

# Agenda

---

## **Deployments**

Jobs and CronJobs

Cluster Scaling

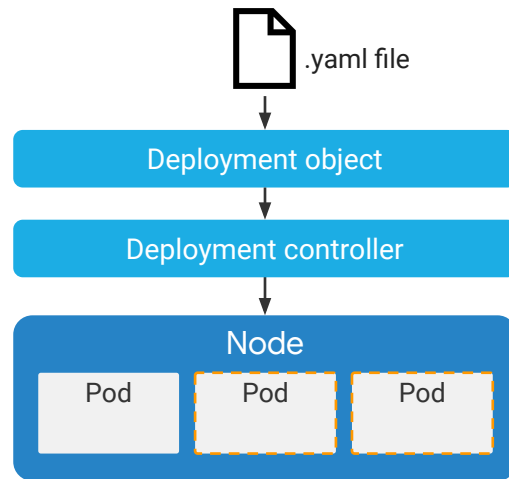
Controlling Pod Placement

Getting Software into Your Cluster



Deployments describe a desired state of Pods. For example, a desired state could be that you want to make sure that 5 nginx pods are running at all times. Its declarative stance means that Kubernetes will continuously make sure the configuration is running across your cluster. Kubernetes also supports various update mechanisms for Deployments, which I'll tell you about later in this module.

## Deployment is a two-part process



The desired state is described in a Deployment YAML file containing the characteristics of the pods, coupled with how to operationally run these pods and handle their lifecycle events. After you submit this file to the Kubernetes master, it creates a deployment controller, which is responsible for converting the desired state into reality and keeping that desired state over time. Remember what a controller is: it's a loop process created by Kubernetes that takes care of routine tasks to ensure the desired state of an object, or set of objects, running on the cluster matches the observed state.

During this process, a ReplicaSet is created. A ReplicaSet is a controller that ensures that a certain number of Pod replicas are running at any given time. The Deployment is a high level controller for a Pod that declares its state. The Deployment configures a ReplicaSet controller to instantiate and maintain a specific version

of the Pods specified in the Deployment.

## Deployments declare the state of of Pods



Roll out updates  
to the Pods



Roll back Pods  
to previous  
revision



Scale or  
autoscale Pods



Well-suited for  
stateless  
applications



Every time you update the specification of the pods, for example, updating them to use a newer container image, a new ReplicaSet is created that matches the altered version of the Deployment. This is how deployments roll out updated Pods in a controlled manner: old Pods from the old ReplicaSet are replaced with newer Pods in a new ReplicaSet.

If the updated Pods are not stable, the administrator can roll back the Pod to a previous Deployment revision.

You can scale Pods manually by modifying the Deployment configuration. You can also configure the Deployment to manage the workload automatically.

Deployments are designed for stateless applications. Stateless applications don't store data or application state to a cluster or to

persistent storage. A typical example of a stateless application is a Web front end. Some backend owns the problem of making sure that data gets stored durably, and you'll use Kubernetes objects other than Deployments to manage these back ends.

## Deployment object file in YAML format

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app
        image: gcr.io/demo/my-app:1.0
        ports:
        - containerPort: 8080
```



Here is a simple example of a Deployment object file in YAML format.

The Deployment named my-app is created with 3 replicated Pods. In the spec.template section, a Pod template defines the metadata and specification of each of the Pods in this ReplicaSet.

In the Pod specification, an image is pulled from Google Container Registry, and port 8080 is exposed to send and accept traffic for the container.

## There are three ways to create a Deployment

1

```
$ kubectl apply -f [DEPLOYMENT_FILE]
```

2

```
$ kubectl run [DEPLOYMENT_NAME] \  
  --image [IMAGE]:[TAG] \  
  --replicas 3 \  
  --labels [KEY]=[VALUE] \  
  --port 8080 \  
  --generator deployment/apps.v1 \  
  --save-config
```



You can create a Deployment in three different ways. First, you create the Deployment declaratively using a manifest file, such as the YAML file you've just seen, and a `kubectl apply` command.

The second method creates a Deployment imperatively using a `kubectl run` command that specifies the parameters inline. Here, the image and tag specifies which image and image version to run in the container. This Deployment will launch 3 replicas and expose port 8080. Labels are defined using key and value. `--generator` specifies the API version to be used, and `--save-config` saves the configuration for future use.

← Create a deployment

A deployment is a configuration which defines how Kubernetes deploys, manages, and scales your container image. Kubernetes will ensure your system matches this configuration.

Deployment

Container

Container image

nginx:latest

Select Google Container Registry image

Environment variables

+ Add environment variable

Initial command (Optional)

Done

Cancel

+ Add container

3

Application name

nginx-1

Namespace

default

Labels

Key	Value
app	nginx-1

+ Add label

Cluster

Your deployment will use compute instances managed in a logical grouping called a "cluster", which will be configured in a way that's great for getting started with Kubernetes.

The cluster will be named nginx-1-cluster

Zone

us-central1-a

Deploy

View YAML

Your third option is to use the GKE Workloads menu in the GCP Console. Here, you can specify the container image and version, or even select it directly from Container Registry. You can specify environment variables and initialization commands. You can also add an application name and namespace along with labels. You can use the View YAML button on the last page of the Deployment wizard to view the Deployment specification in YAML format.



Use kubectl to inspect your Deployment, or output the Deployment config in a YAML format

```
$ kubectl get deployment [DEPLOYMENT_NAME]
```

```
master $ kubectl get deployment nginx-deployment
```

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

```
$ kubectl get deployment [DEPLOYMENT_NAME] -o yaml > this.yaml
```



The ReplicaSet created by the Deployment ensures that the desired number of Pods are running and always available at any given time. If a Pod fails or is evicted, the ReplicaSet automatically launches a new Pod. You can use the kubectl 'get' and 'describe' commands to inspect the state and details of the Deployment.

As shown here, you can get the desired, current, up-to-date, and available status of all the replicas within a Deployment, along with their ages, using the kubectl 'get deployment' command.

'Desired' shows the desired number of replicas in the Deployment specification.

'Current' is the number of replicas currently running.

'Up-to-date' shows the number of replicas that are fully up to date as per the current Deployment specification.

'Available' displays the number of replicas available to the users.

You can also output the Deployment configuration in a YAML format. Maybe you originally created a Deployment with `kubectl` run, and then you decide you'd like to make it a permanent, managed part of your infrastructure. Edit that YAML file to remove the unique details of the Deployment you created it from, and then you can add it to your repository of YAML files for future Deployments.

## Use the 'describe' command to get detailed info

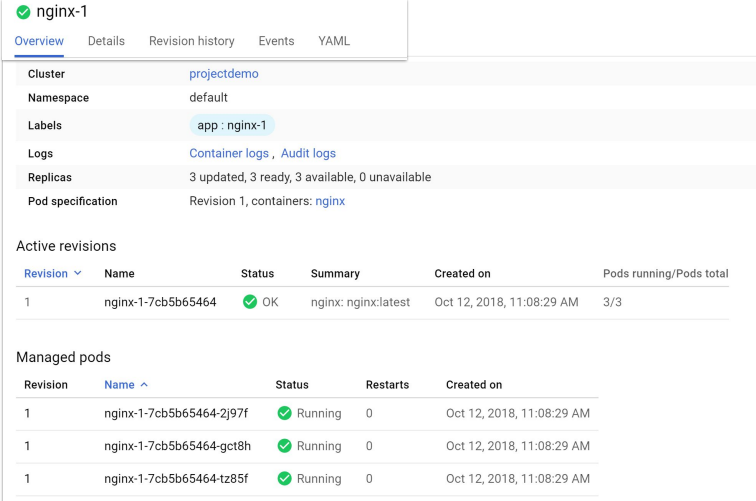
```
$ kubectl describe deployment [DEPLOYMENT_NAME]
```

