernetes 1.26 and later, you can run Windows containers in a similarly privileged mode by setting the `windowsOptions.hostProcess` flag on the security context of the Pod spec. For details and instructions, see Create a Windows HostProcess Pod.

- To learn about kernel-level security constraints that you can use, see Linux kernel security constraints for Pods and containers.
- To learn more about the Pod security context, see Configure a Security Context for a Pod or Container.

## Static Pods

*Static Pods* are managed directly by the kubelet daemon on a specific node, without the API server observing them. Whereas most Pods are managed by the control plane (for example, a Deployment), for static Pods, the kubelet directly supervises each static Pod (and restarts it if it fails).

Static Pods are always bound to one Kubelet on a specific node. The main use for static Pods is to run a self-hosted control plane: in other words, using the kubelet to supervise the individual control plane components.

The kubelet automatically tries to create a mirror Pod on the Kubernetes API server for each static Pod. This means that the Pods running on a node are visible on the API server, but cannot be controlled from there. See the guide Create static Pods for more information.

#### Note:

The `spec` of a static Pod cannot refer to other API objects (e.g., ServiceAccount, ConfigMap, Secret, etc).

## Pods with multiple containers

Pods are designed to support multiple cooperating processes (as containers) that form a cohesive unit of service. The containers in a Pod are automatically co-located and co-scheduled on the same physical or virtual machine in the cluster. The containers can share resources and dependencies, communicate with one another, and coordinate when and how they are terminated.

Pods in a Kubernetes cluster are used in two main ways:

- **Pods that run a single container**. The "one-container-per-Pod" model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container; Kubernetes manages Pods rather than managing the containers directly.
- **Pods that run multiple containers that need to work together**. A Pod can encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers form a single cohesive unit of service—for example, one container serving data stored in a shared volume to the public, while a separate sidecar container refreshes or updates those files. The Pod wraps these containers, storage resources, and an ephemeral network identity together as a single unit.

For example, you might have a container that acts as a web server for files in a shared volume, and a separate sidecar container that updates those files from a remote source, as in the following diagram:

Pod creation diagram

Some Pods have init containers as well as app containers. By default, init containers run and complete before the app containers are started.

You can also have sidecar containers that provide auxiliary services to the main application Pod (for example: a service mesh).

FEATURE STATE: `Kubernetes v1.33 [stable]` (enabled by default: true)

Enabled by default, the `SidecarContainers` feature gate allows you to specify `restartPolicy: Always` for init containers. Setting the `Always` restart policy ensures that the containers where you set it are treated as *sidecars* that are kept running during the entire lifetime of the Pod. Containers that you explicitly define as sidecar containers start up before the main application Pod and remain running until the Pod is shut down.

## Container probes

A *probe* is a diagnostic performed periodically by the kubelet on a container. To perform a diagnostic, the kubelet can invoke different actions:

- `ExecAction` (performed with the help of the container runtime)
- `TCPSocketAction` (checked directly by the kubelet)
- `HTTPGetAction` (checked directly by the kubelet)

You can read more about probes in the Pod Lifecycle documentation.

## What's next

- Learn about the lifecycle of a Pod.
- Learn about RuntimeClass and how you can use it to configure different Pods with different container runtime configurations.
- Read about PodDisruptionBudget and how you can use it to manage application availability during disruptions.
- Pod is a top-level resource in the Kubernetes REST API. The Pod object definition describes the object in detail.
- The Distributed System Toolkit: Patterns for Composite Containers explains common layouts for Pods with more than one container.
- Read about Pod topology spread constraints

To understand the context for why Kubernetes wraps a common Pod API in other resources (such as StatefulSets or Deployments), you can read about the prior art, including:

- Aurora
- Borg
- Marathon
- Omega
- Tupperware.

---

# Automatic Cleanup for Finished Jobs

A time-to-live mechanism to clean up old Jobs that have finished execution.
FEATURE STATE: `Kubernetes v1.23 [stable]`

When your Job has finished, it's useful to keep that Job in the API (and not immediately delete the Job) so that you can tell whether the Job succeeded or failed.

Kubernetes' TTL-after-finished controller provides a TTL (time to live) mechanism to limit the lifetime of Job objects that have finished execution.

## Cleanup for finished Jobs

The TTL-after-finished controller is only supported for Jobs. You can use this mechanism to clean up finished Jobs (either `Complete` or `Failed`) automatically by specifying the `.spec.ttlSecondsAfterFinished` field of a Job, as in this example.

The TTL-after-finished controller assumes that a Job is eligible to be cleaned up TTL seconds after the Job has finished. The timer starts once the status condition of the Job changes to show that the Job is either `Complete` or `Failed`; once the TTL has expired, that Job becomes eligible for cascading removal.

When the TTL-after-finished controller cleans up a job, it will delete it cascadingly, that is to say it will delete its dependent objects together with it.

Kubernetes honors object lifecycle guarantees on the Job, such as waiting for finalizers.

You can set the TTL seconds at any time. Here are some examples for setting the `.spec.ttlSecondsAfterFinished` field of a Job:

- Specify this field in the Job manifest, so that a Job can be cleaned up automatically some time after it finishes.
- Manually set this field of existing, already finished Jobs, so that they become eligible for cleanup.
- Use a mutating admission webhook to set this field dynamically at Job creation time. Cluster administrators can use this to enforce a TTL policy for finished jobs.
- Use a mutating admission webhook to set this field dynamically after the Job has finished, and choose different TTL values based on job status, labels. For this case, the webhook needs to detect changes to the `.status` of the Job and only set a TTL when the Job is being marked as completed.
- Write your own controller to manage the cleanup TTL for Jobs that match a particular selector.

## Caveats

### Updating TTL for finished Jobs

You can modify the TTL period, e.g. `.spec.ttlSecondsAfterFinished` field of Jobs, after the job is created or has finished. If you extend the TTL period after the existing `ttlSecondsAfterFinished` period has expired, Kubernetes doesn't guarantee to retain that Job, even if an update to extend the TTL returns a successful API response.

### Time skew

Because the TTL-after-finished controller uses timestamps stored in the Kubernetes jobs to determine whether the TTL has expired or not, this feature is sensitive to time skew in your cluster, which may cause the control plane to clean up Job objects at the wrong time.

Clocks aren't always correct, but the difference should be very small. Please be aware of this risk when setting a non-zero TTL.

## What's next

- Read Clean up Jobs automatically

- Refer to the Kubernetes Enhancement Proposal (KEP) for adding this mechanism.

# CronJob

A CronJob starts one-time Jobs on a repeating schedule.
FEATURE STATE: `Kubernetes v1.21 [stable]`

A *CronJob* creates Jobs on a repeating schedule.

CronJob is meant for performing regular scheduled actions such as backups, report generation, and so on. One CronJob object is like one line of a *crontab* (cron table) file on a Unix system. It runs a Job periodically on a given schedule, written in Cron format.

CronJobs have limitations and idiosyncrasies. For example, in certain circumstances, a single CronJob can create multiple concurrent Jobs. See the limitations below.

When the control plane creates new Jobs and (indirectly) Pods for a CronJob, the `.metadata.name` of the CronJob is part of the basis for naming those Pods. The name of a CronJob must be a valid DNS subdomain value, but this can produce unexpected results for the Pod hostnames. For best compatibility, the name should follow the more restrictive rules for a DNS label. Even when the name is a DNS subdomain, the name must be no longer than 52 characters. This is because the CronJob controller will automatically append 11 characters to the name you provide and there is a constraint that the length of a Job name is no more than 63 characters.

## Example

This example CronJob manifest prints the current time and a hello message every minute:

`application/job/cronjob.yaml` Copy application/job/cronjob.yaml to clipboard

```
apiVersion: batch/v1
kind: CronJobmetadata:  name: hellospec:  schedule: "* * * * *"  jobTemplate:    spec:      template:        spec:          contai
```

(Running Automated Tasks with a CronJob takes you through this example in more detail).

## Writing a CronJob spec

### Schedule syntax

The `.spec.schedule` field is required. The value of that field follows the Cron syntax:

```
# ┌───────────── minute (0 - 59)
# │ ┌───────────── hour (0 - 23)
# │ │ ┌───────────── day of the month (1 - 31)
# │ │ │ ┌───────────── month (1 - 12)
# │ │ │ │ ┌───────────── day of the week (0 - 6) (Sunday to Saturday)
# │ │ │ │ │                                   OR sun, mon, tue, wed, thu, fri, sat
# │ │ │ │ │
# | | | | |
# * * * * *
```

For example, `0 3 * * 1` means this task is scheduled to run weekly on a Monday at 3 AM.

The format also includes extended "Vixie cron" step values. As explained in the FreeBSD manual:

> Step values can be used in conjunction with ranges. Following a range with `/<number>` specifies skips of the number's value through the range. For example, `0-23/2` can be used in the hours field to specify command execution every other hour (the alternative in the V7 standard is `0,2,4,6,8,10,12,14,16,18,20,22`). Steps are also permitted after an asterisk, so if you want to say "every two hours", just use `*/2`.

**Note:**

A question mark (`?`) in the schedule has the same meaning as an asterisk `*`, that is, it stands for any of available value for a given field.

Other than the standard syntax, some macros like `@monthly` can also be used:

| Entry | Description | Equivalent to |
|---|---|---|
| @yearly (or @annually) | Run once a year at midnight of 1 January | 0 0 1 1 * |
| @monthly | Run once a month at midnight of the first day of the month | 0 0 1 * * |
| @weekly | Run once a week at midnight on Sunday morning | 0 0 * * 0 |
| @daily (or @midnight) | Run once a day at midnight | 0 0 * * * |
| @hourly | Run once an hour at the beginning of the hour | 0 * * * * |

To generate CronJob schedule expressions, you can also use web tools like crontab.guru.

## Job template

The `.spec.jobTemplate` defines a template for the Jobs that the CronJob creates, and it is required. It has exactly the same schema as a Job, except that it is nested and does not have an `apiVersion` or `kind`. You can specify common metadata for the templated Jobs, such as labels or annotations. For information about writing a Job `.spec`, see Writing a Job Spec.

## Deadline for delayed Job start

The `.spec.startingDeadlineSeconds` field is optional. This field defines a deadline (in whole seconds) for starting the Job, if that Job misses its scheduled time for any reason.

After missing the deadline, the CronJob skips that instance of the Job (future occurrences are still scheduled). For example, if you have a backup Job that runs twice a day, you might allow it to start up to 8 hours late, but no later, because a backup taken any later wouldn't be useful: you would instead prefer to wait for the next scheduled run.

For Jobs that miss their configured deadline, Kubernetes treats them as failed Jobs. If you don't specify `startingDeadlineSeconds` for a CronJob, the Job occurrences have no deadline.

If the `.spec.startingDeadlineSeconds` field is set (not null), the CronJob controller measures the time between when a Job is expected to be created and now. If the difference is higher than that limit, it will skip this execution.

For example, if it is set to `200`, it allows a Job to be created for up to 200 seconds after the actual schedule.

## Concurrency policy

The `.spec.concurrencyPolicy` field is also optional. It specifies how to treat concurrent executions of a Job that is created by this CronJob. The spec may specify only one of the following concurrency policies:

- `Allow` (default): The CronJob allows concurrently running Jobs
- `Forbid`: The CronJob does not allow concurrent runs; if it is time for a new Job run and the previous Job run hasn't finished yet, the CronJob skips the new Job run. Also note that when the previous Job run finishes, `.spec.startingDeadlineSeconds` is still taken into account and may result in a new Job run.
- `Replace`: If it is time for a new Job run and the previous Job run hasn't finished yet, the CronJob replaces the currently running Job run with a new Job run

Note that concurrency policy only applies to the Jobs created by the same CronJob. If there are multiple CronJobs, their respective Jobs are always allowed to run concurrently.

## Schedule suspension

You can suspend execution of Jobs for a CronJob, by setting the optional `.spec.suspend` field to true. The field defaults to false.

This setting does *not* affect Jobs that the CronJob has already started.

If you do set that field to true, all subsequent executions are suspended (they remain scheduled, but the CronJob controller does not start the Jobs to run the tasks) until you unsuspend the CronJob.

**Caution:**

Executions that are suspended during their scheduled time count as missed Jobs. When `.spec.suspend` changes from `true` to `false` on an existing CronJob without a starting deadline, the missed Jobs are scheduled immediately.

## Jobs history limits

The `.spec.successfulJobsHistoryLimit` and `.spec.failedJobsHistoryLimit` fields specify how many completed and failed Jobs should be kept. Both fields are optional.

- `.spec.successfulJobsHistoryLimit`: This field specifies the number of successful finished jobs to keep. The default value is `3`. Setting this field to `0` will not keep any successful jobs.
- `.spec.failedJobsHistoryLimit`: This field specifies the number of failed finished jobs to keep. The default value is `1`. Setting this field to `0` will not keep any failed jobs.

For another way to clean up Jobs automatically, see Clean up finished Jobs automatically.

### Time zones

FEATURE STATE: `Kubernetes v1.27 [stable]`

For CronJobs with no time zone specified, the kube-controller-manager interprets schedules relative to its local time zone.

You can specify a time zone for a CronJob by setting `.spec.timeZone` to the name of a valid time zone. For example, setting `.spec.timeZone: "Etc/UTC"` instructs Kubernetes to interpret the schedule relative to Coordinated Universal Time.

A time zone database from the Go standard library is included in the binaries and used as a fallback in case an external database is not available on the system.

## CronJob limitations

### Unsupported TimeZone specification

Specifying a timezone using `CRON_TZ` or `TZ` variables inside `.spec.schedule` is **not officially supported** (and never has been). If you try to set a schedule that includes `TZ` or `CRON_TZ` timezone specification, Kubernetes will fail to create or update the resource with a validation error. You should specify time zones using the time zone field, instead.

### Modifying a CronJob

By design, a CronJob contains a template for *new* Jobs. If you modify an existing CronJob, the changes you make will apply to new Jobs that start to run after your modification is complete. Jobs (and their Pods) that have already started continue to run without changes. That is, the CronJob does *not* update existing Jobs, even if those remain running.

### Job creation

A CronJob creates a Job object approximately once per execution time of its schedule. The scheduling is approximate because there are certain circumstances where two Jobs might be created, or no Job might be created. Kubernetes tries to avoid those situations, but does not completely prevent them. Therefore, the Jobs that you define should be *idempotent*.

Starting with Kubernetes v1.32, CronJobs apply an annotation `batch.kubernetes.io/cronjob-scheduled-timestamp` to their created Jobs. This annotation indicates the originally scheduled creation time for the Job and is formatted in RFC3339.

If `startingDeadlineSeconds` is set to a large value or left unset (the default) and if `concurrencyPolicy` is set to `Allow`, the Jobs will always run at least once.

**Caution:**

If `startingDeadlineSeconds` is set to a value less than 10 seconds, the CronJob may not be scheduled. This is because the CronJob controller checks things every 10 seconds.

For every CronJob, the CronJob Controller checks how many schedules it missed in the duration from its last scheduled time until now. If there are more than 100 missed schedules, then it does not start the Job and logs the error.

```
Cannot determine if job needs to be started. Too many missed start time (> 100). Set or decrease .spec.startingDeadlineSeconds or c
```

It is important to note that if the `startingDeadlineSeconds` field is set (not `nil`), the controller counts how many missed Jobs occurred from the value of `startingDeadlineSeconds` until now rather than from the last scheduled time until now. For example, if `startingDeadlineSeconds` is `200`, the controller counts how many missed Jobs occurred in the last 200 seconds.

A CronJob is counted as missed if it has failed to be created at its scheduled time. For example, if `concurrencyPolicy` is set to `Forbid` and a CronJob was attempted to be scheduled when there was a previous schedule still running, then it would count as missed.

