Deployments, Jobs, and Scaling

Welcome to the Deployment, Jobs, and Scaling module. GKE works with containerized applications: applications packaged into hardware-independent, isolated user-space instances. In GKE and Kubernetes, these containers, whether for applications or batch jobs, are collectively called *workloads*. In this module you will learn about Deployments and Jobs, two of the main types of workload. You will also learn about the mechanisms that are used to scale the GKE clusters where you run your applications.
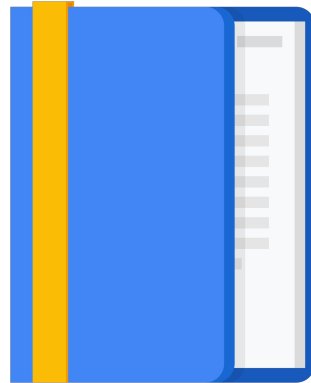
Learn how to ...

Create and use Deployments.

Create and run Jobs and CronJobs.

Scale clusters manually and automatically.

Configure Node and Pod affinity.

Get software into your cluster.

Google Cloud

In this module, you'll learn how to create and use Deployments, including scaling them, create and run Jobs and CronJobs, scale GKE clusters manually and automatically, control on which Nodes Pods may and may not run, and explore ways to get software into your cluster.

Agenda

Google Cloud

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.

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 is a two-part process

.yaml file

Deployment object

Deployment controller

Node

Pod | Pod | Pod
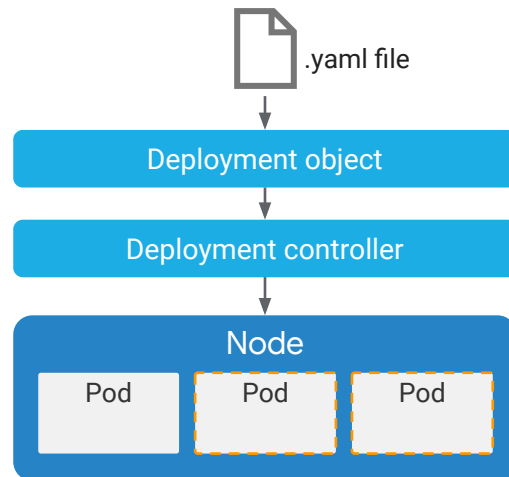
The desired state is described in a Deployment YAML file containing the characteristics of the pods, coupled with how to operationally run these pods and handle their lifecycle events. After you submit this file to the Kubernetes control plane, it creates a deployment controller, which is responsible 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.

Deployment object file in YAML format

```yaml
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
```

Google Cloud

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.

Deployment has three different lifecycle states

Progressing
State

Complete
State

Failed
State

Google Cloud

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 course series, we recommended that you keep your YAML files in a source code repository? That will help you manage some of this complexity.

There are three ways to create a Deployment

1
```
$ kubectl apply -f [DEPLOYMENT_FILE]
```

2
```
$ kubectl create deployment \
[DEPLOYMENT_NAME] \
  --image [IMAGE]:[TAG] \
  --replicas 3 \
  --labels [KEY]=[VALUE] \
  --port 8080 \
  --generator deployment/apps.v1 \
  --save-config
```

Google Cloud

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 create deployment 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.

Your third option is to use the GKE Workloads menu in the Cloud 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
```

Google Cloud

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
```

Google Cloud

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 Cloud Console

nginx-1

Overview | Details | Revision history | Events | YAML

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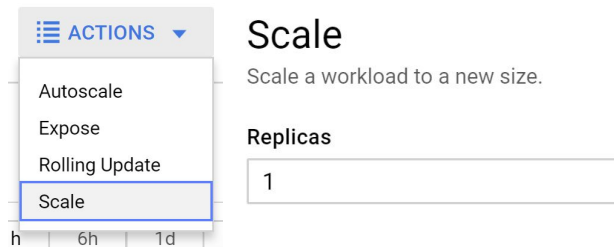
Revision	Name ^	Status	Restarts	Created on
1	nginx-1-7cb5b65464-2j97f	✓ Running	0	Oct 12, 2018, 11:08:29 AM
1	nginx-1-7cb5b65464-gct8h	✓ Running	0	Oct 12, 2018, 11:08:29 AM
1	nginx-1-7cb5b65464-tz85f	✓ Running	0	Oct 12, 2018, 11:08:29 AM

Google Cloud

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

You can scale the Deployment manually

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

ACTIONS ▼

Autoscale
Expose
Rolling Update
Scale

h | 6h | 1d

Scale

Scale a workload to a new size.