```
master $ kubectl describe deployment nginx-deployment
Name:          nginx-deployment
Namespace:     default
CreationTimestamp:  Fri, 12 Oct 2018 15:23:46 +0000
Labels:        app=nginx
Annotations:   deployment.kubernetes.io/revision=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:1.15.4
      Port:    80/TCP
      Host Port:  0/TCP
```



For more detailed information about the Deployment, use the kubectl 'describe' command. You'll learn more about this command in the lab.

## Or use the GCP Console



The screenshot displays the GCP Console interface for a Deployment named 'nginx-1'. The top navigation bar includes tabs for Overview, Details, Revision history, Events, and YAML. The Overview tab is selected, showing a summary of the Deployment's configuration and status.

**Deployment Summary:**

- Cluster: projectdemo
- Namespace: default
- Labels: app: nginx-1
- Logs: Container logs, Audit logs
- Replicas: 3 updated, 3 ready, 3 available, 0 unavailable
- Pod specification: Revision 1, containers: nginx

**Active revisions:**

Revision	Name	Status	Summary	Created on	Pods running/Pods total
1	nginx-1-7cb5b65464	OK	nginx: nginx:latest	Oct 12, 2018, 11:08:29 AM	3/3

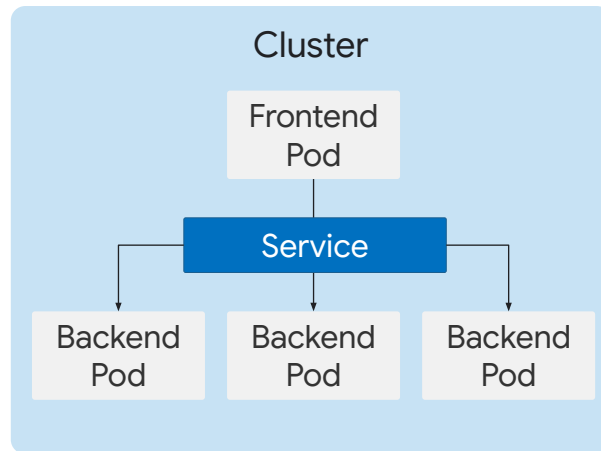
**Managed pods:**

Revision	Name	Status	Restarts	Created on
1	nginx-1-7cb5b65464-2j97f	Running	0	Oct 12, 2018, 11:08:29 AM
1	nginx-1-7cb5b65464-gct8h	Running	0	Oct 12, 2018, 11:08:29 AM
1	nginx-1-7cb5b65464-tz85f	Running	0	Oct 12, 2018, 11:08:29 AM



Another way to inspect a Deployment is to use the GCP Console. Here you can see detailed information about the Deployment, revision history, the Pods, events, and also view the live configuration in YAML format.

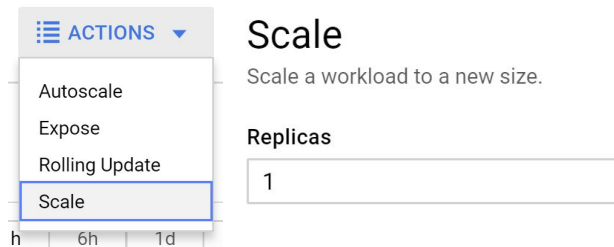
Service is a stable network representation of a set of pods



We haven't yet discussed how to locate and connect to the applications running in these Pods, especially as new Pods are created or updated by your Deployments. While you can connect to individual Pods directly, Pods themselves are transient. A Kubernetes Service is a static IP address that represents a Service, or a function, in your infrastructure. It's a stable network abstraction for a set of Pods that deliver that Service and, and it hides the ephemeral nature of the individual Pods. Services will be covered in detail in the Networking module.

## You can scale the Deployment manually

```
$ kubectl scale deployment  
[DEPLOYMENT_NAME] -replicas=5
```



The screenshot shows the 'Scale' interface in the GCP Console. On the left, there is a sidebar with a menu icon and the word 'ACTIONS' with a dropdown arrow. The dropdown menu is open, showing four options: 'Autoscale', 'Expose', 'Rolling Update', and 'Scale'. The 'Scale' option is highlighted with a blue border. Below the menu, there are time range filters: 'h', '6h', and '1d'. On the right, the title 'Scale' is displayed, followed by the subtitle 'Scale a workload to a new size.' Below this, the label 'Replicas' is shown next to a text input field containing the number '1'.



You now understand that a Deployment will maintain the desired number of replicas for an application. However, at some point you'll probably need to scale the Deployment. Maybe you need more web front end instances, for example. You can scale the Deployment manually using a `kubectl` command, or in the GCP Console by defining the total number of replicas. Also, manually changing the manifest will scale the Deployment.

## You can also autoscale the Deployment

```
$ kubectl autoscale deployment [DEPLOYMENT_NAME] --min=5 --max=15  
--cpu-percent=75
```

### Autoscale

Automatically scale the number of pods.

Minimum number of Pods (Optional)

Maximum number of Pods

Target CPU utilization in percent (Optional)

[CANCEL](#) [DISABLE AUTOSCALER](#) [AUTOSCALE](#)



You can also autoscale the Deployment by specifying the minimum and maximum number of desired Pods along with a CPU utilization threshold. Again, you can perform autoscaling by using the `kubectl autoscale` command, or from the GCP Console directly. This leads to the creation of a Kubernetes object called `HorizontalPodAutoscaler`. This object performs the actual scaling to match the target CPU utilization. Keep in mind that we're not scaling the cluster as a whole, just a particular Deployment within that cluster. Later in this module you'll learn how to scale clusters.

One problem of any type of autoscaling is thrashing

Cooldown/delay support:

```
--horizontal-pod-autoscaler-downscale-delay
```



Thrashing sounds bad, and it is bad. It's a phenomenon where the number of deployed replicas frequently fluctuates, because the metric you used to control scaling also frequently fluctuates. The Horizontal Pod Autoscaler supports a cooldown, or delay, feature. It allows you to specify a wait period before performing another scale-down action. The default value is 5 minutes.



## You can update a Deployment in different ways

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  template:
    spec:
      containers:
        - name: my-app
          image: gcr.io/demo/my-app:1.0
          ports:
            - containerPort: 8080
```

```
$ kubectl apply -f [DEPLOYMENT_FILE]
```

```
$ kubectl set image deployment
[DEPLOYMENT_NAME] [IMAGE] [IMAGE]:[TAG]
```

```
$ kubectl edit \
  deployment/[DEPLOYMENT_NAME]
```



When you make a change to a Deployment's Pod specification, such as changing the image version, an automatic update rollout happens. Again, note that these automatic updates are only applicable to the changes in Pod specifications.

You can update a Deployment in different ways. One way is to use the kubectl 'apply' command with an updated Deployment specification YAML file. This method allows you to update other specifications of a Deployment, such as the number of replicas, outside the Pod template.

You can also use a kubectl 'set' command. This allows you to change the Pod template specifications for the Deployment, such as the image, resources, and selector values.

Another way is to use a kubectl 'edit' command. This opens the

specification file using the vim editor that allows you to make changes directly. Once you exit and save the file, kubectl automatically applies the updated file.

## You can update a Deployment in different ways

[REFRESH](#) [EDIT](#) [DELETE](#) [ACTIONS](#) [KUBECTL](#)

### Rolling update

Update workload Pods to a new application version.