For example, suppose a CronJob is set to schedule a new Job every one minute beginning at `08:30:00`, and its `startingDeadlineSeconds` field is not set. If the CronJob controller happens to be down from `08:29:00` to `10:21:00`, the Job will not start as the number of missed Jobs which missed their schedule is greater than 100.

To illustrate this concept further, suppose a CronJob is set to schedule a new Job every one minute beginning at `08:30:00`, and its `startingDeadlineSeconds` is set to 200 seconds. If the CronJob controller happens to be down for the same period as the previous example (`08:29:00` to `10:21:00`,) the Job will still start at 10:22:00. This happens as the controller now checks how many missed schedules happened in the last 200 seconds (i.e., 3 missed schedules), rather than from the last scheduled time until now.

The CronJob is only responsible for creating Jobs that match its schedule, and the Job in turn is responsible for the management of the Pods it represents.

## What's next

- Learn about Pods and Jobs, two concepts that CronJobs rely upon.
- Read about the detailed format of CronJob `.spec.schedule` fields.
- For instructions on creating and working with CronJobs, and for an example of a CronJob manifest, see Running automated tasks with CronJobs.
- `CronJob` is part of the Kubernetes REST API. Read the CronJob API reference for more details.

# Sidecar Containers

FEATURE STATE: `Kubernetes v1.33 [stable]` (enabled by default: true)

Sidecar containers are the secondary containers that run along with the main application container within the same Pod. These containers are used to enhance or to extend the functionality of the primary *app container* by providing additional services, or functionality such as logging, monitoring, security, or data synchronization, without directly altering the primary application code.

Typically, you only have one app container in a Pod. For example, if you have a web application that requires a local webserver, the local webserver is a sidecar and the web application itself is the app container.

## Sidecar containers in Kubernetes

Kubernetes implements sidecar containers as a special case of init containers; sidecar containers remain running after Pod startup. This document uses the term *regular init containers* to clearly refer to containers that only run during Pod startup.

Provided that your cluster has the `SidecarContainers` feature gate enabled (the feature is active by default since Kubernetes v1.29), you can specify a `restartPolicy` for containers listed in a Pod's `initContainers` field. These restartable *sidecar* containers are independent from other init containers and from the main application container(s) within the same pod. These can be started, stopped, or restarted without affecting the main application container and other init containers.

You can also run a Pod with multiple containers that are not marked as init or sidecar containers. This is appropriate if the containers within the Pod are required for the Pod to work overall, but you don't need to control which containers start or stop first. You could also do this if you need to support older versions of Kubernetes that don't support a container-level `restartPolicy` field.

### Example application

Here's an example of a Deployment with two containers, one of which is a sidecar:

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

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: myapp  labels:    app: myappspec:  replicas: 1  selector:    matchLabels:      app: myapp  templa
```

## Sidecar containers and Pod lifecycle

If an init container is created with its `restartPolicy` set to `Always`, it will start and remain running during the entire life of the Pod. This can be helpful for running supporting services separated from the main application containers.

If a `readinessProbe` is specified for this init container, its result will be used to determine the `ready` state of the Pod.

Since these containers are defined as init containers, they benefit from the same ordering and sequential guarantees as regular init containers, allowing you to mix sidecar containers with regular init containers for complex Pod initialization flows.

Compared to regular init containers, sidecars defined within `initContainers` continue to run after they have started. This is important when there is more than one entry inside `.spec.initContainers` for a Pod. After a sidecar-style init container is running (the kubelet has set the `started` status for that init container to true), the kubelet then starts the next init container from the ordered `.spec.initContainers` list. That status either becomes true because there is a process running in the container and no startup probe defined, or as a result of its `startupProbe` succeeding.

Upon Pod termination, the kubelet postpones terminating sidecar containers until the main application container has fully stopped. The sidecar containers are then shut down in the opposite order of their appearance in the Pod specification. This approach ensures that the sidecars remain operational, supporting other containers within the Pod, until their service is no longer required.

### Jobs with sidecar containers

If you define a Job that uses sidecar using Kubernetes-style init containers, the sidecar container in each Pod does not prevent the Job from completing after the main container has finished.

Here's an example of a Job with two containers, one of which is a sidecar:

`application/job/job-sidecar.yaml` Copy application/job/job-sidecar.yaml to clipboard

```
apiVersion: batch/v1
kind: Jobmetadata:  name: myjobspec:  template:    spec:      containers:        - name: myjob          image: alpine:latest
```

## Differences from application containers

Sidecar containers run alongside *app containers* in the same pod. However, they do not execute the primary application logic; instead, they provide supporting functionality to the main application.

Sidecar containers have their own independent lifecycles. They can be started, stopped, and restarted independently of app containers. This means you can update, scale, or maintain sidecar containers without affecting the primary application.

Sidecar containers share the same network and storage namespaces with the primary container. This co-location allows them to interact closely and share resources.

From a Kubernetes perspective, the sidecar container's graceful termination is less important. When other containers take all allotted graceful termination time, the sidecar containers will receive the `SIGTERM` signal, followed by the `SIGKILL` signal, before they have time to terminate gracefully. So exit codes different from `0` (`0` indicates successful exit), for sidecar containers are normal on Pod termination and should be generally ignored by the external tooling.

## Differences from init containers

Sidecar containers work alongside the main container, extending its functionality and providing additional services.

Sidecar containers run concurrently with the main application container. They are active throughout the lifecycle of the pod and can be started and stopped independently of the main container. Unlike init containers, sidecar containers support probes to control their lifecycle.

Sidecar containers can interact directly with the main application containers, because like init containers they always share the same network, and can optionally also share volumes (filesystems).

Init containers stop before the main containers start up, so init containers cannot exchange messages with the app container in a Pod. Any data passing is one-way (for example, an init container can put information inside an `emptyDir` volume).

Changing the image of a sidecar container will not cause the Pod to restart, but will trigger a container restart.

## Resource sharing within containers

Given the order of execution for init, sidecar and app containers, the following rules for resource usage apply:

- The highest of any particular resource request or limit defined on all init containers is the *effective init request/limit*. If any resource has no resource limit specified this is considered as the highest limit.
- The Pod's *effective request/limit* for a resource is the sum of pod overhead and the higher of:
  - the sum of all non-init containers(app and sidecar containers) request/limit for a resource
  - the effective init request/limit for a resource
- Scheduling is done based on effective requests/limits, which means init containers can reserve resources for initialization that are not used during the life of the Pod.
- The QoS (quality of service) tier of the Pod's *effective QoS tier* is the QoS tier for all init, sidecar and app containers alike.

Quota and limits are applied based on the effective Pod request and limit.

### Sidecar containers and Linux cgroups

On Linux, resource allocations for Pod level control groups (cgroups) are based on the effective Pod request and limit, the same as the scheduler.

## What's next

- Learn how to Adopt Sidecar Containers
- Read a blog post on native sidecar containers.
- Read about creating a Pod that has an init container.
- Learn about the types of probes: liveness, readiness, startup probe.
- Learn about pod overhead.

# Workload Management

Kubernetes provides several built-in APIs for declarative management of your workloads and the components of those workloads.

Ultimately, your applications run as containers inside Pods; however, managing individual Pods would be a lot of effort. For example, if a Pod fails, you probably want to run a new Pod to replace it. Kubernetes can do that for you.

You use the Kubernetes API to create a workload object that represents a higher abstraction level than a Pod, and then the Kubernetes control plane automatically manages Pod objects on your behalf, based on the specification for the workload object you defined.

The built-in APIs for managing workloads are:

Deployment (and, indirectly, ReplicaSet), the most common way to run an application on your cluster. Deployment is a good fit for managing a stateless application workload on your cluster, where any Pod in the Deployment is interchangeable and can be replaced if needed. (Deployments are a replacement for the legacy ReplicationController API).

A StatefulSet lets you manage one or more Pods – all running the same application code – where the Pods rely on having a distinct identity. This is different from a Deployment where the Pods are expected to be interchangeable. The most common use for a StatefulSet is to be able to make a link between its Pods and their persistent storage. For example, you can run a StatefulSet that associates each Pod with a PersistentVolume. If one of the Pods in the StatefulSet fails, Kubernetes makes a replacement Pod that is connected to the same PersistentVolume.

A DaemonSet defines Pods that provide facilities that are local to a specific node; for example, a driver that lets containers on that node access a storage system. You use a DaemonSet when the driver, or other node-level service, has to run on the node where it's useful. Each Pod in a DaemonSet performs a role similar to a system daemon on a classic Unix / POSIX server. A DaemonSet might be fundamental to the operation of your cluster, such as a plugin to let that node access cluster networking, it might help you to manage the node, or it could provide less essential facilities that enhance the container platform you are running. You can run DaemonSets (and their pods) across every node in your cluster, or across just a subset (for example, only install the GPU accelerator driver on nodes that have a GPU installed).

You can use a Job and / or a CronJob to define tasks that run to completion and then stop. A Job represents a one-off task, whereas each CronJob repeats according to a schedule.

Other topics in this section:

- Automatic Cleanup for Finished Jobs
- ReplicationController

# Ephemeral Containers

FEATURE STATE: `Kubernetes v1.25 [stable]`

This page provides an overview of ephemeral containers: a special type of container that runs temporarily in an existing Pod to accomplish user-initiated actions such as troubleshooting. You use ephemeral containers to inspect services rather than to build applications.

## Understanding ephemeral containers

Pods are the fundamental building block of Kubernetes applications. Since Pods are intended to be disposable and replaceable, you cannot add a container to a Pod once it has been created. Instead, you usually delete and replace Pods in a controlled fashion using deployments.

Sometimes it's necessary to inspect the state of an existing Pod, however, for example to troubleshoot a hard-to-reproduce bug. In these cases you can run an ephemeral container in an existing Pod to inspect its state and run arbitrary commands.

### What is an ephemeral container?

Ephemeral containers differ from other containers in that they lack guarantees for resources or execution, and they will never be automatically restarted, so they are not appropriate for building applications. Ephemeral containers are described using the same `ContainerSpec` as regular containers, but many fields are incompatible and disallowed for ephemeral containers.

- Ephemeral containers may not have ports, so fields such as `ports`, `livenessProbe`, `readinessProbe` are disallowed.
- Pod resource allocations are immutable, so setting `resources` is disallowed.
- For a complete list of allowed fields, see the EphemeralContainer reference documentation.

Ephemeral containers are created using a special `ephemeralcontainers` handler in the API rather than by adding them directly to `pod.spec`, so it's not possible to add an ephemeral container using `kubectl edit`.

Like regular containers, you may not change or remove an ephemeral container after you have added it to a Pod.

**Note:**

Ephemeral containers are not supported by static pods.

## Uses for ephemeral containers

Ephemeral containers are useful for interactive troubleshooting when `kubectl exec` is insufficient because a container has crashed or a container image doesn't include debugging utilities.

In particular, distroless images enable you to deploy minimal container images that reduce attack surface and exposure to bugs and vulnerabilities. Since distroless images do not include a shell or any debugging utilities, it's difficult to troubleshoot distroless images using `kubectl exec` alone.

When using ephemeral containers, it's helpful to enable process namespace sharing so you can view processes in other containers.

## What's next

- Learn how to debug pods using ephemeral containers.

# Pod Hostname

This page explains how to set a Pod's hostname, potential side effects after configuration, and the underlying mechanics.

## Default Pod hostname

When a Pod is created, its hostname (as observed from within the Pod) is derived from the Pod's metadata.name value. Both the hostname and its corresponding fully qualified domain name (FQDN) are set to the metadata.name value (from the Pod's perspective)

```
apiVersion: v1
kind: Podmetadata:  name: busybox-1spec:  containers:  - image: busybox:1.28    command:      - sleep      - "3600"    name: busyb
```

The Pod created by this manifest will have its hostname and fully qualified domain name (FQDN) set to `busybox-1`.

## Hostname with pod's hostname and subdomain fields

The Pod spec includes an optional `hostname` field. When set, this value takes precedence over the Pod's `metadata.name` as the hostname (observed from within the Pod). For example, a Pod with spec.hostname set to `my-host` will have its hostname set to `my-host`.

The Pod spec also includes an optional `subdomain` field, indicating the Pod belongs to a subdomain within its namespace. If a Pod has `spec.hostname` set to "foo" and spec.subdomain set to "bar" in the namespace `my-namespace`, its hostname becomes `foo` and its fully qualified domain name (FQDN) becomes `foo.bar.my-namespace.svc.cluster-domain.example` (observed from within the Pod).

When both hostname and subdomain are set, the cluster's DNS server will create A and/or AAAA records based on these fields. Refer to: Pod's hostname and subdomain fields.

## Hostname with pod's setHostnameAsFQDN fields

FEATURE STATE: `Kubernetes v1.22 [stable]`

When a Pod is configured to have fully qualified domain name (FQDN), its hostname is the short hostname. For example, if you have a Pod with the fully qualified domain name `busybox-1.busybox-subdomain.my-namespace.svc.cluster-domain.example`, then by default the `hostname` command inside that Pod returns `busybox-1` and the `hostname --fqdn` command returns the FQDN.

When both `setHostnameAsFQDN: true` and the subdomain field is set in the Pod spec, the kubelet writes the Pod's FQDN into the hostname for that Pod's namespace. In this case, both `hostname` and `hostname --fqdn` return the Pod's FQDN.

The Pod's FQDN is constructed in the same manner as previously defined. It is composed of the Pod's `spec.hostname` (if specified) or `metadata.name` field, the `spec.subdomain`, the `namespace` name, and the cluster domain suffix.

**Note:**

In Linux, the hostname field of the kernel (the `nodename` field of `struct utsname`) is limited to 64 characters.

If a Pod enables this feature and its FQDN is longer than 64 character, it will fail to start. The Pod will remain in `Pending` status (`ContainerCreating` as seen by `kubectl`) generating error events, such as "Failed to construct FQDN from Pod hostname and cluster domain".

This means that when using this field, you must ensure the combined length of the Pod's `metadata.name` (or `spec.hostname`) and `spec.subdomain` fields results in an FQDN that does not exceed 64 characters.

## Hostname with pod's hostnameOverride

FEATURE STATE: `Kubernetes v1.34 [alpha]` (enabled by default: false)

Setting a value for `hostnameOverride` in the Pod spec causes the kubelet to unconditionally set both the Pod's hostname and fully qualified domain name (FQDN) to the `hostnameOverride` value.

The `hostnameOverride` field has a length limitation of 64 characters and must adhere to the DNS subdomain names standard defined in RFC 1123.

Example:

```
apiVersion: v1
kind: Podmetadata:  name: busybox-2-busybox-example-domainspec:  hostnameOverride: busybox-2.busybox.example.domain  containers:  
```

**Note:**

This only affects the hostname within the Pod; it does not affect the Pod's A or AAAA records in the cluster DNS server.

If `hostnameOverride` is set alongside `hostname` and `subdomain` fields:

- The hostname inside the Pod is overridden to the `hostnameOverride` value.
- The Pod's A and/or AAAA records in the cluster DNS server are still generated based on the `hostname` and `subdomain` fields.

Note: If `hostnameOverride` is set, you cannot simultaneously set the `hostNetwork` and `setHostnameAsFQDN` fields. The API server will explicitly reject any create request attempting this combination.

For details on behavior when `hostnameOverride` is set in combination with other fields (hostname, subdomain, setHostnameAsFQDN, hostNetwork), see the table in the KEP-4762 design details.

---

# ReplicaSet

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. Usually, you define a Deployment and let that Deployment manage ReplicaSets automatically.

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.

## How a ReplicaSet works

A ReplicaSet is defined with fields, including a selector that specifies how to identify Pods it can acquire, a number of replicas indicating how many Pods it should be maintaining, and a pod template specifying the data of new Pods it should create to meet the number of replicas criteria. A ReplicaSet then fulfills its purpose by creating and deleting Pods as needed to reach the desired number. When a ReplicaSet needs to create new Pods, it uses its Pod template.

A ReplicaSet is linked to its Pods via the Pods' metadata.ownerReferences field, which specifies what resource the current object is owned by. All Pods acquired by a ReplicaSet have their owning ReplicaSet's identifying information within their ownerReferences field. It's through this link that the ReplicaSet knows of the state of the Pods it is maintaining and plans accordingly.

A ReplicaSet identifies new Pods to acquire by using its selector. If there is a Pod that has no OwnerReference or the OwnerReference is not a Controller and it matches a ReplicaSet's selector, it will be immediately acquired by said ReplicaSet.

## When to use a ReplicaSet

A ReplicaSet ensures that a specified number of pod replicas are running at any given time. However, a Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features. Therefore, we recommend using Deployments instead of directly using ReplicaSets, unless you require custom update orchestration or don't require updates at all.

This actually means that you may never need to manipulate ReplicaSet objects: use a Deployment instead, and define your application in the spec section.

## Example

`controllers/frontend.yaml` Copy controllers/frontend.yaml to clipboard

```
apiVersion: apps/v1
kind: ReplicaSetmetadata:  name: frontend  labels:    app: guestbook    tier: frontendspec:  # modify replicas according to your c
```

Saving this manifest into `frontend.yaml` and submitting it to a Kubernetes cluster will create the defined ReplicaSet and the Pods that it manages.