Replicas

1

Google Cloud

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 Cloud 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)

```
1
```

Maximum number of Pods

```
3
```

Target CPU utilization in percent (Optional)

```
80
```

CANCEL   DISABLE AUTOSCALER   AUTOSCALE

Google Cloud

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 Cloud 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.

Thrashing is a problem with any type of autoscaling

Cooldown/delay support:

```
--horizontal-pod-autoscaler-downscale-stabilization
```

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]
```

Google Cloud

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

C REFRESH   ✏ EDIT   🗑 DELETE   ☰ ACTIONS ▾   ⌨ KUBECTL ▾

Rolling update
Update workload Pods to a new application version.

Minimum seconds ready ❓ (Optional)

```
0
```

Maximum surge ❓ (Optional)

```
30%
```

Maximum unavailable ❓ (Optional)

```
1
```

Container name	Image
nginx	nginx:latest

Google Cloud

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

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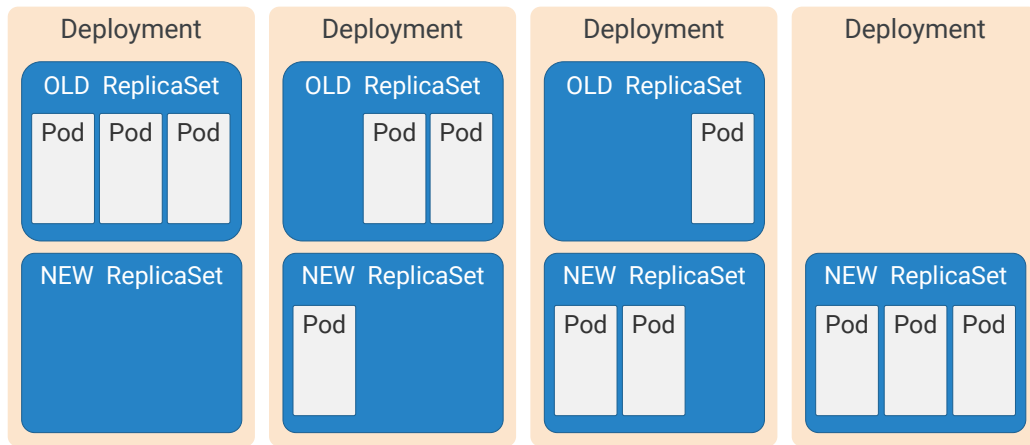
When a Deployment is updated, it launches a new ReplicaSet and creates a new set of Pods in a controlled fashion.

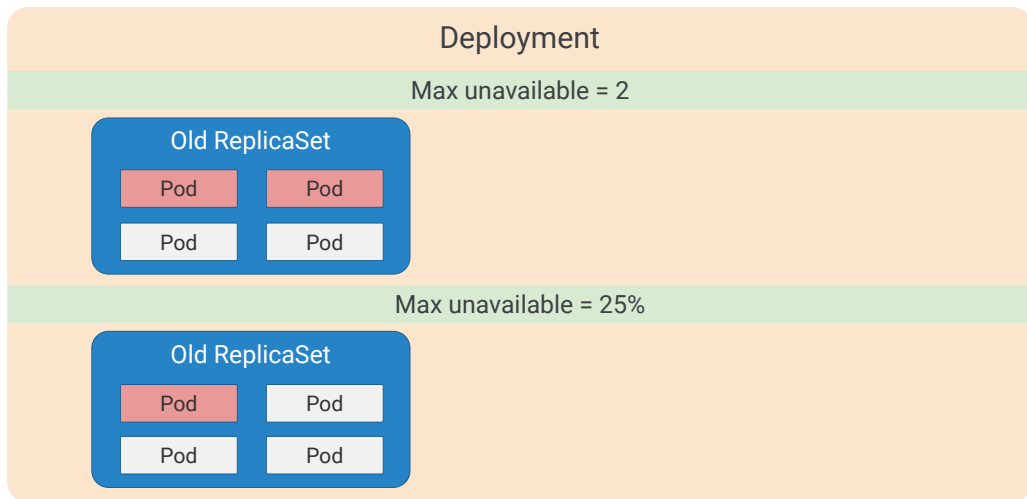First, new Pods are launched in a new ReplicaSet.

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

Deployment

Max unavailable = 2

Old ReplicaSet

| Pod | Pod |
| Pod | Pod |

Max unavailable = 25%

Old ReplicaSet

| Pod | Pod |
| Pod | Pod |

Google Cloud

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

Deployment

Max surge = 2

Old ReplicaSet

Pod	Pod
Pod	Pod

New ReplicaSet

Pod	Pod

Max surge = 25%

Old ReplicaSet

Pod	Pod
Pod	Pod

New ReplicaSet

Pod

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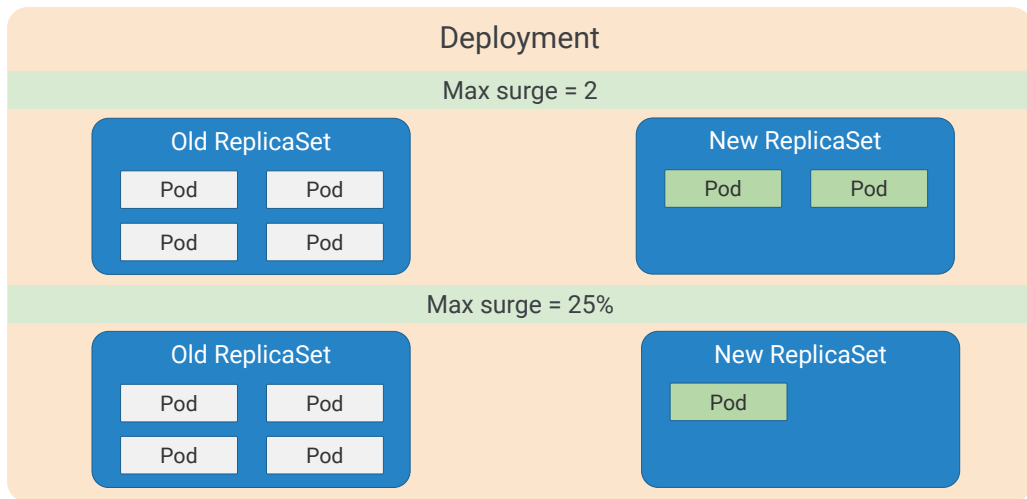
Max surge allows you to specify the maximum number of extra Pods that can be created concurrently in a new ReplicaSet. For example, you can specify that you want to add up to two new Pods at a time in a new ReplicaSet, and the Deployment controller will do exactly that.

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: 10%
[…]
```