**Minimum seconds ready** ⓘ (Optional)

**Maximum surge** ⓘ (Optional)

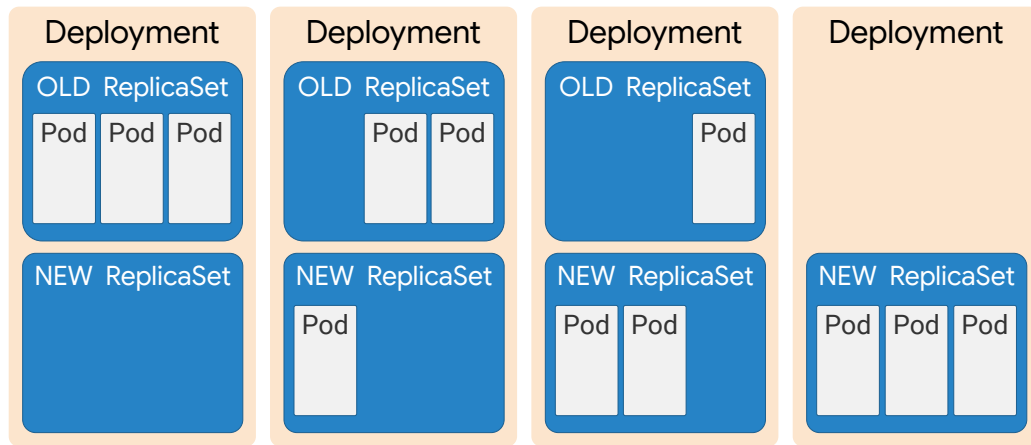
**Maximum unavailable** ⓘ (Optional)

Container name	Image
nginx	<input type="text" value="nginx:latest"/>



The last option for you to update a Deployment is through the GCP Console. You can edit the Deployment manifest from the GCP Console and perform a rolling update along with its additional options. Rolling updates are discussed next.

## The process behind updating a Deployment



When a Deployment is updated, it launches a new ReplicaSet and creates a new set of Pods in a controlled fashion.

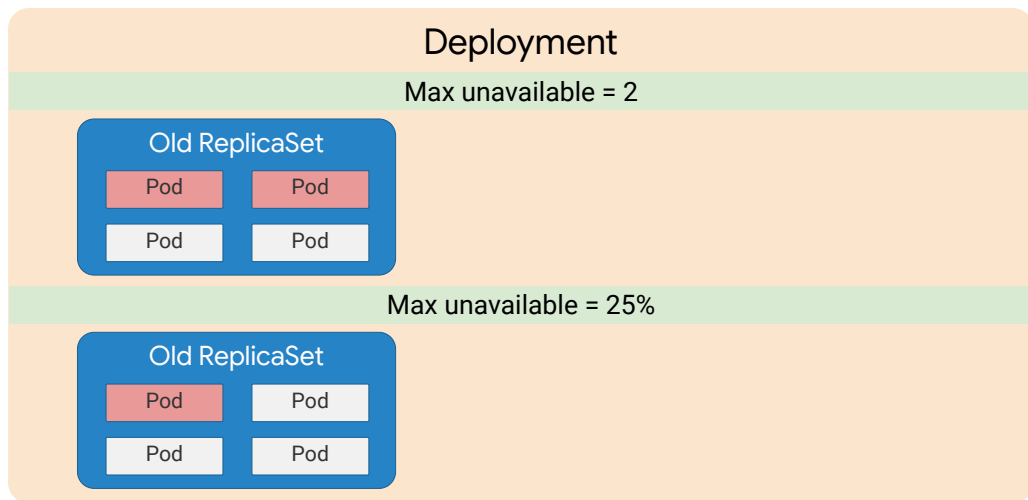
First, new Pods are launched in a new ReplicaSet.

Next, old Pods are deleted from the old ReplicaSet.

This is an example of a rolling update strategy also known as a ramped strategy.

Its advantage is that updates are slowly released, which ensures the availability of the application. However, this process can take time, and there's no control over how the traffic is directed to the old and new Pods.

## Set parameters for your rolling update strategy

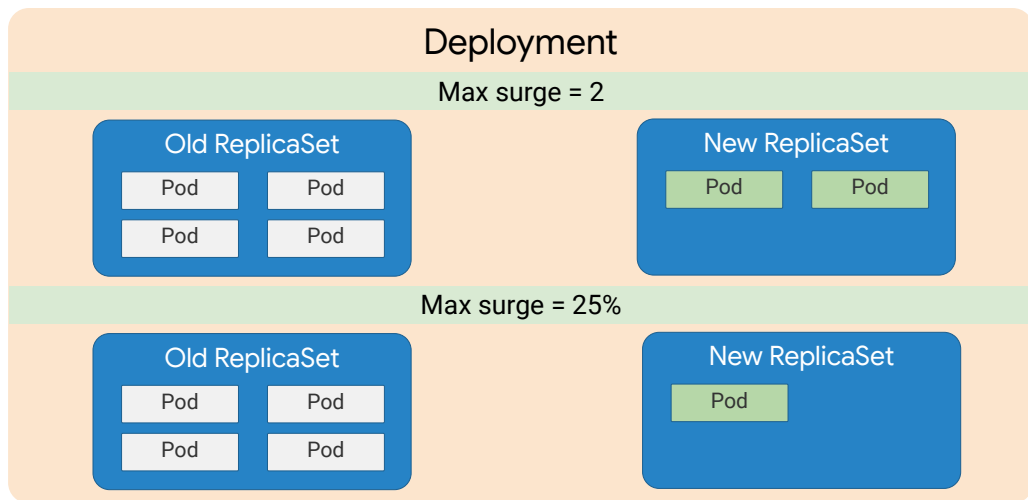


In a rolling update strategy, the max unavailable and max surge fields can be used to control how the Pods are updated. These fields define a range for the total number of running Pods within the Deployment, regardless of whether they are in the old or new ReplicaSets.

The max unavailable field lets you specify the maximum number of Pods that can be unavailable during the rollout process. This number can be either absolute or a percentage. For example, you can say you want to have no more than 2 Pods unavailable during the upgrade process.

Specifying max unavailable at 25% means you want to have at least 75% of the total desired number of Pods running at the same time. The default max unavailable is 25%.

## Set parameters for your rolling update strategy



Max surge allows you to specify the maximum number of extra Pods that can be created concurrently in a new ReplicaSet. For example, you can specify that you want to add up to two new Pods at a time in a new ReplicaSet, and the Deployment controller will do exactly that.

You can also set max surge as a percentage. The Deployment controller looks at the total number of running Pods in both ReplicaSets, Old and New.

In this example, a Deployment with the desired number of Pods as 4 and a max surge of 25% will allow a maximum total of 5 Pods running at any given time between the old and new ReplicaSets. In other words, it'll allow 125% of the desired number of Pods, which is 5. Again, the default max surge is 25%.

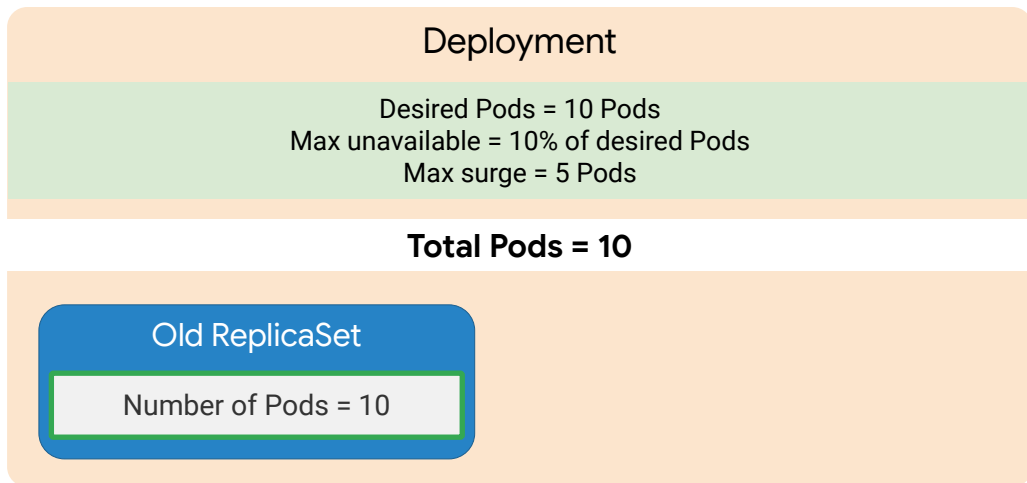
## An example of a rolling update strategy

```
[...]
kind: deployment
spec:
  replicas: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 5
      maxUnavailable: 30%
[...]
```



Let's look at a Deployment with a desired number of Pods set to 10, max unavailable set to 10%, and max surge set to 5.