```
kubectl apply -f https://kubernetes.io/examples/controllers/frontend.yaml
```

You can then get the current ReplicaSets deployed:

```
kubectl get rs
```

And see the frontend one you created:

```
NAME       DESIRED   CURRENT   READY   AGE
frontend   3         3         3       6s
```

You can also check on the state of the ReplicaSet:

```
kubectl describe rs/frontend
```

And you will see output similar to:

```
Name:         frontend
Namespace:    default
Selector:     tier=frontend
Labels:       app=guestbook
              tier=frontend
Annotations:  <none>
Replicas:     3 current / 3 desired
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  tier=frontend
  Containers:
   php-redis:
    Image:        us-docker.pkg.dev/google-samples/containers/gke/gb-frontend:v5
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From                   Message
  ----    ------            ----  ----                   -------
  Normal  SuccessfulCreate  13s   replicaset-controller  Created pod: frontend-gbgfx
  Normal  SuccessfulCreate  13s   replicaset-controller  Created pod: frontend-rwz57
  Normal  SuccessfulCreate  13s   replicaset-controller  Created pod: frontend-wkl7w
```

And lastly you can check for the Pods brought up:

```
kubectl get pods
```

You should see Pod information similar to:

```
NAME             READY   STATUS    RESTARTS   AGE
frontend-gbgfx   1/1     Running   0          10m
frontend-rwz57   1/1     Running   0          10m
frontend-wkl7w   1/1     Running   0          10m
```

You can also verify that the owner reference of these pods is set to the frontend ReplicaSet. To do this, get the yaml of one of the Pods running:

```
kubectl get pods frontend-gbgfx -o yaml
```

The output will look similar to this, with the frontend ReplicaSet's info set in the metadata's ownerReferences field:

```
apiVersion: v1
kind: Podmetadata:  creationTimestamp: "2024-02-28T22:30:44Z"  generateName: frontend-  labels:    tier: frontend  name: frontend-
```

## Non-Template Pod acquisitions

While you can create bare Pods with no problems, it is strongly recommended to make sure that the bare Pods do not have labels which match the selector of one of your ReplicaSets. The reason for this is because a ReplicaSet is not limited to owning Pods specified by its template-- it can acquire other Pods in the manner specified in the previous sections.

Take the previous frontend ReplicaSet example, and the Pods specified in the following manifest:

[pods/pod-rs.yaml](pods/pod-rs.yaml) Copy pods/pod-rs.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: pod1  labels:    tier: frontendspec:  containers:  - name: hello1    image: gcr.io/google-samples/hello-a
```

As those Pods do not have a Controller (or any object) as their owner reference and match the selector of the frontend ReplicaSet, they will immediately be acquired by it.

Suppose you create the Pods after the frontend ReplicaSet has been deployed and has set up its initial Pod replicas to fulfill its replica count requirement:

```
kubectl apply -f https://kubernetes.io/examples/pods/pod-rs.yaml
```

The new Pods will be acquired by the ReplicaSet, and then immediately terminated as the ReplicaSet would be over its desired count.

Fetching the Pods:

```
kubectl get pods
```

The output shows that the new Pods are either already terminated, or in the process of being terminated:

```
NAME             READY   STATUS        RESTARTS   AGE
frontend-b2zdv   1/1     Running       0          10m
frontend-vcmts   1/1     Running       0          10m
frontend-wtsmm   1/1     Running       0          10m
pod1             0/1     Terminating   0          1s
pod2             0/1     Terminating   0          1s
```

If you create the Pods first:

```
kubectl apply -f https://kubernetes.io/examples/pods/pod-rs.yaml
```

And then create the ReplicaSet however:

```
kubectl apply -f https://kubernetes.io/examples/controllers/frontend.yaml
```

You shall see that the ReplicaSet has acquired the Pods and has only created new ones according to its spec until the number of its new Pods and the original matches its desired count. As fetching the Pods:

```
kubectl get pods
```

Will reveal in its output:

```
NAME             READY   STATUS    RESTARTS   AGE
frontend-hmmj2   1/1     Running   0          9s
pod1             1/1     Running   0          36s
pod2             1/1     Running   0          36s
```

In this manner, a ReplicaSet can own a non-homogeneous set of Pods

## Writing a ReplicaSet manifest

As with all other Kubernetes API objects, a ReplicaSet needs the `apiVersion`, `kind`, and `metadata` fields. For ReplicaSets, the `kind` is always a ReplicaSet.

When the control plane creates new Pods for a ReplicaSet, the `.metadata.name` of the ReplicaSet is part of the basis for naming those Pods. The name of a ReplicaSet must be a valid DNS subdomain value, but this can produce unexpected results for the Pod hostnames. For best compatibility, the name should follow the more restrictive rules for a DNS label.

A ReplicaSet also needs a `.spec` section.

### Pod Template

The `.spec.template` is a pod template which is also required to have labels in place. In our `frontend.yaml` example we had one label: `tier: frontend`. Be careful not to overlap with the selectors of other controllers, lest they try to adopt this Pod.

For the template's restart policy field, `.spec.template.spec.restartPolicy`, the only allowed value is `Always`, which is the default.

### Pod Selector

The `.spec.selector` field is a label selector. As discussed earlier these are the labels used to identify potential Pods to acquire. In our `frontend.yaml` example, the selector was:

```
matchLabels:
  tier: frontend
```

In the ReplicaSet, `.spec.template.metadata.labels` must match `spec.selector`, or it will be rejected by the API.

**Note:**

For 2 ReplicaSets specifying the same `.spec.selector` but different `.spec.template.metadata.labels` and `.spec.template.spec` fields, each ReplicaSet ignores the Pods created by the other ReplicaSet.

### Replicas

You can specify how many Pods should run concurrently by setting `.spec.replicas`. The ReplicaSet will create/delete its Pods to match this number.

If you do not specify `.spec.replicas`, then it defaults to 1.

## Working with ReplicaSets

### Deleting a ReplicaSet and its Pods

To delete a ReplicaSet and all of its Pods, use `kubectl delete`. The Garbage collector automatically deletes all of the dependent Pods by default.

When using the REST API or the `client-go` library, you must set `propagationPolicy` to `Background` or `Foreground` in the `-d` option. For example:

```
kubectl proxy --port=8080
curl -X DELETE  'localhost:8080/apis/apps/v1/namespaces/default/replicasets/frontend' \
  -d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Foreground"}' \  -H "Content-Type: application/json"
```

### Deleting just a ReplicaSet

You can delete a ReplicaSet without affecting any of its Pods using `kubectl delete` with the `--cascade=orphan` option. When using the REST API or the `client-go` library, you must set `propagationPolicy` to `Orphan`. For example:

```
kubectl proxy --port=8080
curl -X DELETE  'localhost:8080/apis/apps/v1/namespaces/default/replicasets/frontend' \
  -d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Orphan"}' \  -H "Content-Type: application/json"
```

Once the original is deleted, you can create a new ReplicaSet to replace it. As long as the old and new `.spec.selector` are the same, then the new one will adopt the old Pods. However, it will not make any effort to make existing Pods match a new, different pod template. To update Pods to a new spec in a controlled way, use a Deployment, as ReplicaSets do not support a rolling update directly.

### Terminating Pods

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

You can enable this feature by setting the `DeploymentReplicaSetTerminatingReplicas` feature gate on the API server and on the kube-controller-manager

Pods that become terminating due to deletion or scale down may take a long time to terminate, and may consume additional resources during that period. As a result, the total number of all pods can temporarily exceed `.spec.replicas`. Terminating pods can be tracked using the `.status.terminatingReplicas` field of the ReplicaSet.

### Isolating Pods from a ReplicaSet

You can remove Pods from a ReplicaSet by changing their labels. This technique may be used to remove Pods from service for debugging, data recovery, etc. Pods that are removed in this way will be replaced automatically ( assuming that the number of replicas is not also changed).

### Scaling a ReplicaSet

A ReplicaSet can be easily scaled up or down by simply updating the `.spec.replicas` field. The ReplicaSet controller ensures that a desired number of Pods with a matching label selector are available and operational.

When scaling down, the ReplicaSet controller chooses which pods to delete by sorting the available pods to prioritize scaling down pods based on the following general algorithm:

1. Pending (and unschedulable) pods are scaled down first
2. If `controller.kubernetes.io/pod-deletion-cost` annotation is set, then the pod with the lower value will come first.
3. Pods on nodes with more replicas come before pods on nodes with fewer replicas.
4. If the pods' creation times differ, the pod that was created more recently comes before the older pod (the creation times are bucketed on an integer log scale).

If all of the above match, then selection is random.

### Pod deletion cost

FEATURE STATE: `Kubernetes v1.22 [beta]`

Using the `controller.kubernetes.io/pod-deletion-cost` annotation, users can set a preference regarding which pods to remove first when downscaling a ReplicaSet.

The annotation should be set on the pod, the range is [-2147483648, 2147483647]. It represents the cost of deleting a pod compared to other pods belonging to the same ReplicaSet. Pods with lower deletion cost are preferred to be deleted before pods with higher deletion cost.

The implicit value for this annotation for pods that don't set it is 0; negative values are permitted. Invalid values will be rejected by the API server.

This feature is beta and enabled by default. You can disable it using the feature gate `PodDeletionCost` in both kube-apiserver and kube-controller-manager.

**Note:**

- This is honored on a best-effort basis, so it does not offer any guarantees on pod deletion order.
- Users should avoid updating the annotation frequently, such as updating it based on a metric value, because doing so will generate a significant number of pod updates on the apiserver.

#### Example Use Case

The different pods of an application could have different utilization levels. On scale down, the application may prefer to remove the pods with lower utilization. To avoid frequently updating the pods, the application should update `controller.kubernetes.io/pod-deletion-cost` once before issuing a scale down (setting the annotation to a value proportional to pod utilization level). This works if the application itself controls the down scaling; for example, the driver pod of a Spark deployment.

### ReplicaSet as a Horizontal Pod Autoscaler Target

A ReplicaSet can also be a target for Horizontal Pod Autoscalers (HPA). That is, a ReplicaSet can be auto-scaled by an HPA. Here is an example HPA targeting the ReplicaSet we created in the previous example.

`controllers/hpa-rs.yaml` Copy controllers/hpa-rs.yaml to clipboard

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscalermetadata:  name: frontend-scalerspec:  scaleTargetRef:    apiVersion: apps/v1    kind: ReplicaSet
```

Saving this manifest into `hpa-rs.yaml` and submitting it to a Kubernetes cluster should create the defined HPA that autoscales the target ReplicaSet depending on the CPU usage of the replicated Pods.

```
kubectl apply -f https://k8s.io/examples/controllers/hpa-rs.yaml
```

Alternatively, you can use the `kubectl autoscale` command to accomplish the same (and it's easier!)

```
kubectl autoscale rs frontend --max=10 --min=3 --cpu=50%
```

## Alternatives to ReplicaSet

### Deployment (recommended)

`Deployment` is an object which can own ReplicaSets and update them and their Pods via declarative, server-side rolling updates. While ReplicaSets can be used independently, today they're mainly used by Deployments as a mechanism to orchestrate Pod creation, deletion and updates. When you use Deployments you don't have to worry about managing the ReplicaSets that they create. Deployments own and manage their ReplicaSets. As such, it is recommended to use Deployments when you want ReplicaSets.

### Bare Pods

Unlike the case where a user directly created Pods, a ReplicaSet replaces Pods that are deleted or terminated for any reason, such as in the case of node failure or disruptive node maintenance, such as a kernel upgrade. For this reason, we recommend that you use a ReplicaSet even if your application requires only a single Pod. Think of it similarly to a process supervisor, only it supervises multiple Pods across multiple nodes instead of individual processes on a single node. A ReplicaSet delegates local container restarts to some agent on the node such as Kubelet.

### Job

Use a `Job` instead of a ReplicaSet for Pods that are expected to terminate on their own (that is, batch jobs).

### DaemonSet

Use a `DaemonSet` instead of a ReplicaSet for Pods that provide a machine-level function, such as machine monitoring or machine logging. These Pods have a lifetime that is tied to a machine lifetime: the Pod needs to be running on the machine before other Pods start, and are safe to terminate when the machine is otherwise ready to be rebooted/shutdown.

### ReplicationController

ReplicaSets are the successors to ReplicationControllers. The two serve the same purpose, and behave similarly, except that a ReplicationController does not support set-based selector requirements as described in the labels user guide. As such, ReplicaSets are preferred over ReplicationControllers

## What's next

- Learn about Pods.
- Learn about Deployments.
- Run a Stateless Application Using a Deployment, which relies on ReplicaSets to work.
- `ReplicaSet` is a top-level resource in the Kubernetes REST API. Read the ReplicaSet object definition to understand the API for replica sets.
- Read about PodDisruptionBudget and how you can use it to manage application availability during disruptions.

# Workloads

Understand Pods, the smallest deployable compute object in Kubernetes, and the higher-level abstractions that help you to run them.

A workload is an application running on Kubernetes. Whether your workload is a single component or several that work together, on Kubernetes you run it inside a set of *pods*. In Kubernetes, a Pod represents a set of running containers on your cluster.

Kubernetes pods have a defined lifecycle. For example, once a pod is running in your cluster then a critical fault on the node where that pod is running means that all the pods on that node fail. Kubernetes treats that level of failure as final: you would need to create a new Pod to recover, even if the node later becomes healthy.

However, to make life considerably easier, you don't need to manage each Pod directly. Instead, you can use *workload resources* that manage a set of pods on your behalf. These resources configure controllers that make sure the right number of the right kind of pod are running, to match the state you specified.

Kubernetes provides several built-in workload resources:

- Deployment and ReplicaSet (replacing the legacy resource ReplicationController). Deployment is a good fit for managing a stateless application workload on your cluster, where any Pod in the Deployment is interchangeable and can be replaced if needed.
- StatefulSet lets you run one or more related Pods that do track state somehow. For example, if your workload records data persistently, you can run a StatefulSet that matches each Pod with a PersistentVolume. Your code, running in the Pods for that StatefulSet, can replicate data to other Pods in the same StatefulSet to improve overall resilience.
- DaemonSet defines Pods that provide facilities that are local to nodes. Every time you add a node to your cluster that matches the specification in a DaemonSet, the control plane schedules a Pod for that DaemonSet onto the new node. Each pod in a DaemonSet performs a job similar to a system daemon on a classic Unix / POSIX server. A DaemonSet might be fundamental to the operation of your cluster, such as a plugin to run cluster networking, it might help you to manage the node, or it could provide optional behavior that enhances the container platform you are running.
- Job and CronJob provide different ways to define tasks that run to completion and then stop. You can use a Job to define a task that runs to completion, just once. You can use a CronJob to run the same Job multiple times according a schedule.

In the wider Kubernetes ecosystem, you can find third-party workload resources that provide additional behaviors. Using a custom resource definition, you can add in a third-party workload resource if you want a specific behavior that's not part of Kubernetes' core. For example, if you wanted to run a group of Pods for your application but stop work unless *all* the Pods are available (perhaps for some high-throughput distributed task), then you can implement or install an extension that does provide that feature.

## What's next

As well as reading about each API kind for workload management, you can read how to do specific tasks:

- Run a stateless application using a Deployment
- Run a stateful application either as a single instance or as a replicated set
- Run automated tasks with a CronJob

To learn about Kubernetes' mechanisms for separating code from configuration, visit Configuration.

There are two supporting concepts that provide backgrounds about how Kubernetes manages pods for applications:

- Garbage collection tidies up objects from your cluster after their *owning resource* has been removed.
- The *time-to-live after finished* controller removes Jobs once a defined time has passed since they completed.

Once your application is running, you might want to make it available on the internet as a Service or, for web application only, using an Ingress.

---

# StatefulSets

A StatefulSet runs a group of Pods, and maintains a sticky identity for each of those Pods. This is useful for managing applications that need persistent storage or a stable, unique network identity.

StatefulSet is the workload API object used to manage stateful applications.

Manages the deployment and scaling of a set of Pods, *and provides guarantees about the ordering and uniqueness* of these Pods.

Like a Deployment, a StatefulSet manages Pods that are based on an identical container spec. Unlike a Deployment, a StatefulSet maintains a sticky identity for each of its Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.

If you want to use storage volumes to provide persistence for your workload, you can use a StatefulSet as part of the solution. Although individual Pods in a StatefulSet are susceptible to failure, the persistent Pod identifiers make it easier to match existing volumes to the new Pods that replace any that have failed.

## Using StatefulSets

StatefulSets are valuable for applications that require one or more of the following:

- Stable, unique network identifiers.
- Stable, persistent storage.
- Ordered, graceful deployment and scaling.
- Ordered, automated rolling updates.

In the above, stable is synonymous with persistence across Pod (re)scheduling. If an application doesn't require any stable identifiers or ordered deployment, deletion, or scaling, you should deploy your application using a workload object that provides a set of stateless replicas. Deployment or ReplicaSet may be better suited to your stateless needs.

## Limitations

- The storage for a given Pod must either be provisioned by a PersistentVolume Provisioner based on the requested *storage class*, or pre-provisioned by an admin.
- Deleting and/or scaling a StatefulSet down will *not* delete the volumes associated with the StatefulSet. This is done to ensure data safety, which is generally more valuable than an automatic purge of all related StatefulSet resources.

- StatefulSets currently require a Headless Service to be responsible for the network identity of the Pods. You are responsible for creating this Service.
- StatefulSets do not provide any guarantees on the termination of pods when a StatefulSet is deleted. To achieve ordered and graceful termination of the pods in the StatefulSet, it is possible to scale the StatefulSet down to 0 prior to deletion.
- When using Rolling Updates with the default Pod Management Policy (`OrderedReady`), it's possible to get into a broken state that requires manual intervention to repair.

# Components

The example below demonstrates the components of a StatefulSet.

