**Deployments**

A *Deployment* provides declarative updates for [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) and [ReplicaSets](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/).

You describe a *desired state* in a Deployment, and the Deployment [Controller](https://kubernetes.io/docs/concepts/architecture/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](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a-deployment). 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](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#updating-a-deployment) by updating the PodTemplateSpec of the Deployment. A new ReplicaSet is created and the Deployment manages moving the Pods from the old ReplicaSet to the new one at a controlled rate. Each new ReplicaSet updates the revision of the Deployment.
* [Rollback to an earlier Deployment revision](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#rolling-back-a-deployment) 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](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#scaling-a-deployment).
* [Pause the rollout of a Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#pausing-and-resuming-a-deployment) to apply multiple fixes to its PodTemplateSpec and then resume it to start a new rollout.
* [Use the status of the Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#deployment-status) as an indicator that a rollout has stuck.
* [Clean up older ReplicaSets](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#clean-up-policy) 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](https://raw.githubusercontent.com/kubernetes/website/main/content/en/examples/controllers/nginx-deployment.yaml)

**apiVersion**: apps/v1

**kind**: Deployment

**metadata**:

**name**: nginx-deployment

**labels**:

**app**: nginx

**spec**:

**replicas**: 3

**selector**:

**matchLabels**:

**app**: nginx

**template**:

**metadata**:

**labels**:

**app**: nginx

**spec**:

**containers**:

- **name**: nginx

**image**: nginx:1.14.2

**ports**:

- **containerPort**: 80

In this example:

* A Deployment named nginx-deployment is created, indicated by the .metadata.name field.
* 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 template field contains the following sub-fields:
  + The Pods are labeled app: nginxusing the .metadata.labels field.
  + The Pod template's specification, or .template.spec field, indicates that the Pods run one container, nginx, which runs the nginx [Docker Hub](https://hub.docker.com/) image at version 1.14.2.
  + Create one container and name it nginx using the .spec.template.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:
2. kubectl apply -f https://k8s.io/examples/controllers/nginx-deployment.yaml
3. 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.

1. 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

1. Run the kubectl get deployments again a few seconds later. The output is similar to this:
2. NAME READY UP-TO-DATE AVAILABLE AGE
3. 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.

1. To see the ReplicaSet (rs) created by the Deployment, run kubectl get rs. The output is similar to this:
2. NAME DESIRED CURRENT READY AGE
3. 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]. The HASH string is the same as the pod-template-hash label on the ReplicaSet.

1. To see the labels automatically generated for each Pod, run kubectl get pods --show-labels. The output is similar to:
2. NAME READY STATUS RESTARTS AGE LABELS
3. nginx-deployment-75675f5897-7ci7o 1/1 Running 0 18s app=nginx,pod-template-hash=3123191453
4. nginx-deployment-75675f5897-kzszj 1/1 Running 0 18s app=nginx,pod-template-hash=3123191453
5. nginx-deployment-75675f5897-qqcnn 1/1 Running 0 18s app=nginx,pod-template-hash=3123191453

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.
2. 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

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

1. To see the rollout status, run:
2. 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 are 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](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#deployment-status).
* You see that the number of old replicas (nginx-deployment-1564180365 and nginx-deployment-2035384211) is 2, and new replicas (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 nginx-deployment-2035384211 to 3

22s 22s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-1564180365 to 1

22s 22s 1 {deployment-controller } Normal ScalingReplicaSet Scaled down replica set nginx-deployment-2035384211 to 2

22s 22s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-1564180365 to 2

21s 21s 1 {deployment-controller } Normal ScalingReplicaSet Scaled down replica set nginx-deployment-2035384211 to 1

21s 21s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-1564180365 to 3

13s 13s 1 {deployment-controller } Normal ScalingReplicaSet Scaled down replica set nginx-deployment-2035384211 to 0

13s 13s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-3066724191 to 1

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:
2. kubectl rollout history deployment/nginx-deployment

The output is similar to this:

deployments "nginx-deployment"

REVISION CHANGE-CAUSE

1 kubectl apply --filename=https://k8s.io/examples/controllers/nginx-deployment.yaml

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

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

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

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

Annotations: kubernetes.io/change-cause=kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1

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:
2. 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](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#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.

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

1. Get the description of the Deployment:
2. 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

kubernetes.io/change-cause=kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1

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](https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/) 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](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#max-surge)=3, and [maxUnavailable](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#max-unavailable)=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 -w

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](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#progressing-deployment) while rolling out a new ReplicaSet, it can be [complete](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#complete-deployment), or it can [fail to progress](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#failed-deployment).

**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](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#min-ready-seconds)).

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](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#progress-deadline-seconds)). .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](https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#typical-status-properties) 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.

**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](https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/#canary-deployments).

**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](https://kubernetes.io/docs/tasks/run-application/run-stateless-application-deployment/), configuring containers, and [using kubectl to manage resources](https://kubernetes.io/docs/concepts/overview/working-with-objects/object-management/) documents. The name of a Deployment object must be a valid [DNS subdomain name](https://kubernetes.io/docs/concepts/overview/working-with-objects/names#dns-subdomain-names).

A Deployment also needs a [.spec section](https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status).

**Pod Template**

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

The .spec.template is a [Pod template](https://kubernetes.io/docs/concepts/workloads/pods/#pod-templates). It has exactly the same schema as a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/), 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](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#selector).

Only a [.spec.template.spec.restartPolicy](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy) 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](https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/) (or any similar API for horizontal scaling) is managing scaling for a Deployment, don't set .spec.replicas.

Instead, allow the Kubernetes [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) to manage the .spec.replicas field automatically.

**Selector**

.spec.selector is a required field that specifies a [label selector](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/) 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](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/).

**Rolling Update Deployment**

The Deployment updates Pods in a rolling update fashion 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.

**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](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#failed-deployment) - 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](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#container-probes).

**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.