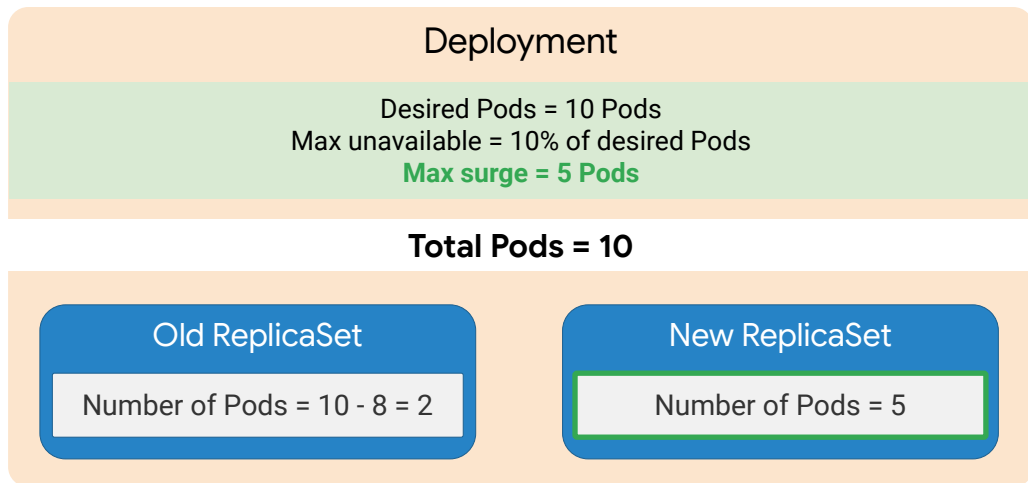
## An example of a rolling update strategy



The Old ReplicaSet has 10 Pods.

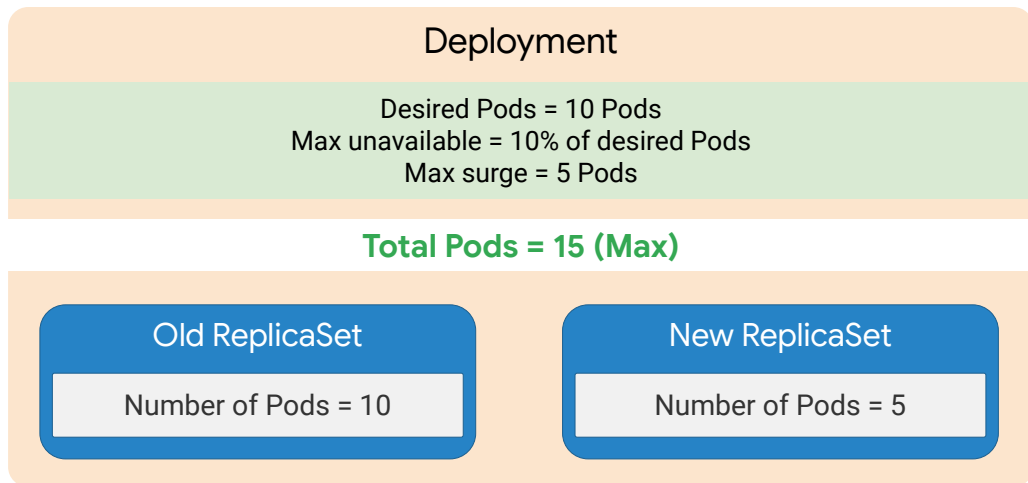


## An example of a rolling update strategy



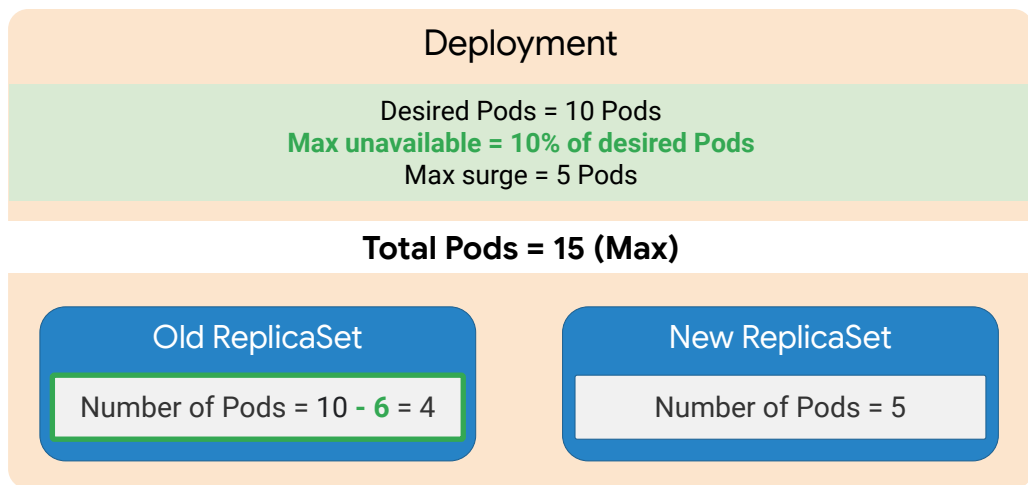
The Deployment will begin by creating 5 new Pods in a new ReplicaSet based on max surge.

## An example of a rolling update strategy



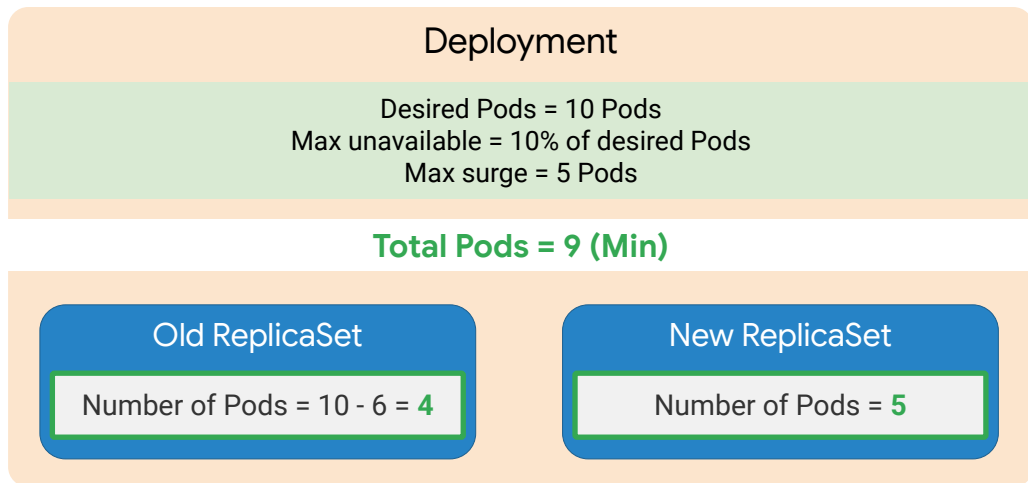
When those new Pods are ready, the total number of Pods changes to 15.