```
apiVersion: v1
kind: Servicemetadata:  name: nginx  labels:    app: nginxspec:  ports:  - port: 80    name: web  clusterIP: None  selector:    ap
```

**Note:**

This example uses the `ReadWriteOnce` access mode, for simplicity. For production use, the Kubernetes project recommends using the `ReadWriteOncePod` access mode instead.

In the above example:

- A Headless Service, named `nginx`, is used to control the network domain.
- The StatefulSet, named `web`, has a Spec that indicates that 3 replicas of the nginx container will be launched in unique Pods.
- The `volumeClaimTemplates` will provide stable storage using PersistentVolumes provisioned by a PersistentVolume Provisioner.

The name of a StatefulSet object must be a valid DNS label.

## Pod Selector

You must set the `.spec.selector` field of a StatefulSet to match the labels of its `.spec.template.metadata.labels`. Failing to specify a matching Pod Selector will result in a validation error during StatefulSet creation.

## Volume Claim Templates

You can set the `.spec.volumeClaimTemplates` field to create a PersistentVolumeClaim. This will provide stable storage to the StatefulSet if either:

- The StorageClass specified for the volume claim is set up to use dynamic provisioning.
- The cluster already contains a PersistentVolume with the correct StorageClass and sufficient available storage space.

## Minimum ready seconds

FEATURE STATE: `Kubernetes v1.25 [stable]`

`.spec.minReadySeconds` is an optional field that specifies the minimum number of seconds for which a newly created Pod should be running and ready without any of its containers crashing, for it to be considered available. This is used to check progression of a rollout when using a Rolling Update strategy. This field defaults to 0 (the Pod will be considered available as soon as it is ready). To learn more about when a Pod is considered ready, see Container Probes.

# Pod Identity

StatefulSet Pods have a unique identity that consists of an ordinal, a stable network identity, and stable storage. The identity sticks to the Pod, regardless of which node it's (re)scheduled on.

## Ordinal Index

For a StatefulSet with N replicas, each Pod in the StatefulSet will be assigned an integer ordinal, that is unique over the Set. By default, pods will be assigned ordinals from 0 up through N-1. The StatefulSet controller will also add a pod label with this index: `apps.kubernetes.io/pod-index`.

## Start ordinal

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

`.spec.ordinals` is an optional field that allows you to configure the integer ordinals assigned to each Pod. It defaults to nil. Within the field, you can configure the following options:

- `.spec.ordinals.start`: If the `.spec.ordinals.start` field is set, Pods will be assigned ordinals from `.spec.ordinals.start` up through `.spec.ordinals.start + .spec.replicas - 1`.

## Stable Network ID

Each Pod in a StatefulSet derives its hostname from the name of the StatefulSet and the ordinal of the Pod. The pattern for the constructed hostname is `$(statefulset name)-$(ordinal)`. The example above will create three Pods named `web-0,web-1,web-2`. A StatefulSet can use a Headless Service to control the domain of its Pods. The domain managed by this Service takes the form: `$(service name).$(namespace).svc.cluster.local`, where "cluster.local" is the cluster domain. As each Pod is created, it gets a matching DNS subdomain, taking the form: `$(podname).$(governing service domain)`, where the governing service is defined by the `serviceName` field on the StatefulSet.

Depending on how DNS is configured in your cluster, you may not be able to look up the DNS name for a newly-run Pod immediately. This behavior can occur when other clients in the cluster have already sent queries for the hostname of the Pod before it was created. Negative caching (normal in DNS) means that the results of previous failed lookups are remembered and reused, even after the Pod is running, for at least a few seconds.

If you need to discover Pods promptly after they are created, you have a few options:

- Query the Kubernetes API directly (for example, using a watch) rather than relying on DNS lookups.
- Decrease the time of caching in your Kubernetes DNS provider (typically this means editing the config map for CoreDNS, which currently caches for 30 seconds).

As mentioned in the limitations section, you are responsible for creating the Headless Service responsible for the network identity of the pods.

Here are some examples of choices for Cluster Domain, Service name, StatefulSet name, and how that affects the DNS names for the StatefulSet's Pods.

| Cluster Domain | Service (ns/name) | StatefulSet (ns/name) | StatefulSet Domain | Pod DNS | Pod Hostname |
|---|---|---|---|---|---|
| cluster.local | default/nginx | default/web | nginx.default.svc.cluster.local | web-{0..N-1}.nginx.default.svc.cluster.local | web-{0..N-1} |
| cluster.local | foo/nginx | foo/web | nginx.foo.svc.cluster.local | web-{0..N-1}.nginx.foo.svc.cluster.local | web-{0..N-1} |
| kube.local | foo/nginx | foo/web | nginx.foo.svc.kube.local | web-{0..N-1}.nginx.foo.svc.kube.local | web-{0..N-1} |

**Note:**

Cluster Domain will be set to `cluster.local` unless otherwise configured.

### Stable Storage

For each VolumeClaimTemplate entry defined in a StatefulSet, each Pod receives one PersistentVolumeClaim. In the nginx example above, each Pod receives a single PersistentVolume with a StorageClass of `my-storage-class` and 1 GiB of provisioned storage. If no StorageClass is specified, then the default StorageClass will be used. When a Pod is (re)scheduled onto a node, its `volumeMounts` mount the PersistentVolumes associated with its PersistentVolume Claims. Note that, the PersistentVolumes associated with the Pods' PersistentVolume Claims are not deleted when the Pods, or StatefulSet are deleted. This must be done manually.

### Pod Name Label

When the StatefulSet controller creates a Pod, it adds a label, `statefulset.kubernetes.io/pod-name`, that is set to the name of the Pod. This label allows you to attach a Service to a specific Pod in the StatefulSet.

### Pod index label

FEATURE STATE: `Kubernetes v1.32 [stable]` (enabled by default: true)

When the StatefulSet controller creates a Pod, the new Pod is labelled with `apps.kubernetes.io/pod-index`. The value of this label is the ordinal index of the Pod. This label allows you to route traffic to a particular pod index, filter logs/metrics using the pod index label, and more. Note the feature gate `PodIndexLabel` is enabled and locked by default for this feature, in order to disable it, users will have to use server emulated version v1.31.

## Deployment and Scaling Guarantees

- For a StatefulSet with N replicas, when Pods are being deployed, they are created sequentially, in order from {0..N-1}.
- When Pods are being deleted, they are terminated in reverse order, from {N-1..0}.
- Before a scaling operation is applied to a Pod, all of its predecessors must be Running and Ready.
- Before a Pod is terminated, all of its successors must be completely shutdown.

The StatefulSet should not specify a `pod.Spec.TerminationGracePeriodSeconds` of 0. This practice is unsafe and strongly discouraged. For further explanation, please refer to force deleting StatefulSet Pods.

When the nginx example above is created, three Pods will be deployed in the order web-0, web-1, web-2. web-1 will not be deployed before web-0 is Running and Ready, and web-2 will not be deployed until web-1 is Running and Ready. If web-0 should fail, after web-1 is Running and Ready, but before web-2 is launched, web-2 will not be launched until web-0 is successfully relaunched and becomes Running and Ready.

If a user were to scale the deployed example by patching the StatefulSet such that `replicas=1`, web-2 would be terminated first. web-1 would not be terminated until web-2 is fully shutdown and deleted. If web-0 were to fail after web-2 has been terminated and is completely shutdown, but prior to web-1's termination, web-1 would not be terminated until web-0 is Running and Ready.

### Pod Management Policies

StatefulSet allows you to relax its ordering guarantees while preserving its uniqueness and identity guarantees via its `.spec.podManagementPolicy` field.

#### OrderedReady Pod Management

`OrderedReady` pod management is the default for StatefulSets. It implements the behavior described in Deployment and Scaling Guarantees.

#### Parallel Pod Management

`Parallel` pod management tells the StatefulSet controller to launch or terminate all Pods in parallel, and to not wait for Pods to become Running and Ready or completely terminated prior to launching or terminating another Pod. This option only affects the behavior for scaling operations. Updates are not affected.

## Update strategies

A StatefulSet's `.spec.updateStrategy` field allows you to configure and disable automated rolling updates for containers, labels, resource request/limits, and annotations for the Pods in a StatefulSet. There are two possible values:

`OnDelete`

When a StatefulSet's `.spec.updateStrategy.type` is set to `OnDelete`, the StatefulSet controller will not automatically update the Pods in a StatefulSet. Users must manually delete Pods to cause the controller to create new Pods that reflect modifications made to a StatefulSet's `.spec.template`.

`RollingUpdate`

The `RollingUpdate` update strategy implements automated, rolling updates for the Pods in a StatefulSet. This is the default update strategy.

## Rolling Updates

When a StatefulSet's `.spec.updateStrategy.type` is set to `RollingUpdate`, the StatefulSet controller will delete and recreate each Pod in the StatefulSet. It will proceed in the same order as Pod termination (from the largest ordinal to the smallest), updating each Pod one at a time.

The Kubernetes control plane waits until an updated Pod is Running and Ready prior to updating its predecessor. If you have set `.spec.minReadySeconds` (see Minimum Ready Seconds), the control plane additionally waits that amount of time after the Pod turns ready, before moving on.

### Partitioned rolling updates

The `RollingUpdate` update strategy can be partitioned, by specifying a `.spec.updateStrategy.rollingUpdate.partition`. If a partition is specified, all Pods with an ordinal that is greater than or equal to the partition will be updated when the StatefulSet's `.spec.template` is updated. All Pods with an ordinal that is less than the partition will not be updated, and, even if they are deleted, they will be recreated at the previous version. If a StatefulSet's `.spec.updateStrategy.rollingUpdate.partition` is greater than its `.spec.replicas`, updates to its `.spec.template` will not be propagated to its Pods. In most cases you will not need to use a partition, but they are useful if you want to stage an update, roll out a canary, or perform a phased roll out.

### Maximum unavailable Pods

FEATURE STATE: `Kubernetes v1.24 [alpha]`

You can control the maximum number of Pods that can be unavailable during an update by specifying the `.spec.updateStrategy.rollingUpdate.maxUnavailable` field. The value can be an absolute number (for example, `5`) or a percentage of desired Pods (for example, `10%`). Absolute number is calculated from the percentage value by rounding it up. This field cannot be 0. The default setting is 1.

This field applies to all Pods in the range `0` to `replicas - 1`. If there is any unavailable Pod in the range `0` to `replicas - 1`, it will be counted towards `maxUnavailable`.

**Note:**

The `maxUnavailable` field is in Alpha stage and it is honored only by API servers that are running with the `MaxUnavailableStatefulSet` feature gate enabled.

### Forced rollback

When using Rolling Updates with the default Pod Management Policy (`OrderedReady`), it's possible to get into a broken state that requires manual intervention to repair.

If you update the Pod template to a configuration that never becomes Running and Ready (for example, due to a bad binary or application-level configuration error), StatefulSet will stop the rollout and wait.

In this state, it's not enough to revert the Pod template to a good configuration. Due to a known issue, StatefulSet will continue to wait for the broken Pod to become Ready (which never happens) before it will attempt to revert it back to the working configuration.

After reverting the template, you must also delete any Pods that StatefulSet had already attempted to run with the bad configuration. StatefulSet will then begin to recreate the Pods using the reverted template.

## Revision history

ControllerRevision is a Kubernetes API resource used by controllers, such as the StatefulSet controller, to track historical configuration changes.

StatefulSets use ControllerRevisions to maintain a revision history, enabling rollbacks and version tracking.

### How StatefulSets track changes using ControllerRevisions

When you update a StatefulSet's Pod template (`spec.template`), the StatefulSet controller:

1. Prepares a new ControllerRevision object
2. Stores a snapshot of the Pod template and metadata
3. Assigns an incremental revision number

#### Key Properties

See ControllerRevision to learn more about key properties and other details.

---

### Managing Revision History

Control retained revisions with `.spec.revisionHistoryLimit`:

```
apiVersion: apps/v1
kind: StatefulSetmetadata:  name: webappspec:  revisionHistoryLimit: 5  # Keep last 5 revisions  # ... other spec fields ...
```

- **Default**: 10 revisions retained if unspecified
- **Cleanup**: Oldest revisions are garbage-collected when exceeding the limit

#### Performing Rollbacks

You can revert to a previous configuration using:

```
# View revision history
kubectl rollout history statefulset/webapp

# Rollback to a specific revision
kubectl rollout undo statefulset/webapp --to-revision=3
```

This will:

- Apply the Pod template from revision 3
- Create a new ControllerRevision with an updated revision number

#### Inspecting ControllerRevisions

To view associated ControllerRevisions:

```
# List all revisions for the StatefulSet
kubectl get controllerrevisions -l app.kubernetes.io/name=webapp

# View detailed configuration of a specific revision
kubectl get controllerrevision/webapp-3 -o yaml
```

#### Best Practices

##### Retention Policy

- Set `revisionHistoryLimit` between **5–10** for most workloads.
- Increase only if **deep rollback history** is required.

##### Monitoring

- Regularly check revisions with:

  ```
  kubectl get controllerrevisions
  ```

- Alert on **rapid revision count growth**.

##### Avoid

- Manual edits to ControllerRevision objects.
- Using revisions as a backup mechanism (use actual backup tools).
- Setting `revisionHistoryLimit: 0` (disables rollback capability).

## PersistentVolumeClaim retention

FEATURE STATE: `Kubernetes v1.32 [stable]` (enabled by default: true)

The optional `.spec.persistentVolumeClaimRetentionPolicy` field controls if and how PVCs are deleted during the lifecycle of a StatefulSet. You must enable the `StatefulSetAutoDeletePVC` feature gate on the API server and the controller manager to use this field. Once enabled, there are two policies you can configure for each StatefulSet:

`whenDeleted`
: Configures the volume retention behavior that applies when the StatefulSet is deleted.

`whenScaled`
: Configures the volume retention behavior that applies when the replica count of the StatefulSet is reduced; for example, when scaling down the set.

For each policy that you can configure, you can set the value to either `Delete` or `Retain`.

`Delete`
: The PVCs created from the StatefulSet `volumeClaimTemplate` are deleted for each Pod affected by the policy. With the `whenDeleted` policy all PVCs from the `volumeClaimTemplate` are deleted after their Pods have been deleted. With the `whenScaled` policy, only PVCs corresponding to Pod replicas being scaled down are deleted, after their Pods have been deleted.

`Retain` (default)
: PVCs from the `volumeClaimTemplate` are not affected when their Pod is deleted. This is the behavior before this new feature.

Bear in mind that these policies **only** apply when Pods are being removed due to the StatefulSet being deleted or scaled down. For example, if a Pod associated with a StatefulSet fails due to node failure, and the control plane creates a replacement Pod, the StatefulSet retains the existing PVC. The existing volume is unaffected, and the cluster will attach it to the node where the new Pod is about to launch.

The default for policies is `Retain`, matching the StatefulSet behavior before this new feature.

Here is an example policy:

```
apiVersion: apps/v1
kind: StatefulSet...spec:   persistentVolumeClaimRetentionPolicy:    whenDeleted: Retain    whenScaled: Delete...
```

The StatefulSet controller adds owner references to its PVCs, which are then deleted by the garbage collector after the Pod is terminated. This enables the Pod to cleanly unmount all volumes before the PVCs are deleted (and before the backing PV and volume are deleted, depending on the retain policy). When you set the `whenDeleted` policy to `Delete`, an owner reference to the StatefulSet instance is placed on all PVCs associated with that StatefulSet.

The `whenScaled` policy must delete PVCs only when a Pod is scaled down, and not when a Pod is deleted for another reason. When reconciling, the StatefulSet controller compares its desired replica count to the actual Pods present on the cluster. Any StatefulSet Pod whose id greater than the replica count is condemned and marked for deletion. If the `whenScaled` policy is `Delete`, the condemned Pods are first set as owners to the associated StatefulSet template PVCs, before the Pod is deleted. This causes the PVCs to be garbage collected after only the condemned Pods have terminated.

This means that if the controller crashes and restarts, no Pod will be deleted before its owner reference has been updated appropriate to the policy. If a condemned Pod is force-deleted while the controller is down, the owner reference may or may not have been set up, depending on when the controller crashed. It may take several reconcile loops to update the owner references, so some condemned Pods may have set up owner references and others may not. For this reason we recommend waiting for the controller to come back up, which will verify owner references before terminating Pods. If that is not possible, the operator should verify the owner references on PVCs to ensure the expected objects are deleted when Pods are force-deleted.

### Replicas

`.spec.replicas` is an optional field that specifies the number of desired Pods. It defaults to 1.

Should you manually scale a deployment, example via `kubectl scale statefulset statefulset --replicas=X`, and then you update that StatefulSet based on a manifest (for example: by running `kubectl apply -f statefulset.yaml`), then applying that manifest overwrites the manual scaling that you previously did.

If a HorizontalPodAutoscaler (or any similar API for horizontal scaling) is managing scaling for a Statefulset, don't set `.spec.replicas`. Instead, allow the Kubernetes control plane to manage the `.spec.replicas` field automatically.

## What's next

- Learn about Pods.
- Find out how to use StatefulSets
  - Follow an example of deploying a stateful application.
  - Follow an example of deploying Cassandra with Stateful Sets.
  - Follow an example of running a replicated stateful application.
  - Learn how to scale a StatefulSet.
  - Learn what's involved when you delete a StatefulSet.
  - Learn how to configure a Pod to use a volume for storage.
  - Learn how to configure a Pod to use a PersistentVolume for storage.
- `StatefulSet` is a top-level resource in the Kubernetes REST API. Read the StatefulSet object definition to understand the API for stateful sets.
- Read about PodDisruptionBudget and how you can use it to manage application availability during disruptions.

---

# Autoscaling Workloads

With autoscaling, you can automatically update your workloads in one way or another. This allows your cluster to react to changes in resource demand more elastically and efficiently.

In Kubernetes, you can *scale* a workload depending on the current demand of resources. This allows your cluster to react to changes in resource demand more elastically and efficiently.

When you scale a workload, you can either increase or decrease the number of replicas managed by the workload, or adjust the resources available to the replicas in-place.

The first approach is referred to as *horizontal scaling*, while the second is referred to as *vertical scaling*.

There are manual and automatic ways to scale your workloads, depending on your use case.

## Scaling workloads manually

Kubernetes supports *manual scaling* of workloads. Horizontal scaling can be done using the `kubectl` CLI. For vertical scaling, you need to *patch* the resource definition of your workload.

See below for examples of both strategies.

- **Horizontal scaling**: Running multiple instances of your app
- **Vertical scaling**: Resizing CPU and memory resources assigned to containers

## Scaling workloads automatically

Kubernetes also supports *automatic scaling* of workloads, which is the focus of this page.

The concept of *Autoscaling* in Kubernetes refers to the ability to automatically update an object that manages a set of Pods (for example a Deployment).

### Scaling workloads horizontally

In Kubernetes, you can automatically scale a workload horizontally using a *HorizontalPodAutoscaler* (HPA).

It is implemented as a Kubernetes API resource and a controller and periodically adjusts the number of replicas in a workload to match observed resource utilization such as CPU or memory usage.

There is a walkthrough tutorial of configuring a HorizontalPodAutoscaler for a Deployment.

### Scaling workloads vertically

FEATURE STATE: `Kubernetes v1.25 [stable]`

You can automatically scale a workload vertically using a *VerticalPodAutoscaler* (VPA). Unlike the HPA, the VPA doesn't come with Kubernetes by default, but is a separate project that can be found on GitHub.

Once installed, it allows you to create CustomResourceDefinitions (CRDs) for your workloads which define *how* and *when* to scale the resources of the managed replicas.

**Note:**

You will need to have the Metrics Server installed to your cluster for the VPA to work.

At the moment, the VPA can operate in four different modes:

| Mode | Description |
|---|---|
| `Auto` | Currently `Recreate`. This might change to in-place updates in the future. |
| `Recreate` | The VPA assigns resource requests on pod creation as well as updates them on existing pods by evicting them when the requested resources differ significantly from the new recommendation |
| `Initial` | The VPA only assigns resource requests on pod creation and never changes them later. |
| `Off` | The VPA does not automatically change the resource requirements of the pods. The recommendations are calculated and can be inspected in the VPA object. |

#### In-place pod vertical scaling

FEATURE STATE: `Kubernetes v1.33 [beta]` (enabled by default: true)

As of Kubernetes 1.34, VPA does not support resizing pods in-place, but this integration is being worked on. For manually resizing pods in-place, see Resize Container Resources In-Place.

### Autoscaling based on cluster size

For workloads that need to be scaled based on the size of the cluster (for example `cluster-dns` or other system components), you can use the *Cluster Proportional Autoscaler*. Just like the VPA, it is not part of the Kubernetes core, but hosted as its own project on GitHub.

The Cluster Proportional Autoscaler watches the number of schedulable nodes and cores and scales the number of replicas of the target workload accordingly.

If the number of replicas should stay the same, you can scale your workloads vertically according to the cluster size using the *Cluster Proportional Vertical Autoscaler*. The project is **currently in beta** and can be found on GitHub.

While the Cluster Proportional Autoscaler scales the number of replicas of a workload, the Cluster Proportional Vertical Autoscaler adjusts the resource requests for a workload (for example a Deployment or DaemonSet) based on the number of nodes and/or cores in the cluster.

### Event driven Autoscaling

It is also possible to scale workloads based on events, for example using the *Kubernetes Event Driven Autoscaler* (**KEDA**).

KEDA is a CNCF-graduated project enabling you to scale your workloads based on the number of events to be processed, for example the amount of messages in a queue. There exists a wide range of adapters for different event sources to choose from.

### Autoscaling based on schedules

Another strategy for scaling your workloads is to **schedule** the scaling operations, for example in order to reduce resource consumption during off-peak hours.

Similar to event driven autoscaling, such behavior can be achieved using KEDA in conjunction with its `Cron` scaler. The `Cron` scaler allows you to define schedules (and time zones) for scaling your workloads in or out.

## Scaling cluster infrastructure

If scaling workloads isn't enough to meet your needs, you can also scale your cluster infrastructure itself.

Scaling the cluster infrastructure normally means adding or removing nodes. Read Node autoscaling for more information.

## What's next

- Learn more about scaling horizontally
  - Scale a StatefulSet
  - HorizontalPodAutoscaler Walkthrough
- Resize Container Resources In-Place
- Autoscale the DNS Service in a Cluster
- Learn about Node autoscaling

# Managing Workloads

You've deployed your application and exposed it via a Service. Now what? Kubernetes provides a number of tools to help you manage your application deployment, including scaling and updating.

## Organizing resource configurations

Many applications require multiple resources to be created, such as a Deployment along with a Service. Management of multiple resources can be simplified by grouping them together in the same file (separated by `---` in YAML). For example:

application/nginx-app.yaml Copy application/nginx-app.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  name: my-nginx-svc  labels:    app: nginxspec:  type: LoadBalancer  ports:  - port: 80  selector:    app:
```

Multiple resources can be created the same way as a single resource:

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

```
service/my-nginx-svc created
deployment.apps/my-nginx created
```

The resources will be created in the order they appear in the manifest. Therefore, it's best to specify the Service first, since that will ensure the scheduler can spread the pods associated with the Service as they are created by the controller(s), such as Deployment.

`kubectl apply` also accepts multiple `-f` arguments:

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

It is a recommended practice to put resources related to the same microservice or application tier into the same file, and to group all of the files associated with your application in the same directory. If the tiers of your application bind to each other using DNS, you can deploy all of the components of your stack together.

A URL can also be specified as a configuration source, which is handy for deploying directly from manifests in your source control system:

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

```
deployment.apps/my-nginx created
```

If you need to define more manifests, such as adding a ConfigMap, you can do that too.

### External tools

This section lists only the most common tools used for managing workloads on Kubernetes. To see a larger list, view Application definition and image build in the CNCF Landscape.

#### Helm

☐ This item links to a third party project or product that is not part of Kubernetes itself. More information

Helm is a tool for managing packages of pre-configured Kubernetes resources. These packages are known as *Helm charts*.

#### Kustomize

Kustomize traverses a Kubernetes manifest to add, remove or update configuration options. It is available both as a standalone binary and as a native feature of kubectl.

## Bulk operations in kubectl

Resource creation isn't the only operation that `kubectl` can perform in bulk. It can also extract resource names from configuration files in order to perform other operations, in particular to delete the same resources you created:

```
kubectl delete -f https://k8s.io/examples/application/nginx-app.yaml
```

```
deployment.apps "my-nginx" deleted
service "my-nginx-svc" deleted
```

In the case of two resources, you can specify both resources on the command line using the resource/name syntax:

```
kubectl delete deployments/my-nginx services/my-nginx-svc
```

For larger numbers of resources, you'll find it easier to specify the selector (label query) specified using `-l` or `--selector`, to filter resources by their labels:

```
kubectl delete deployment,services -l app=nginx
```

```
deployment.apps "my-nginx" deleted
service "my-nginx-svc" deleted
```

### Chaining and filtering

Because `kubectl` outputs resource names in the same syntax it accepts, you can chain operations using `$()` or `xargs`:

```
kubectl get $(kubectl create -f docs/concepts/cluster-administration/nginx/ -o name | grep service/ )
kubectl create -f docs/concepts/cluster-administration/nginx/ -o name | grep service/ | xargs -i kubectl get '{}'
```

The output might be similar to:

```
NAME           TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)      AGE
my-nginx-svc   LoadBalancer   10.0.0.208   <pending>     80/TCP       0s
```

With the above commands, first you create resources under `docs/concepts/cluster-administration/nginx/` and print the resources created with `-o name` output format (print each resource as resource/name). Then you `grep` only the Service, and then print it with [`kubectl get`](https://kubernetes.io/docs/reference/kubectl/generated/kubectl_get/).

### Recursive operations on local files

If you happen to organize your resources across several subdirectories within a particular directory, you can recursively perform the operations on the subdirectories also, by specifying `--recursive` or `-R` alongside the `--filename`/`-f` argument.

For instance, assume there is a directory `project/k8s/development` that holds all of the [manifests](https://kubernetes.io/docs/reference/glossary/?all=true#term-manifest) needed for the development environment, organized by resource type:

```
project/k8s/development
├── configmap
│   └── my-configmap.yaml
├── deployment
│   └── my-deployment.yaml
└── pvc
    └── my-pvc.yaml
```

By default, performing a bulk operation on `project/k8s/development` will stop at the first level of the directory, not processing any subdirectories. If you had tried to create the resources in this directory using the following command, we would have encountered an error:

```
kubectl apply -f project/k8s/development
```

```
error: you must provide one or more resources by argument or filename (.json|.yaml|.yml|stdin)
```

Instead, specify the `--recursive` or `-R` command line argument along with the `--filename`/`-f` argument:

```
kubectl apply -f project/k8s/development --recursive
```

```
configmap/my-config created
deployment.apps/my-deployment created
persistentvolumeclaim/my-pvc created
```

The `--recursive` argument works with any operation that accepts the `--filename`/`-f` argument such as: `kubectl create`, `kubectl get`, `kubectl delete`, `kubectl describe`, or even `kubectl rollout`.

The `--recursive` argument also works when multiple `-f` arguments are provided:

```
kubectl apply -f project/k8s/namespaces -f project/k8s/development --recursive
```

```
namespace/development created
namespace/staging created
configmap/my-config created
deployment.apps/my-deployment created
persistentvolumeclaim/my-pvc created
```

If you're interested in learning more about `kubectl`, go ahead and read [Command line tool (kubectl)](https://kubernetes.io/docs/reference/kubectl/).

## Updating your application without an outage

At some point, you'll eventually need to update your deployed application, typically by specifying a new image or image tag. `kubectl` supports several update operations, each of which is applicable to different scenarios.

You can run multiple copies of your app, and use a *rollout* to gradually shift the traffic to new healthy Pods. Eventually, all the running Pods would have the new software.