## An example of a rolling update strategy



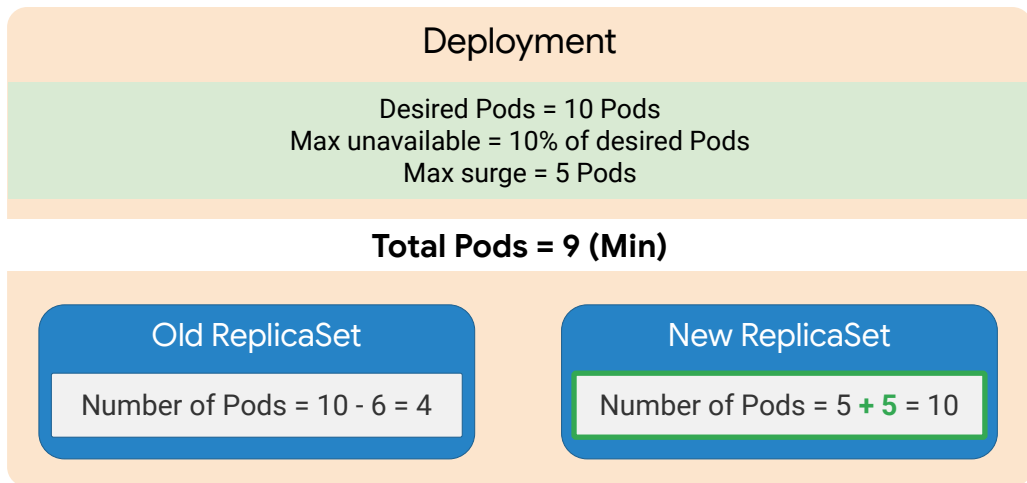
With max unavailable set to 10%, the minimum number of Pods running, regardless of old or new ReplicaSet, is 10 minus 10%, which equates to 9 Pods. So the rollout can remove old Pods until the overall total is no lower than 9 Pods. This means that 6 Pods can be removed from the old ReplicaSet at this stage.

## An example of a rolling update strategy



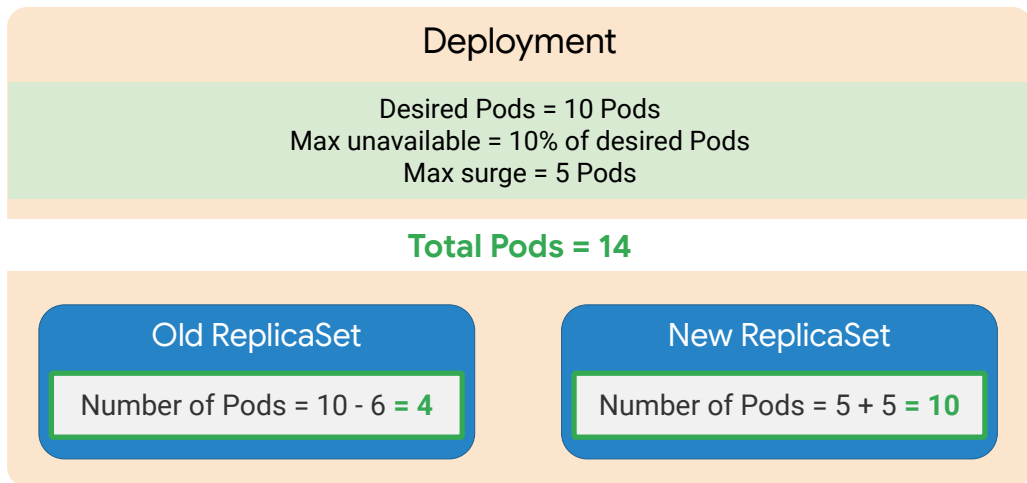
This keeps the minimum total at 9 Pods: 5 in the new ReplicaSet, and 4 in the old ReplicaSet.

## An example of a rolling update strategy



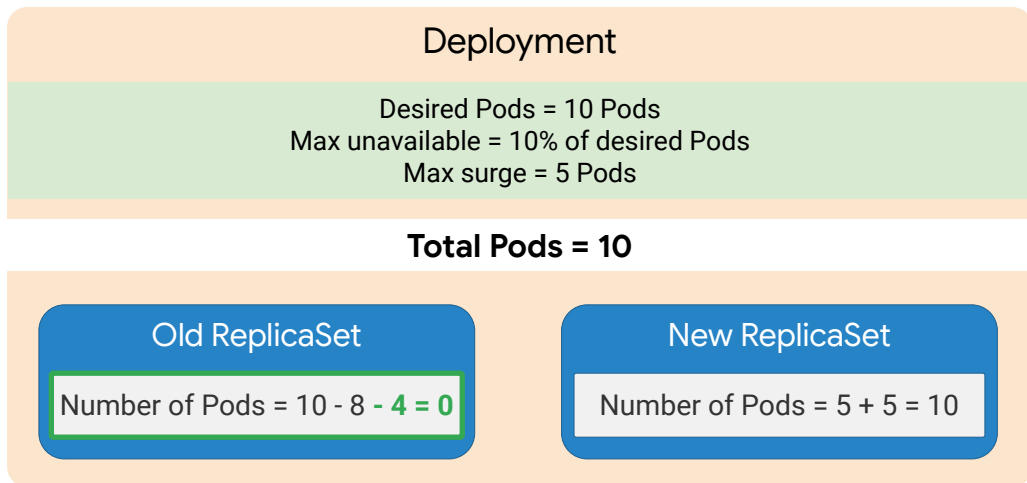
Next, 5 more Pods are launched in the new ReplicaSet.

## An example of a rolling update strategy



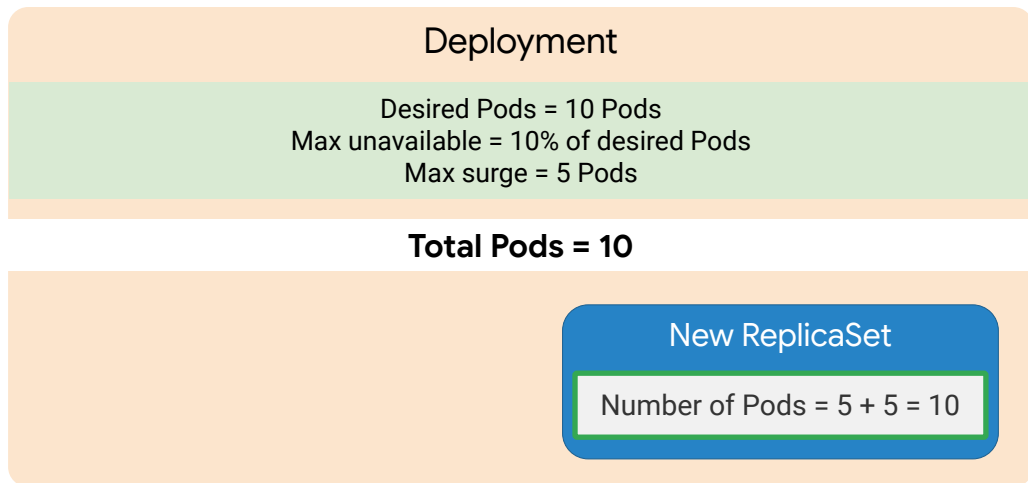
This creates a total of 10 Pods in a new ReplicaSet and a total of 14 across all ReplicaSets.

## An example of a rolling update strategy



Finally, the remaining 4Pods in the old set are deleted. The old ReplicaSet is retained for rollback even though it is now empty.

## An example of a rolling update strategy



This leaves 10 Pods in a new ReplicaSet.

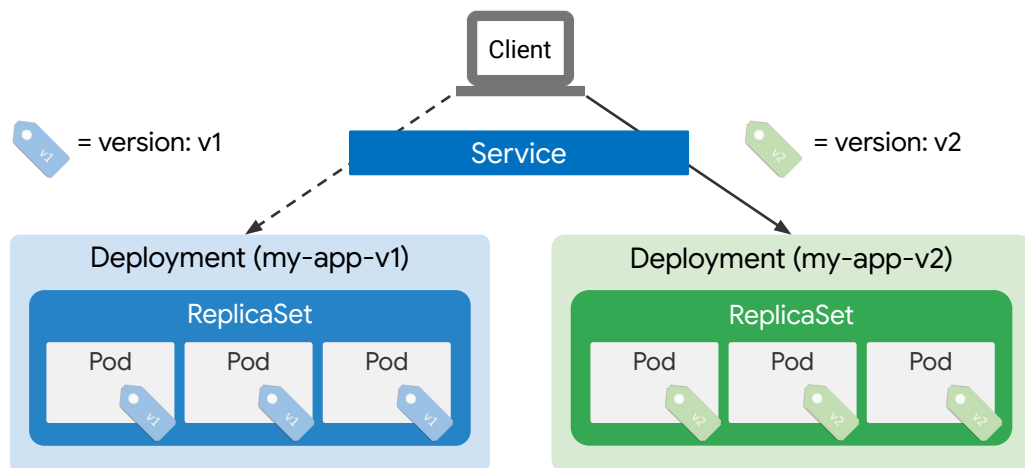
There are a few additional options, such as 'Min Ready Seconds' and 'Progress Deadline Seconds.' 'Min Ready Seconds' defines the number of seconds to wait before a Pod is considered Available, without crashing any of its containers.

The default for `minReadySeconds` is 0, meaning that as soon as the Pod is ready, it's made available.

Another option is 'progressDeadlineSeconds,' where you specify the wait period before a Deployment reports that it has failed to progress.



A blue/green deployment strategy ensures app services remain available



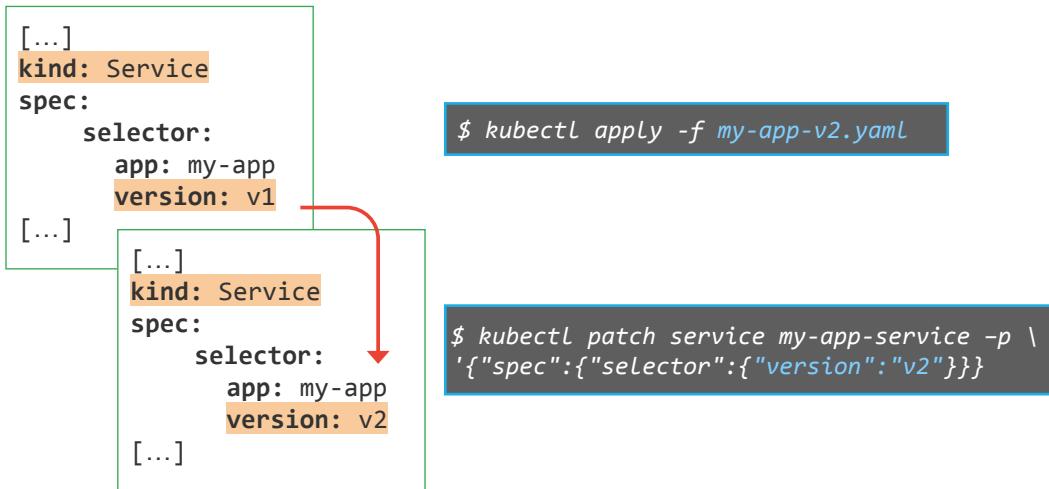
A blue/green deployment strategy is useful when you want to deploy a new version of an application and also ensure that application services remain available while the Deployment is updated.

With a blue/green update strategy, a completely new Deployment is created with a newer version of the application. In this case, it's my-app-v2.

When the Pods in the new Deployment are ready, the traffic can be switched from the old blue version to the new green version. But how can you do this?

This is where a Kubernetes Service is used. Services allow you to manage the network traffic flows to a selection of Pods. This set of Pods is selected using a label selector.

## Applying a blue/green deployment strategy



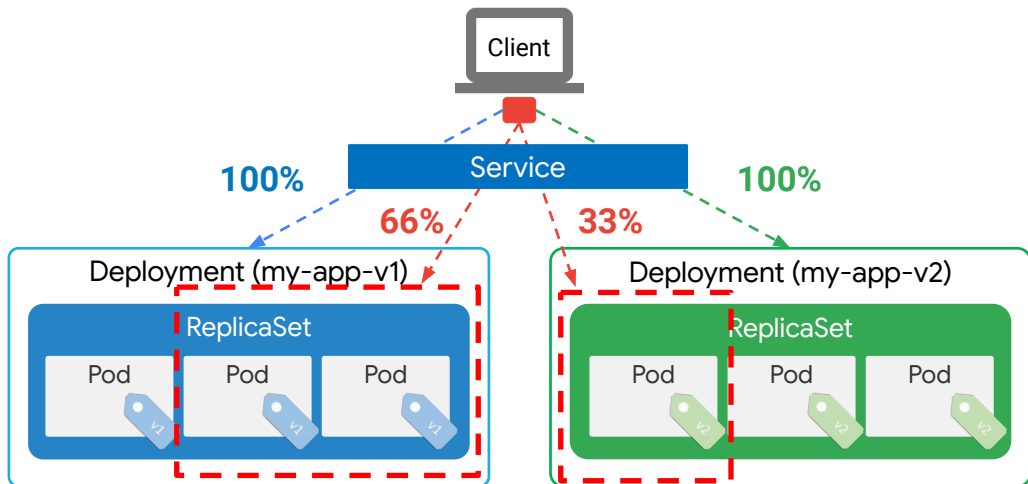
Here, in the Service definition, Pods are selected based on the label selector, where Pods in this example belong to my-app and to version v1.

When a new Deployment, labelled v2 in this case, is created and is ready, the version label in the Service is changed to the newer version, labeled v2 in this example. Now, the traffic will be directed to the newer set of Pods, the green deployment with the v2 version label, instead of to the old blue deployment Pods that have the v1 version label. The blue Deployment with the older version can then be deleted.

The advantage of this update strategy is that the rollouts can be instantaneous, and the newer versions can be tested internally before releasing them to the entire user base, for example by using a separate service definition for test user access.

The disadvantage is that resource usage is doubled during the Deployment process.

Canary deployment is an update strategy where traffic is gradually shifted to the new version



The canary method is another update strategy based on the blue/green method, but traffic is *gradually* shifted to the new version. The main advantages of using canary deployments are that you can minimize excess resource usage during the update, and because the rollout is gradual, issues can be identified before they affect all instances of the application.

In this example, 100% of the application traffic is directed initially to my-app-v1 .

When the canary deployment starts, a subset of the traffic, 33% in this case, or a single pod, is redirected to the new version, my-app-v2, while 66%, or two pods, from the older version, my-app-v1, remain running.

When the stability of the new version is confirmed, 100% of the traffic can be routed to this new version. How is this done?

## Applying a canary deployment

```
[...]
kind: Service
spec:
  selector:
    app: my-app
[...]
```

```
$ kubectl apply -f my-app-v2.yaml
```

```
$ kubectl scale deploy/my-app-v2 --replicas=10
```

```
$ kubectl delete -f my-app-v1.yaml
```



In the blue/green update strategy covered previously, both the app and version labels were selected by the Service, so traffic would only be sent to the Pods that are running the version defined in the Service.

In a Canary update strategy, the Service selector is based only on the application label and does not specify the version. The selector in this example covers all Pods with the app:my-app label. This means that with this Canary update strategy version of the Service, traffic is sent to all Pods, regardless of the version label.

This setting allows the Service to select and direct the traffic to the Pods from both Deployments. Initially, the new version of the Deployment will start with zero replicas running. Over time, as the new version is scaled up, the old version of the Deployment can be scaled down and eventually deleted.

With the canary update strategy, a subset of users will be directed to the new version. This allows you to monitor for errors and performance issues as these users use the new version, and you can roll back quickly, minimizing the impact on your overall user base, if any issues arise.