This section of the page guides you through how to create and update applications with Deployments.

Let's say you were running version 1.14.2 of nginx:

```
kubectl create deployment my-nginx --image=nginx:1.14.2
```

```
deployment.apps/my-nginx created
```

Ensure that there is 1 replica:

```
kubectl scale --replicas 1 deployments/my-nginx --subresource='scale' --type='merge' -p '{"spec":{"replicas": 1}}'
```

```
deployment.apps/my-nginx scaled
```

and allow Kubernetes to add more temporary replicas during a rollout, by setting a *surge maximum* of 100%:

```
kubectl patch --type='merge' -p '{"spec":{"strategy":{"rollingUpdate":{"maxSurge": "100%" }}}}'
```

```
deployment.apps/my-nginx patched
```

To update to version 1.16.1, change `.spec.template.spec.containers[0].image` from `nginx:1.14.2` to `nginx:1.16.1` using `kubectl edit`:

```
kubectl edit deployment/my-nginx
# Change the manifest to use the newer container image, then save your changes
```

That's it! The Deployment will declaratively update the deployed nginx application progressively behind the scene. It ensures that only a certain number of old replicas may be down while they are being updated, and only a certain number of new replicas may be created above the desired number of pods. To learn more details about how this happens, visit Deployment.

You can use rollouts with DaemonSets, Deployments, or StatefulSets.

### Managing rollouts

You can use `kubectl rollout` to manage a progressive update of an existing application.

For example:

```
kubectl apply -f my-deployment.yaml

# wait for rollout to finish
kubectl rollout status deployment/my-deployment --timeout 10m # 10 minute timeout
```

or

```
kubectl apply -f backing-stateful-component.yaml

# don't wait for rollout to finish, just check the status
kubectl rollout status statefulsets/backing-stateful-component --watch=false
```

You can also pause, resume or cancel a rollout. Visit `kubectl rollout` to learn more.

## Canary deployments

Another scenario where multiple labels are needed is to distinguish deployments of different releases or configurations of the same component. It is common practice to deploy a *canary* of a new application release (specified via image tag in the pod template) side by side with the previous release so that the new release can receive live production traffic before fully rolling it out.

For instance, you can use a `track` label to differentiate different releases.

The primary, stable release would have a `track` label with value as `stable`:

```
name: frontend
replicas: 3
...
labels:
   app: guestbook
   tier: frontend
   track: stable
...
image: gb-frontend:v3
```

and then you can create a new release of the guestbook frontend that carries the `track` label with different value (i.e. `canary`), so that two sets of pods would not overlap:

```
name: frontend-canary
replicas: 1
...
labels:
   app: guestbook
   tier: frontend
   track: canary
...
image: gb-frontend:v4
```

The frontend service would span both sets of replicas by selecting the common subset of their labels (i.e. omitting the `track` label), so that the traffic will be redirected to both applications:

```
selector:
   app: guestbook
   tier: frontend
```

You can tweak the number of replicas of the stable and canary releases to determine the ratio of each release that will receive live production traffic (in this case, 3:1). Once you're confident, you can update the stable track to the new application release and remove the canary one.

## Updating annotations

Sometimes you would want to attach annotations to resources. Annotations are arbitrary non-identifying metadata for retrieval by API clients such as tools or libraries. This can be done with `kubectl annotate`. For example:

```
kubectl annotate pods my-nginx-v4-9gw19 description='my frontend running nginx'
kubectl get pods my-nginx-v4-9gw19 -o yaml
```

```
apiVersion: v1
kind: pod
metadata:
  annotations:
    description: my frontend running nginx
...
```

For more information, see annotations and kubectl annotate.

## Scaling your application

When load on your application grows or shrinks, use `kubectl` to scale your application. For instance, to decrease the number of nginx replicas from 3 to 1, do:

```
kubectl scale deployment/my-nginx --replicas=1
```

```
deployment.apps/my-nginx scaled
```

Now you only have one pod managed by the deployment.

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

```
NAME                        READY     STATUS    RESTARTS   AGE
my-nginx-2035384211-j5fhi   1/1       Running   0          30m
```

To have the system automatically choose the number of nginx replicas as needed, ranging from 1 to 3, do:

```
# This requires an existing source of container and Pod metrics
kubectl autoscale deployment/my-nginx --min=1 --max=3
```

```
horizontalpodautoscaler.autoscaling/my-nginx autoscaled
```

Now your nginx replicas will be scaled up and down as needed, automatically.

For more information, please see kubectl scale, kubectl autoscale and horizontal pod autoscaler document.

## In-place updates of resources

Sometimes it's necessary to make narrow, non-disruptive updates to resources you've created.

### kubectl apply

It is suggested to maintain a set of configuration files in source control (see configuration as code), so that they can be maintained and versioned along with the code for the resources they configure. Then, you can use `kubectl apply` to push your configuration changes to the cluster.

This command will compare the version of the configuration that you're pushing with the previous version and apply the changes you've made, without overwriting any automated changes to properties you haven't specified.

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

```
deployment.apps/my-nginx configured
```

To learn more about the underlying mechanism, read server-side apply.

### kubectl edit

Alternatively, you may also update resources with `kubectl edit`:

```
kubectl edit deployment/my-nginx
```

This is equivalent to first `get` the resource, edit it in text editor, and then `apply` the resource with the updated version:

```
kubectl get deployment my-nginx -o yaml > /tmp/nginx.yaml
vi /tmp/nginx.yaml
# do some edit, and then save the file

kubectl apply -f /tmp/nginx.yaml
deployment.apps/my-nginx configured

rm /tmp/nginx.yaml
```

This allows you to do more significant changes more easily. Note that you can specify the editor with your `EDITOR` or `KUBE_EDITOR` environment variables.

For more information, please see kubectl edit.

### kubectl patch

You can use `kubectl patch` to update API objects in place. This subcommand supports JSON patch, JSON merge patch, and strategic merge patch.

See Update API Objects in Place Using kubectl patch for more details.

## Disruptive updates

In some cases, you may need to update resource fields that cannot be updated once initialized, or you may want to make a recursive change immediately, such as to fix broken pods created by a Deployment. To change such fields, use `replace --force`, which deletes and re-creates the resource. In this case, you can modify your original configuration file:

```
kubectl replace -f https://k8s.io/examples/application/nginx/nginx-deployment.yaml --force
```

```
deployment.apps/my-nginx deleted
deployment.apps/my-nginx replaced
```

## What's next

- Learn about how to use `kubectl` for application introspection and debugging.

# Guide for Running Windows Containers in Kubernetes

This page provides a walkthrough for some steps you can follow to run Windows containers using Kubernetes. The page also highlights some Windows specific functionality within Kubernetes.

It is important to note that creating and deploying services and workloads on Kubernetes behaves in much the same way for Linux and Windows containers. The kubectl commands to interface with the cluster are identical. The examples in this page are provided to jumpstart your experience with Windows containers.

## Objectives

Configure an example deployment to run Windows containers on a Windows node.

## Before you begin

You should already have access to a Kubernetes cluster that includes a worker node running Windows Server.

## Getting Started: Deploying a Windows workload

The example YAML file below deploys a simple webserver application running inside a Windows container.

Create a manifest named `win-webserver.yaml` with the contents below:

```
---
apiVersion: v1kind: Servicemetadata:  name: win-webserver  labels:    app: win-webserverspec:  ports:    # the port that this serv
```

**Note:**

Port mapping is also supported, but for simplicity this example exposes port 80 of the container directly to the Service.

1. Check that all nodes are healthy:

   ```
   kubectl get nodes
   ```

2. Deploy the service and watch for pod updates:

   ```
   kubectl apply -f win-webserver.yaml
   kubectl get pods -o wide -w
   ```

   When the service is deployed correctly both Pods are marked as Ready. To exit the watch command, press Ctrl+C.

3. Check that the deployment succeeded. To verify:

   - Several pods listed from the Linux control plane node, use `kubectl get pods`
   - Node-to-pod communication across the network, `curl` port 80 of your pod IPs from the Linux control plane node to check for a web server response
   - Pod-to-pod communication, ping between pods (and across hosts, if you have more than one Windows node) using `kubectl exec`
   - Service-to-pod communication, `curl` the virtual service IP (seen under `kubectl get services`) from the Linux control plane node and from individual pods
   - Service discovery, `curl` the service name with the Kubernetes default DNS suffix
   - Inbound connectivity, `curl` the NodePort from the Linux control plane node or machines outside of the cluster
   - Outbound connectivity, `curl` external IPs from inside the pod using `kubectl exec`

**Note:**

Windows container hosts are not able to access the IP of services scheduled on them due to current platform limitations of the Windows networking stack. Only Windows pods are able to access service IPs.

## Observability

### Capturing logs from workloads

Logs are an important element of observability; they enable users to gain insights into the operational aspect of workloads and are a key ingredient to troubleshooting issues. Because Windows containers and workloads inside Windows containers behave differently from Linux containers, users had a hard time collecting logs, limiting operational visibility. Windows workloads for example are usually configured to log to ETW (Event Tracing for Windows) or push entries to the application event log. LogMonitor, an open source tool by Microsoft, is the recommended way to monitor configured log sources inside a Windows container. LogMonitor supports monitoring event logs, ETW providers, and custom application logs, piping them to STDOUT for consumption by `kubectl logs <pod>`.

Follow the instructions in the LogMonitor GitHub page to copy its binaries and configuration files to all your containers and add the necessary entrypoints for LogMonitor to push your logs to STDOUT.

## Configuring container user

### Using configurable Container usernames

Windows containers can be configured to run their entrypoints and processes with different usernames than the image defaults. Learn more about it here.

### Managing Workload Identity with Group Managed Service Accounts

Windows container workloads can be configured to use Group Managed Service Accounts (GMSA). Group Managed Service Accounts are a specific type of Active Directory account that provide automatic password management, simplified service principal name (SPN) management, and the ability to delegate the management to other administrators across multiple servers. Containers configured with a GMSA can access external Active Directory Domain resources while carrying the identity configured with the GMSA. Learn more about configuring and using GMSA for Windows containers here.

## Taints and tolerations

Users need to use some combination of taint and node selectors in order to schedule Linux and Windows workloads to their respective OS-specific nodes. The recommended approach is outlined below, with one of its main goals being that this approach should not break compatibility for existing Linux workloads.

You can (and should) set `.spec.os.name` for each Pod, to indicate the operating system that the containers in that Pod are designed for. For Pods that run Linux containers, set `.spec.os.name` to `linux`. For Pods that run Windows containers, set `.spec.os.name` to `windows`.

**Note:**

If you are running a version of Kubernetes older than 1.24, you may need to enable the `IdentifyPodOS` feature gate to be able to set a value for `.spec.pod.os`.

The scheduler does not use the value of `.spec.os.name` when assigning Pods to nodes. You should use normal Kubernetes mechanisms for assigning pods to nodes to ensure that the control plane for your cluster places pods onto nodes that are running the appropriate operating system.

The `.spec.os.name` value has no effect on the scheduling of the Windows pods, so taints and tolerations (or node selectors) are still required to ensure that the Windows pods land onto appropriate Windows nodes.

### Ensuring OS-specific workloads land on the appropriate container host

Users can ensure Windows containers can be scheduled on the appropriate host using taints and tolerations. All Kubernetes nodes running Kubernetes 1.34 have the following default labels:

- kubernetes.io/os = [windows|linux]
- kubernetes.io/arch = [amd64|arm64|...]

If a Pod specification does not specify a `nodeSelector` such as `"kubernetes.io/os": windows`, it is possible the Pod can be scheduled on any host, Windows or Linux. This can be problematic since a Windows container can only run on Windows and a Linux container can only run on Linux. The best practice for Kubernetes 1.34 is to use a `nodeSelector`.

However, in many cases users have a pre-existing large number of deployments for Linux containers, as well as an ecosystem of off-the-shelf configurations, such as community Helm charts, and programmatic Pod generation cases, such as with operators. In those situations, you may be hesitant to make the configuration change to add `nodeSelector` fields to all Pods and Pod templates. The alternative is to use taints. Because the kubelet can set taints during registration, it could easily be modified to automatically add a taint when running on Windows only.

For example: `--register-with-taints='os=windows:NoSchedule'`

By adding a taint to all Windows nodes, nothing will be scheduled on them (that includes existing Linux Pods). In order for a Windows Pod to be scheduled on a Windows node, it would need both the `nodeSelector` and the appropriate matching toleration to choose Windows.

```
nodeSelector:
    kubernetes.io/os: windows
    node.kubernetes.io/windows-build: '10.0.17763'
tolerations:    - key: "os"      operator: "Equal"      value: "windows"      effect: "NoSchedule"
```

### Handling multiple Windows versions in the same cluster

The Windows Server version used by each pod must match that of the node. If you want to use multiple Windows Server versions in the same cluster, then you should set additional node labels and `nodeSelector` fields.

Kubernetes automatically adds a label, `node.kubernetes.io/windows-build` to simplify this.

This label reflects the Windows major, minor, and build number that need to match for compatibility. Here are values used for each Windows Server version:

| Product Name | Version |
|---|---|
| Windows Server 2019 | 10.0.17763 |
| Windows Server 2022 | 10.0.20348 |

### Simplifying with RuntimeClass

RuntimeClass can be used to simplify the process of using taints and tolerations. A cluster administrator can create a `RuntimeClass` object which is used to encapsulate these taints and tolerations.

1. Save this file to `runtimeClasses.yml`. It includes the appropriate `nodeSelector` for the Windows OS, architecture, and version.

   ```
   ---
   apiVersion: node.k8s.io/v1kind: RuntimeClassmetadata:  name: windows-2019handler: example-container-runtime-handlerscheduling
   ```

2. Run `kubectl create -f runtimeClasses.yml` using as a cluster administrator
3. Add `runtimeClassName: windows-2019` as appropriate to Pod specs

   For example:

   ```
   ---
   apiVersion: apps/v1kind: Deploymentmetadata:  name: iis-2019  labels:    app: iis-2019spec:  replicas: 1  template:    metada
   ```

---

# Downward API

There are two ways to expose Pod and container fields to a running container: environment variables, and as files that are populated by a special volume type. Together, these two ways of exposing Pod and container fields are called the downward API.

It is sometimes useful for a container to have information about itself, without being overly coupled to Kubernetes. The *downward API* allows containers to consume information about themselves or the cluster without using the Kubernetes client or API server.

An example is an existing application that assumes a particular well-known environment variable holds a unique identifier. One possibility is to wrap the application, but that is tedious and error-prone, and it violates the goal of low coupling. A better option would be to use the Pod's name as an identifier, and inject the Pod's name into the well-known environment variable.

In Kubernetes, there are two ways to expose Pod and container fields to a running container:

- as environment variables
- as files in a `downwardAPI` volume

Together, these two ways of exposing Pod and container fields are called the *downward API*.

## Available fields

Only some Kubernetes API fields are available through the downward API. This section lists which fields you can make available.

You can pass information from available Pod-level fields using `fieldRef`. At the API level, the `spec` for a Pod always defines at least one Container. You can pass information from available Container-level fields using `resourceFieldRef`.

### Information available via `fieldRef`

For some Pod-level fields, you can provide them to a container either as an environment variable or using a `downwardAPI` volume. The fields available via either mechanism are:

`metadata.name`
: the pod's name

`metadata.namespace`
: the pod's namespace

`metadata.uid`
: the pod's unique ID

`metadata.annotations['<KEY>']`
: the value of the pod's annotation named `<KEY>` (for example, `metadata.annotations['myannotation']`)

`metadata.labels['<KEY>']`
: the text value of the pod's label named `<KEY>` (for example, `metadata.labels['mylabel']`)

The following information is available through environment variables **but not as a downwardAPI volume fieldRef**:

`spec.serviceAccountName`
: the name of the pod's service account

`spec.nodeName`
: the name of the node where the Pod is executing

`status.hostIP`
: the primary IP address of the node to which the Pod is assigned

`status.hostIPs`
: the IP addresses is a dual-stack version of `status.hostIP`, the first is always the same as `status.hostIP`.