However, the complete rollout of a Deployment using the canary strategy can be a slow process and may require tools such as Istio to accurately shift the traffic. There are other deployment strategies, such as A/B testing and shadow testing. These strategies are outside the scope of this course.

## Applying a Recreate strategy

```
[...]
kind: deployment
spec:
  replicas: 10
  strategy:
    type: Recreate
[...]
```



‘Recreate’ is a strategy type where all the old Pods are deleted before new Pods are created. This clearly affects the availability of your application, because the new Pods must be created and will not all be available instantly. For example, what if the contract of communication between parts of your application is changing, and you need to make a clean break? In such situations a continuous deployment strategy doesn’t make sense. All the replicas need to change at once. That’s when the Recreate strategy is recommended.

## Rolling back a Deployment

```
$ kubectl rollout undo deployment [DEPLOYMENT_NAME]
```

```
$ kubectl rollout undo deployment [DEPLOYMENT_NAME] --to-revision=2
```

```
$ kubectl rollout history deployment [DEPLOYMENT_NAME] --revision=2
```

Clean up Policy:

- Default: 10 Revision
- Change: `.spec.revisionHistoryLimit`



So that's 'rollout.' Next we'll discuss how to roll back updates, especially in rolling update and recreate strategies.

You roll back using a `kubectl 'rollout undo'` command. A simple 'rollout undo' command will revert the Deployment to its previous revision.

You roll back to a specific version by specifying the revision number.

If you're not sure of the changes, you can inspect the rollout history using the `kubectl 'rollout history'` command.

The GCP Console doesn't have a direct rollback feature; however, you can start Cloud Shell from your Console and use these commands. The GCP Console also shows you the revision list with summaries and creation dates.



By default, the details of 10 previous ReplicaSets are retained, so that you can roll back to them. You can change this default by specifying a revision history limit under the Deployment specification.

## Deployment has three different lifecycle states



Any Deployment has three different lifecycle states.

The Deployment's *Progressing* state indicates that a task is being performed. What tasks? Creating a new ReplicaSet... or scaling up or scaling down a ReplicaSet.

The Deployment's *Complete* state indicates that all new replicas have been updated to the latest version and are available, and no old replicas are running.

Finally, the *Failed* state occurs when the creation of a new ReplicaSet could not be completed. Why might that happen? Maybe Kubernetes couldn't pull images for the new Pods. Or maybe there wasn't enough of some resource quota to complete the operation. Or maybe the user who launched the operation lacks permissions.

When you apply many small fixes across many rollouts, that translates to a large number of revisions, and to management complexity. You have to remember which small fix was applied with which rollout, which can make it challenging to figure out which revision to roll back to when issues arise. Remember, earlier in this specialization, we recommended that you keep your YAML files in a source code repository? That will help you manage some of this complexity.

## Different actions can be applied to a Deployment

Pause

```
$ kubectl rollout pause deployment [DEPLOYMENT_NAME]
```

Resume

```
$ kubectl rollout resume deployment [DEPLOYMENT_NAME]
```

Monitor

```
$ kubectl rollout status deployment [DEPLOYMENT_NAME]
```



When you edit a deployment, your action normally triggers an automatic rollout. But if you have an environment where small fixes are released frequently, you'll have a large number of rollouts. In a situation like that, you'll find it more difficult to link issues with specific rollouts. To help, you can temporarily pause this rollouts by using the `kubectl rollout pause` command. The initial state of the Deployment prior to pausing it will continue its function, but new updates to the Deployment will not have any effect while the rollout is paused. The changes will only be implemented once the rollout is resumed.

When you resume the rollout, all these new changes will be rolled out with a single revision.

You can also monitor the rollout status by using the `kubectl 'rollout status'` command.

## Delete a Deployment

```
$ kubectl delete deployment [DEPLOYMENT_NAME]
```

 REFRESH

 EDIT

 DELETE

 ACTIONS ▼

 KUBECTL ▼

### Delete

Delete a resource

Are you sure you want to delete nginx-1? It will delete all resources managed by it.

The operation cannot be reverted.

☒ Delete horizontal pod autoscaler nginx-1

[CANCEL](#) [DELETE](#)



What if you're done with a Deployment? You can delete it easily by using the kubectl 'delete' command, and you can also delete it from the GCP Console. Either way, Kubernetes will delete all resources managed by the Deployment, especially running Pods.

# Lab

---

## Creating Google Kubernetes Engine Deployments



In this lab, you'll explore the basics of using deployment manifests.

The first task that you'll learn to perform is to create a deployment manifest for a Pod inside the cluster. You'll then use both the GCP Console and Cloud Shell to manually scale Pods up and down. The next task will be to trigger a deployment rollout and a deployment rollback. Various types of service types (ClusterIP, NodePort, LoadBalancer) can be used with deployments to manage connectivity and availability during updates. You'll perform a task where you define service types in the manifest and verify LoadBalancer creation. In your final task, you'll create a new canary deployment for the release of your application.

# Agenda

---

Deployments

Jobs and CronJobs

**Cluster Scaling**

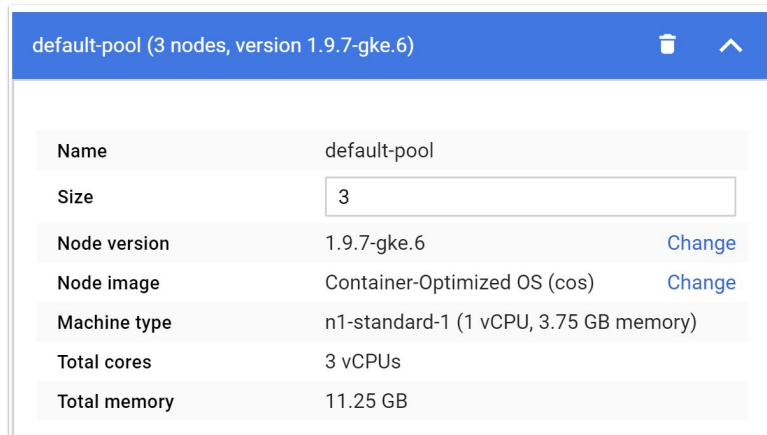
Controlling Pod Placement

Getting Software into Your Cluster



In this lesson, we look at how you manage cluster scaling in GKE. I will review the concept of node pools. You'll learn how you can modify the capacity of your entire cluster... either by manually changing the number of nodes in node pools, or by configuring additional node pools. You will then learn how the GKE cluster autoscaler works, and how you can configure it to manage the size of your cluster automatically.

## Scaling a cluster from the GCP Console



default-pool (3 nodes, version 1.9.7-gke.6)	
Name	default-pool
Size	<input type="text" value="3"/>
Node version	1.9.7-gke.6 <a href="#">Change</a>
Node image	Container-Optimized OS (cos) <a href="#">Change</a>
Machine type	n1-standard-1 (1 vCPU, 3.75 GB memory)
Total cores	3 vCPUs
Total memory	11.25 GB



The level of resources your applications need will vary over time. If you need to change the amount of resources available in your Kubernetes Engine clusters, you can increase or decrease the number of Google Kubernetes Engine nodes in your cluster directly from the GCP Console. New nodes created during the process automatically self-register to the cluster within the GCP environment. A node pool is a subset of node instances within a cluster that all have the same configuration.

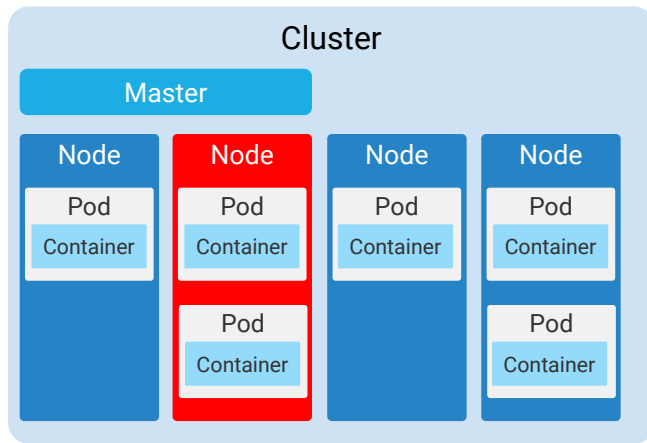
A node pool is a subset of node instances within a cluster that all have the same configuration. Node pools use a NodeConfig specification. Each node in the pool has a Kubernetes node label which has the node pool's name as its value. When you create a container cluster, the number and type of nodes that you specify becomes the default node pool. Then, you can add additional custom node pools of different sizes and types to your cluster. All



nodes in any given node pool are identical to one another.

In the GCP Console, you can manually increase the size of the cluster by increasing the size of the node pools in the cluster. The Node Pool size represents the number of nodes in the node pool per zone. For example, if this particular pool spans two compute zones, and you increase the node size from 3 to 6, each zone will have 6 nodes registered, and the total number of nodes in this pool will be 12. Existing Pods are not moved to the newer nodes when the cluster size is increased.

## Manual Cluster Scale down selects nodes randomly



You can also manually decrease the cluster size. When you reduce the size of a cluster, the nodes to be removed are selected randomly. The resize process doesn't differentiate between nodes that are running Pods and ones that are empty.

When you remove a node from the cluster, all the Pods within that node will be terminated gracefully. Graceful termination means that a TERM signal is first sent to the main process in each container. A grace period is then allowed before a KILL signal is sent and the Pod is deleted. This grace period is defined for each Pod.

If these Pods are managed by a replication controller such as a ReplicaSet or StatefulSet, they'll be rescheduled on the remaining nodes. Otherwise, the Pods won't be restarted elsewhere.

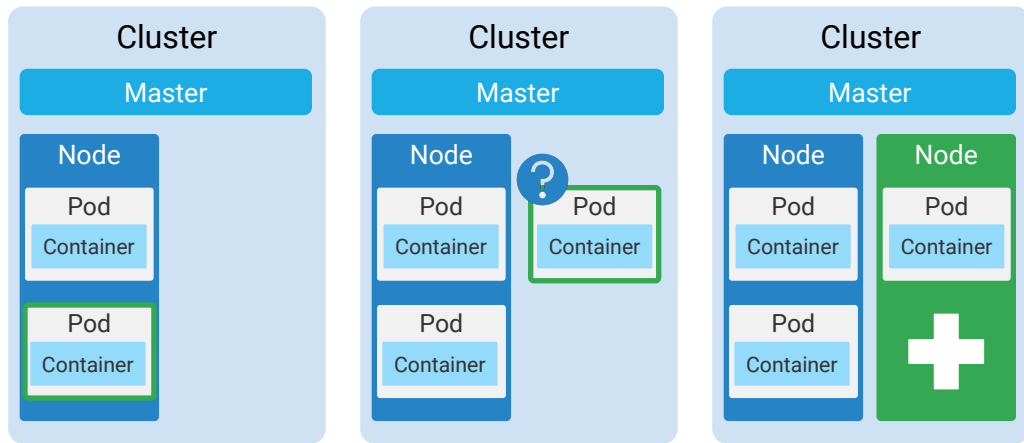
## Scaling a cluster using the gcloud command

```
gcloud container clusters resize  
projectdemo --node-pool default-pool \  
--size 6
```



You can also resize a cluster manually from the command line using the 'resize' gcloud command. As with the GCP Console, if you reduce the size of the cluster, then nodes that are running Pods and nodes without Pods aren't differentiated. Resize will pick instances to remove at random, and any running Pods will be terminated gracefully. If your Pods aren't managed by a replication controller, they won't be restarted.

## Scale up a cluster with autoscaling



Cluster autoscaler controls the number of worker nodes in response to workload demands. GKE's cluster autoscaler can automatically resize a cluster based on the resource demands of your workload. By default, the cluster autoscaler is disabled. Cluster autoscaling allows you to pay only for resources that are needed at any given moment and to automatically get additional resources when demand increases. When autoscaling is enabled, GKE automatically adds a new node to your cluster if you've created new Pods that don't have enough capacity to run. If a node in your cluster is underutilized and its Pods can be run on other nodes, GKE can delete the node. Keep in mind that when nodes are deleted, your applications can experience some disruption. Before enabling autoscaling, you should make sure that your services can tolerate the potential disruption.

Pods have their own CPU and memory resource requirements,

based on the resource requests and limits of their containers. When it schedules a Pod, the Kubernetes scheduler must allocate that Pod to a node that can meet the demands of all of the Pod's containers.

If there isn't enough resource capacity across any of the node pools, the Pod will have to wait until either other Pods terminate and free up capacity or additional nodes are added. When the Pod has to wait for resource capacity, the scheduler marks the Pod as unschedulable by setting its 'schedulable' Pod Condition to false with the reason – Unschedulable.

If you enabled autoscaling, the GKE autoscaler checks whether a scale-up action will help the situation as soon as it detects that any Pod is considered unschedulable. If so, it adds a new node to the node pool where the Pod is waiting for resources to become available, and the Pod is then scheduled on that node. However, this requires a new VM instance to be deployed, which will need to start up and initialize before it can be used to schedule Pods. Also note that while the autoscaler ensures that all nodes in a single node pool have the same set of labels applied, labels that have been manually added after initial cluster or node pool creation are not automatically carried over to the new nodes when the cluster autoscales.

## Scale down a cluster with autoscaling

1 There can be no scale-up events pending.




2 Can the node be deleted safely?



Deploying a Pod to an existing running node may only take a few seconds, but it might take minutes before the new node added by the autoscaler can be used. And adding nodes means spending money, too. So you should think of cluster scaling as a coarse-grained operation that should happen infrequently, and Pod scaling with Deployments as a fine-grained operation that should happen frequently. You can use both kinds of scaling together to balance your performance and your spending. The GKE cluster autoscaler can also scale down nodes. Let's look at how the autoscaler manages scale-down.

First, the cluster autoscaler ensures that there's no scale-up event pending. If a scale-up event happens during the scale-down process, the scale-down is not executed. Second, it checks that the node can be deleted safely.

## Pod conditions that prevent node deletion

-  Not run by a controller
-  Has local storage
-  Restricted by constraint rules






If the node contains Pods that meet any of the following conditions, then the node cannot be deleted ...

Pods that are not managed by a controller. These are any Pods that are not in a Deployment, ReplicaSet, Job, StatefulSet, etc.  
Pods that have local storage.

Pods that are restricted by constraint rules that prevent them from running on any other node, which will be explained in detail later in this module.

## Pod conditions that prevent node deletion

-  `cluster-autoscaler.kubernetes.io/safe-to-evict` is set to False
-  Restrictive `PodDisruptionBudget`
-  `kubernetes.io/scale-down-disabled` set to True



There are a number of non-default settings that can be explicitly set to prevent node deletion.

Pods that have the `safe-to-evict` annotation set to 'False.' The `safe-to-evict` annotation provides a direct setting at the Pod level that tells the autoscaler that the Pod cannot be evicted. As a result, the node that it's running on won't be selected for deletion when the cluster is scaled down.

Pods that have a restricted `PodDisruptionBudget` can also prevent a node from being deleted. You can define `PodDisruptionBudget` to specify the number of controller replicas that must be available at any given time. For example, in a Deployment with 3 replicas, and `PodDisruptionBudget` set to 2, only one replica can be evicted or disrupted at a time.

At the node level, if the node's `scale-down-disabled` annotation is set to 'True,' that node will always be excluded from scale-down actions.