`status.podIP`
: the pod's primary IP address (usually, its IPv4 address)

`status.podIPs`
: the IP addresses is a dual-stack version of `status.podIP`, the first is always the same as `status.podIP`

The following information is available through a `downwardAPI` volume `fieldRef`, **but not as environment variables**:

`metadata.labels`
: all of the pod's labels, formatted as `label-key="escaped-label-value"` with one label per line

`metadata.annotations`
: all of the pod's annotations, formatted as `annotation-key="escaped-annotation-value"` with one annotation per line

### Information available via `resourceFieldRef`

These container-level fields allow you to provide information about requests and limits for resources such as CPU and memory.

**Note:**

FEATURE STATE: `Kubernetes v1.33 [beta]` (enabled by default: true)

Container CPU and memory resources can be resized while the container is running. If this happens, a downward API volume will be updated, but environment variables will not be updated unless the container restarts. See Resize CPU and Memory Resources assigned to Containers for more details.

`resource: limits.cpu`
: A container's CPU limit

`resource: requests.cpu`
: A container's CPU request

`resource: limits.memory`
: A container's memory limit

`resource: requests.memory`
: A container's memory request

`resource: limits.hugepages-*`
: A container's hugepages limit

`resource: requests.hugepages-*`
: A container's hugepages request

`resource: limits.ephemeral-storage`
: A container's ephemeral-storage limit

`resource: requests.ephemeral-storage`
: A container's ephemeral-storage request

#### Fallback information for resource limits

If CPU and memory limits are not specified for a container, and you use the downward API to try to expose that information, then the kubelet defaults to exposing the maximum allocatable value for CPU and memory based on the node allocatable calculation.

## What's next

You can read about `downwardAPI` volumes.

You can try using the downward API to expose container- or Pod-level information:

- as environment variables
- as files in `downwardAPI` volume

---

# Windows containers in Kubernetes

Windows applications constitute a large portion of the services and applications that run in many organizations. Windows containers provide a way to encapsulate processes and package dependencies, making it easier to use DevOps practices and follow cloud native patterns for Windows applications.

Organizations with investments in Windows-based applications and Linux-based applications don't have to look for separate orchestrators to manage their workloads, leading to increased operational efficiencies across their deployments, regardless of operating system.

## Windows nodes in Kubernetes

To enable the orchestration of Windows containers in Kubernetes, include Windows nodes in your existing Linux cluster. Scheduling Windows containers in Pods on Kubernetes is similar to scheduling Linux-based containers.

In order to run Windows containers, your Kubernetes cluster must include multiple operating systems. While you can only run the control plane on Linux, you can deploy worker nodes running either Windows or Linux.

Windows nodes are supported provided that the operating system is Windows Server 2019 or Windows Server 2022.

This document uses the term *Windows containers* to mean Windows containers with process isolation. Kubernetes does not support running Windows containers with Hyper-V isolation.

## Compatibility and limitations

Some node features are only available if you use a specific container runtime; others are not available on Windows nodes, including:

- HugePages: not supported for Windows containers
- Privileged containers: not supported for Windows containers. HostProcess Containers offer similar functionality.
- TerminationGracePeriod: requires containerD

Not all features of shared namespaces are supported. See API compatibility for more details.

See Windows OS version compatibility for details on the Windows versions that Kubernetes is tested against.

From an API and kubectl perspective, Windows containers behave in much the same way as Linux-based containers. However, there are some notable differences in key functionality which are outlined in this section.

### Comparison with Linux

Key Kubernetes elements work the same way in Windows as they do in Linux. This section refers to several key workload abstractions and how they map to Windows.

- Pods

  A Pod is the basic building block of Kubernetes–the smallest and simplest unit in the Kubernetes object model that you create or deploy. You may not deploy Windows and Linux containers in the same Pod. All containers in a Pod are scheduled onto a single Node where each Node represents a specific platform and architecture. The following Pod capabilities, properties and events are supported with Windows containers:

  - Single or multiple containers per Pod with process isolation and volume sharing
  - Pod `status` fields
  - Readiness, liveness, and startup probes
  - postStart & preStop container lifecycle hooks
  - ConfigMap, Secrets: as environment variables or volumes
  - `emptyDir` volumes
  - Named pipe host mounts
  - Resource limits
  - OS field:

    The `.spec.os.name` field should be set to `windows` to indicate that the current Pod uses Windows containers.

    If you set the `.spec.os.name` field to `windows`, you must not set the following fields in the `.spec` of that Pod:

    - `spec.hostPID`
    - `spec.hostIPC`
    - `spec.securityContext.seLinuxOptions`
    - `spec.securityContext.seccompProfile`
    - `spec.securityContext.fsGroup`
    - `spec.securityContext.fsGroupChangePolicy`
    - `spec.securityContext.sysctls`
    - `spec.shareProcessNamespace`
    - `spec.securityContext.runAsUser`
    - `spec.securityContext.runAsGroup`
    - `spec.securityContext.supplementalGroups`
    - `spec.containers[*].securityContext.seLinuxOptions`
    - `spec.containers[*].securityContext.seccompProfile`
    - `spec.containers[*].securityContext.capabilities`
    - `spec.containers[*].securityContext.readOnlyRootFilesystem`
    - `spec.containers[*].securityContext.privileged`
    - `spec.containers[*].securityContext.allowPrivilegeEscalation`
    - `spec.containers[*].securityContext.procMount`
    - `spec.containers[*].securityContext.runAsUser`
    - `spec.containers[*].securityContext.runAsGroup`

    In the above list, wildcards (`*`) indicate all elements in a list. For example, `spec.containers[*].securityContext` refers to the SecurityContext object for all containers. If any of these fields is specified, the Pod will not be admitted by the API server.

- Workload resources including:
  - ReplicaSet
  - Deployment
  - StatefulSet
  - DaemonSet
  - Job
  - CronJob
  - ReplicationController
- Services See Load balancing and Services for more details.

Pods, workload resources, and Services are critical elements to managing Windows workloads on Kubernetes. However, on their own they are not enough to enable the proper lifecycle management of Windows workloads in a dynamic cloud native environment.

- `kubectl exec`
- Pod and container metrics
- Horizontal pod autoscaling
- Resource quotas
- Scheduler preemption

## Command line options for the kubelet

Some kubelet command line options behave differently on Windows, as described below:

- The `--windows-priorityclass` lets you set the scheduling priority of the kubelet process (see CPU resource management)
- The `--kube-reserved`, `--system-reserved` , and `--eviction-hard` flags update NodeAllocatable
- Eviction by using `--enforce-node-allocable` is not implemented
- When running on a Windows node the kubelet does not have memory or CPU restrictions. `--kube-reserved` and `--system-reserved` only subtract from `NodeAllocatable` and do not guarantee resource provided for workloads. See Resource Management for Windows nodes for more information.
- The `PIDPressure` Condition is not implemented
- The kubelet does not take OOM eviction actions

## API compatibility

There are subtle differences in the way the Kubernetes APIs work for Windows due to the OS and container runtime. Some workload properties were designed for Linux, and fail to run on Windows.

At a high level, these OS concepts are different:

- Identity - Linux uses userID (UID) and groupID (GID) which are represented as integer types. User and group names are not canonical - they are just an alias in `/etc/groups` or `/etc/passwd` back to UID+GID. Windows uses a larger binary security identifier (SID) which is stored in the Windows Security Access Manager (SAM) database. This database is not shared between the host and containers, or between containers.
- File permissions - Windows uses an access control list based on (SIDs), whereas POSIX systems such as Linux use a bitmask based on object permissions and UID+GID, plus *optional* access control lists.
- File paths - the convention on Windows is to use `\` instead of `/`. The Go IO libraries typically accept both and just make it work, but when you're setting a path or command line that's interpreted inside a container, `\` may be needed.
- Signals - Windows interactive apps handle termination differently, and can implement one or more of these:
  - A UI thread handles well-defined messages including `WM_CLOSE`.
  - Console apps handle Ctrl-C or Ctrl-break using a Control Handler.
  - Services register a Service Control Handler function that can accept `SERVICE_CONTROL_STOP` control codes.

Container exit codes follow the same convention where 0 is success, and nonzero is failure. The specific error codes may differ across Windows and Linux. However, exit codes passed from the Kubernetes components (kubelet, kube-proxy) are unchanged.

#### Field compatibility for container specifications

The following list documents differences between how Pod container specifications work between Windows and Linux:

- Huge pages are not implemented in the Windows container runtime, and are not available. They require asserting a user privilege that's not configurable for containers.
- `requests.cpu` and `requests.memory` - requests are subtracted from node available resources, so they can be used to avoid overprovisioning a node. However, they cannot be used to guarantee resources in an overprovisioned node. They should be applied to all containers as a best practice if the operator wants to avoid overprovisioning entirely.
- `securityContext.allowPrivilegeEscalation` - not possible on Windows; none of the capabilities are hooked up
- `securityContext.capabilities` - POSIX capabilities are not implemented on Windows
- `securityContext.privileged` - Windows doesn't support privileged containers, use HostProcess Containers instead
- `securityContext.procMount` - Windows doesn't have a `/proc` filesystem
- `securityContext.readOnlyRootFilesystem` - not possible on Windows; write access is required for registry & system processes to run inside the container
- `securityContext.runAsGroup` - not possible on Windows as there is no GID support
- `securityContext.runAsNonRoot` - this setting will prevent containers from running as `ContainerAdministrator` which is the closest equivalent to a root user on Windows.
- `securityContext.runAsUser` - use `runAsUserName` instead
- `securityContext.seLinuxOptions` - not possible on Windows as SELinux is Linux-specific
- `terminationMessagePath` - this has some limitations in that Windows doesn't support mapping single files. The default value is `/dev/termination-log`, which does work because it does not exist on Windows by default.

#### Field compatibility for Pod specifications

The following list documents differences between how Pod specifications work between Windows and Linux:

- `hostIPC` and `hostpid` - host namespace sharing is not possible on Windows
- `hostNetwork` - host networking is not possible on Windows
- `dnsPolicy` - setting the Pod `dnsPolicy` to `ClusterFirstWithHostNet` is not supported on Windows because host networking is not provided. Pods always run with a container network.
- `podSecurityContext` see below
- `shareProcessNamespace` - this is a beta feature, and depends on Linux namespaces which are not implemented on Windows. Windows cannot share process namespaces or the container's root filesystem. Only the network can be shared.
- `terminationGracePeriodSeconds` - this is not fully implemented in Docker on Windows, see the GitHub issue. The behavior today is that the ENTRYPOINT process is sent CTRL_SHUTDOWN_EVENT, then Windows waits 5 seconds by default, and finally shuts down all processes using the normal Windows shutdown behavior. The 5 second default is actually in the Windows registry inside the container, so it can be overridden when the container is built.
- `volumeDevices` - this is a beta feature, and is not implemented on Windows. Windows cannot attach raw block devices to pods.
- `volumes`
  - If you define an `emptyDir` volume, you cannot set its volume source to `memory`.
- You cannot enable `mountPropagation` for volume mounts as this is not supported on Windows.

#### Host network access

Kubernetes v1.26 to v1.32 included alpha support for running Windows Pods in the host's network namespace.

Kubernetes v1.34 does **not** include the `WindowsHostNetwork` feature gate or support for running Windows Pods in the host's network namespace.

#### Field compatibility for Pod security context

Only the `securityContext.runAsNonRoot` and `securityContext.windowsOptions` from the Pod `securityContext` fields work on Windows.

## Node problem detector

The node problem detector (see Monitor Node Health) has preliminary support for Windows. For more information, visit the project's GitHub page.

## Pause container

In a Kubernetes Pod, an infrastructure or "pause" container is first created to host the container. In Linux, the cgroups and namespaces that make up a pod need a process to maintain their continued existence; the pause process provides this. Containers that belong to the same pod, including infrastructure and worker containers, share a common network endpoint (same IPv4 and / or IPv6 address, same network port spaces). Kubernetes uses pause containers to allow for worker containers crashing or restarting without losing any of the networking configuration.

Kubernetes maintains a multi-architecture image that includes support for Windows. For Kubernetes v1.34.0 the recommended pause image is `registry.k8s.io/pause:3.6`. The source code is available on GitHub.

Microsoft maintains a different multi-architecture image, with Linux and Windows amd64 support, that you can find as `mcr.microsoft.com/oss/kubernetes/pause:3.6`. This image is built from the same source as the Kubernetes maintained image but all of the Windows binaries are authenticode signed by Microsoft. The Kubernetes project recommends using the Microsoft maintained image if you are deploying to a production or production-like environment that requires signed binaries.

## Container runtimes

You need to install a container runtime into each node in the cluster so that Pods can run there.

The following container runtimes work with Windows:

**Note:** This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

### ContainerD

FEATURE STATE: `Kubernetes v1.20 [stable]`

You can use ContainerD 1.4.0+ as the container runtime for Kubernetes nodes that run Windows.

Learn how to install ContainerD on a Windows node.

**Note:**

There is a known limitation when using GMSA with containerd to access Windows network shares, which requires a kernel patch.

### Mirantis Container Runtime

Mirantis Container Runtime (MCR) is available as a container runtime for all Windows Server 2019 and later versions.

See Install MCR on Windows Servers for more information.

## Windows OS version compatibility

On Windows nodes, strict compatibility rules apply where the host OS version must match the container base image OS version. Only Windows containers with a container operating system of Windows Server 2019 are fully supported.

For Kubernetes v1.34, operating system compatibility for Windows nodes (and Pods) is as follows:

Windows Server LTSC release
    Windows Server 2019
    Windows Server 2022
Windows Server SAC release
    Windows Server version 20H2

The Kubernetes version-skew policy also applies.

## Hardware recommendations and considerations

**Note:** This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

**Note:**

The following hardware specifications outlined here should be regarded as sensible default values. They are not intended to represent minimum requirements or specific recommendations for production environments. Depending on the requirements for your workload these values may need to be adjusted.

- 64-bit processor 4 CPU cores or more, capable of supporting virtualization
- 8GB or more of RAM
- 50GB or more of free disk space

Refer to Hardware requirements for Windows Server Microsoft documentation for the most up-to-date information on minimum hardware requirements. For guidance on deciding on resources for production worker nodes refer to Production worker nodes Kubernetes documentation.

To optimize system resources, if a graphical user interface is not required, it may be preferable to use a Windows Server OS installation that excludes the Windows Desktop Experience installation option, as this configuration typically frees up more system resources.

In assessing disk space for Windows worker nodes, take note that Windows container images are typically larger than Linux container images, with container image sizes ranging from 300MB to over 10GB for a single image. Additionally, take note that the `C:` drive in Windows containers represents a virtual free size of 20GB by default, which is not the actual consumed space, but rather the disk size for which a single container can grow to occupy when using local storage on the host. See Containers on Windows - Container Storage Documentation for more detail.

## Getting help and troubleshooting

Your main source of help for troubleshooting your Kubernetes cluster should start with the Troubleshooting page.

Some additional, Windows-specific troubleshooting help is included in this section. Logs are an important element of troubleshooting issues in Kubernetes. Make sure to include them any time you seek troubleshooting assistance from other contributors. Follow the instructions in the SIG Windows contributing guide on gathering logs.

### Reporting issues and feature requests

If you have what looks like a bug, or you would like to make a feature request, please follow the SIG Windows contributing guide to create a new issue. You should first search the list of issues in case it was reported previously and comment with your experience on the issue and add additional logs. SIG Windows channel on the Kubernetes Slack is also a great avenue to get some initial support and troubleshooting ideas prior to creating a ticket.

### Validating the Windows cluster operability

The Kubernetes project provides a *Windows Operational Readiness* specification, accompanied by a structured test suite. This suite is split into two sets of tests, core and extended, each containing categories aimed at testing specific areas. It can be used to validate all the functionalities of a Windows and hybrid system (mixed with Linux nodes) with full coverage.

To set up the project on a newly created cluster, refer to the instructions in the project guide.

## Deployment tools

The kubeadm tool helps you to deploy a Kubernetes cluster, providing the control plane to manage the cluster it, and nodes to run your workloads.

The Kubernetes cluster API project also provides means to automate deployment of Windows nodes.

## Windows distribution channels

For a detailed explanation of Windows distribution channels see the Microsoft documentation.

Information on the different Windows Server servicing channels including their support models can be found at Windows Server servicing channels.

# Pod Quality of Service Classes

This page introduces *Quality of Service (QoS) classes* in Kubernetes, and explains how Kubernetes assigns a QoS class to each Pod as a consequence of the resource constraints that you specify for the containers in that Pod. Kubernetes relies on this classification to make decisions about which Pods to evict when there are not enough available resources on a Node.

## Quality of Service classes

Kubernetes classifies the Pods that you run and allocates each Pod into a specific *quality of service (QoS) class*. Kubernetes uses that classification to influence how different pods are handled. Kubernetes does this classification based on the resource requests of the Containers in that Pod, along with how those requests relate to resource limits. This is known as Quality of Service (QoS) class. Kubernetes assigns every Pod a QoS class based on the resource requests and limits of its component Containers. QoS classes are used by Kubernetes to decide which Pods to evict from a Node experiencing Node Pressure. The possible QoS classes are `Guaranteed`, `Burstable`, and `BestEffort`. When a Node runs out of resources, Kubernetes will first evict `BestEffort` Pods running on that Node, followed by `Burstable` and finally `Guaranteed` Pods. When this eviction is due to resource pressure, only Pods exceeding resource requests are candidates for eviction.

### Guaranteed

Pods that are `Guaranteed` have the strictest resource limits and are least likely to face eviction. They are guaranteed not to be killed until they exceed their limits or there are no lower-priority Pods that can be preempted from the Node. They may not acquire resources beyond their specified limits. These Pods can also make use of exclusive CPUs using the `static` CPU management policy.

#### Criteria

For a Pod to be given a QoS class of `Guaranteed`:

- Every Container in the Pod must have a memory limit and a memory request.
- For every Container in the Pod, the memory limit must equal the memory request.
- Every Container in the Pod must have a CPU limit and a CPU request.
- For every Container in the Pod, the CPU limit must equal the CPU request.

### Burstable

Pods that are `Burstable` have some lower-bound resource guarantees based on the request, but do not require a specific limit. If a limit is not specified, it defaults to a limit equivalent to the capacity of the Node, which allows the Pods to flexibly increase their resources if resources are available. In the event of Pod eviction due to Node resource pressure, these Pods are evicted only after all `BestEffort` Pods are evicted. Because a `Burstable` Pod can include a Container that has no resource limits or requests, a Pod that is `Burstable` can try to use any amount of node resources.

#### Criteria

A Pod is given a QoS class of `Burstable` if:

- The Pod does not meet the criteria for QoS class `Guaranteed`.
- At least one Container in the Pod has a memory or CPU request or limit.

### BestEffort

Pods in the `BestEffort` QoS class can use node resources that aren't specifically assigned to Pods in other QoS classes. For example, if you have a node with 16 CPU cores available to the kubelet, and you assign 4 CPU cores to a `Guaranteed` Pod, then a Pod in the `BestEffort` QoS class can try to use any amount of the remaining 12 CPU cores.

The kubelet prefers to evict `BestEffort` Pods if the node comes under resource pressure.

#### Criteria

A Pod has a QoS class of `BestEffort` if it doesn't meet the criteria for either `Guaranteed` or `Burstable`. In other words, a Pod is `BestEffort` only if none of the Containers in the Pod have a memory limit or a memory request, and none of the Containers in the Pod have a CPU limit or a CPU request. Containers in a Pod can request other resources (not CPU or memory) and still be classified as `BestEffort`.

## Memory QoS with cgroup v2

FEATURE STATE: `Kubernetes v1.22 [alpha]` (enabled by default: false)

Memory QoS uses the memory controller of cgroup v2 to guarantee memory resources in Kubernetes. Memory requests and limits of containers in pod are used to set specific interfaces `memory.min` and `memory.high` provided by the memory controller. When `memory.min` is set to memory requests, memory resources are reserved and never reclaimed by the kernel; this is how Memory QoS ensures memory availability for Kubernetes pods. And if memory limits are set in the container, this means that the system needs to limit container memory usage; Memory QoS uses `memory.high` to throttle workload approaching its memory limit, ensuring that the system is not overwhelmed by instantaneous memory allocation.

Memory QoS relies on QoS class to determine which settings to apply; however, these are different mechanisms that both provide controls over quality of service.

## Some behavior is independent of QoS class

Certain behavior is independent of the QoS class assigned by Kubernetes. For example:

- Any Container exceeding a resource limit will be killed and restarted by the kubelet without affecting other Containers in that Pod.

- If a Container exceeds its resource request and the node it runs on faces resource pressure, the Pod it is in becomes a candidate for eviction. If this occurs, all Containers in the Pod will be terminated. Kubernetes may create a replacement Pod, usually on a different node.

- The resource request of a Pod is equal to the sum of the resource requests of its component Containers, and the resource limit of a Pod is equal to the sum of the resource limits of its component Containers.

- The kube-scheduler does not consider QoS class when selecting which Pods to preempt. Preemption can occur when a cluster does not have enough resources to run all the Pods you defined.

## What's next

- Learn about resource management for Pods and Containers.
- Learn about Node-pressure eviction.
- Learn about Pod priority and preemption.
- Learn about Pod disruptions.
- Learn how to assign memory resources to containers and pods.
- Learn how to assign CPU resources to containers and pods.
- Learn how to configure Quality of Service for Pods.

---

# DaemonSet

A DaemonSet defines Pods that provide node-local facilities. These might be fundamental to the operation of your cluster, such as a networking helper tool, or be part of an add-on.

A *DaemonSet* ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

Some typical uses of a DaemonSet are:

- running a cluster storage daemon on every node
- running a logs collection daemon on every node
- running a node monitoring daemon on every node

In a simple case, one DaemonSet, covering all nodes, would be used for each type of daemon. A more complex setup might use multiple DaemonSets for a single type of daemon, but with different flags and/or different memory and cpu requests for different hardware types.

## Writing a DaemonSet Spec

### Create a DaemonSet

You can describe a DaemonSet in a YAML file. For example, the `daemonset.yaml` file below describes a DaemonSet that runs the fluentd-elasticsearch Docker image:

`controllers/daemonset.yaml` Copy controllers/daemonset.yaml to clipboard

```
apiVersion: apps/v1
kind: DaemonSetmetadata:  name: fluentd-elasticsearch  namespace: kube-system  labels:    k8s-app: fluentd-loggingspec:  selector:
```

Create a DaemonSet based on the YAML file:

```
kubectl apply -f https://k8s.io/examples/controllers/daemonset.yaml
```

### Required Fields

As with all other Kubernetes config, a DaemonSet needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see running stateless applications and object management using kubectl.

The name of a DaemonSet object must be a valid DNS subdomain name.

A DaemonSet also needs a `.spec` section.

### Pod Template

The `.spec.template` is one of the required fields in `.spec`.

The `.spec.template` is a pod template. It has exactly the same schema as a Pod, except it is nested and does not have an `apiVersion` or `kind`.

In addition to required fields for a Pod, a Pod template in a DaemonSet has to specify appropriate labels (see pod selector).

A Pod Template in a DaemonSet must have a `RestartPolicy` equal to `Always`, or be unspecified, which defaults to `Always`.

### Pod Selector

The `.spec.selector` field is a pod selector. It works the same as the `.spec.selector` of a Job.

You must specify a pod selector that matches the labels of the `.spec.template`. Also, once a DaemonSet is created, its `.spec.selector` can not be mutated. Mutating the pod selector can lead to the unintentional orphaning of Pods, and it was found to be confusing to users.

The `.spec.selector` is an object consisting of two fields:

- `matchLabels` - works the same as the `.spec.selector` of a ReplicationController.
- `matchExpressions` - allows to build more sophisticated selectors by specifying key, list of values and an operator that relates the key and values.

When the two are specified the result is ANDed.

The `.spec.selector` must match the `.spec.template.metadata.labels`. Config with these two not matching will be rejected by the API.

### Running Pods on select Nodes

If you specify a `.spec.template.spec.nodeSelector`, then the DaemonSet controller will create Pods on nodes which match that node selector. Likewise if you specify a `.spec.template.spec.affinity`, then DaemonSet controller will create Pods on nodes which match that node affinity. If you do not specify either, then the DaemonSet controller will create Pods on all nodes.

## How Daemon Pods are scheduled

A DaemonSet can be used to ensure that all eligible nodes run a copy of a Pod. The DaemonSet controller creates a Pod for each eligible node and adds the `spec.affinity.nodeAffinity` field of the Pod to match the target host. After the Pod is created, the default scheduler typically takes over and then binds the Pod to the target host by setting the `.spec.nodeName` field. If the new Pod cannot fit on the node, the default scheduler may preempt (evict) some of the existing Pods based on the priority of the new Pod.

**Note:**

If it's important that the DaemonSet pod run on each node, it's often desirable to set the `.spec.template.spec.priorityClassName` of the DaemonSet to a PriorityClass with a higher priority to ensure that this eviction occurs.

The user can specify a different scheduler for the Pods of the DaemonSet, by setting the `.spec.template.spec.schedulerName` field of the DaemonSet.

The original node affinity specified at the `.spec.template.spec.affinity.nodeAffinity` field (if specified) is taken into consideration by the DaemonSet controller when evaluating the eligible nodes, but is replaced on the created Pod with the node affinity that matches the name of the eligible node.

```
nodeAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    nodeSelectorTerms:
    - matchFields:
      - key: metadata.name
        operator: In
        values:
        - target-host-name
```

### Taints and tolerations

The DaemonSet controller automatically adds a set of tolerations to DaemonSet Pods:

| Toleration key | Effect | Details |
|---|---|---|
| `node.kubernetes.io/not-ready` | `NoExecute` | DaemonSet Pods can be scheduled onto nodes that are not healthy or ready to accept Pods. Any DaemonSet Pods running on such nodes will not be evicted. |
| `node.kubernetes.io/unreachable` | `NoExecute` | DaemonSet Pods can be scheduled onto nodes that are unreachable from the node controller. Any DaemonSet Pods running on such nodes will not be evicted. |
| `node.kubernetes.io/disk-pressure` | `NoSchedule` | DaemonSet Pods can be scheduled onto nodes with disk pressure issues. |
| `node.kubernetes.io/memory-pressure` | `NoSchedule` | DaemonSet Pods can be scheduled onto nodes with memory pressure issues. |
| `node.kubernetes.io/pid-pressure` | `NoSchedule` | DaemonSet Pods can be scheduled onto nodes with process pressure issues. |
| `node.kubernetes.io/unschedulable` | `NoSchedule` | DaemonSet Pods can be scheduled onto nodes that are unschedulable. |
| `node.kubernetes.io/network-unavailable` | `NoSchedule` | **Only added for DaemonSet Pods that request host networking**, i.e., Pods having `spec.hostNetwork: true`. Such DaemonSet Pods can be scheduled onto nodes with unavailable network. |

You can add your own tolerations to the Pods of a DaemonSet as well, by defining these in the Pod template of the DaemonSet.

Because the DaemonSet controller sets the `node.kubernetes.io/unschedulable:NoSchedule` toleration automatically, Kubernetes can run DaemonSet Pods on nodes that are marked as *unschedulable*.

If you use a DaemonSet to provide an important node-level function, such as cluster networking, it is helpful that Kubernetes places DaemonSet Pods on nodes before they are ready. For example, without that special toleration, you could end up in a deadlock situation where the node is not marked as ready because the network plugin is not running there, and at the same time the network plugin is not running on that node because the node is not yet ready.

## Communicating with Daemon Pods

Some possible patterns for communicating with Pods in a DaemonSet are:

- **Push**: Pods in the DaemonSet are configured to send updates to another service, such as a stats database. They do not have clients.
- **NodeIP and Known Port**: Pods in the DaemonSet can use a `hostPort`, so that the pods are reachable via the node IPs. Clients know the list of node IPs somehow, and know the port by convention.
- **DNS**: Create a headless service with the same pod selector, and then discover DaemonSets using the `endpoints` resource or retrieve multiple A records from DNS.
- **Service**: Create a service with the same Pod selector, and use the service to reach a daemon on a random node. Use Service Internal Traffic Policy to limit to pods on the same node.

## Updating a DaemonSet

If node labels are changed, the DaemonSet will promptly add Pods to newly matching nodes and delete Pods from newly not-matching nodes.

You can modify the Pods that a DaemonSet creates. However, Pods do not allow all fields to be updated. Also, the DaemonSet controller will use the original template the next time a node (even with the same name) is created.

You can delete a DaemonSet. If you specify `--cascade=orphan` with `kubectl`, then the Pods will be left on the nodes. If you subsequently create a new DaemonSet with the same selector, the new DaemonSet adopts the existing Pods. If any Pods need replacing the DaemonSet replaces them according to its `updateStrategy`.

You can perform a rolling update on a DaemonSet.

## Alternatives to DaemonSet

### Init scripts

It is certainly possible to run daemon processes by directly starting them on a node (e.g. using `init`, `upstartd`, or `systemd`). This is perfectly fine. However, there are several advantages to running such processes via a DaemonSet:

- Ability to monitor and manage logs for daemons in the same way as applications.
- Same config language and tools (e.g. Pod templates, `kubectl`) for daemons and applications.
- Running daemons in containers with resource limits increases isolation between daemons from app containers. However, this can also be accomplished by running the daemons in a container but not in a Pod.

### Bare Pods

It is possible to create Pods directly which specify a particular node to run on. However, a DaemonSet replaces Pods that are deleted or terminated for any reason, such as in the case of node failure or disruptive node maintenance, such as a kernel upgrade. For this reason, you should use a DaemonSet rather than creating individual Pods.

### Static Pods

It is possible to create Pods by writing a file to a certain directory watched by Kubelet. These are called static pods. Unlike DaemonSet, static Pods cannot be managed with kubectl or other Kubernetes API clients. Static Pods do not depend on the apiserver, making them useful in cluster bootstrapping cases. Also, static Pods may be deprecated in the future.

### Deployments

DaemonSets are similar to Deployments in that they both create Pods, and those Pods have processes which are not expected to terminate (e.g. web servers, storage servers).

Use a Deployment for stateless services, like frontends, where scaling up and down the number of replicas and rolling out updates are more important than controlling exactly which host the Pod runs on. Use a DaemonSet when it is important that a copy of a Pod always run on all or certain hosts, if the DaemonSet provides node-level functionality that allows other Pods to run correctly on that particular node.

For example, network plugins often include a component that runs as a DaemonSet. The DaemonSet component makes sure that the node where it's running has working cluster networking.

## What's next

- Learn about Pods:
  - Learn about static Pods, which are useful for running Kubernetes control plane components.
- Find out how to use DaemonSets:
  - Perform a rolling update on a DaemonSet.
  - Perform a rollback on a DaemonSet (for example, if a roll out didn't work how you expected).
- Understand how Kubernetes assigns Pods to Nodes.
- Learn about device plugins and add ons, which often run as DaemonSets.
- `DaemonSet` is a top-level resource in the Kubernetes REST API. Read the DaemonSet object definition to understand the API for daemon sets.