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

Deployment
Desired Pods = 10 Pods Max unavailable = 10% of desired Pods Max surge = 5 Pods

Total Pods = 10

Old ReplicaSet

Number of Pods = 10

The Old ReplicaSet has 10 Pods.

An example of a rolling update strategy

Deployment

Desired Pods = 10 Pods
Max unavailable = 10% of desired Pods
Max surge = 5 Pods

Total Pods = 10

Old ReplicaSet	New ReplicaSet
Number of Pods = 10	Number of Pods = 5
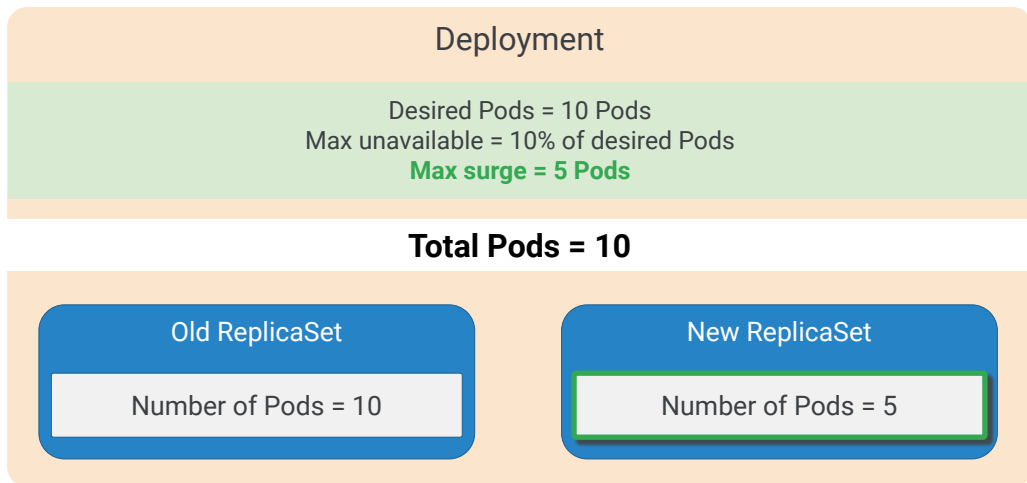
The Deployment will begin by creating 5 new Pods in a new ReplicaSet based on max surge.

An example of a rolling update strategy

Deployment

Desired Pods = 10 Pods
Max unavailable = 10% of desired Pods
Max surge = 5 Pods

Total Pods = 15 (Max)

Old ReplicaSet	New ReplicaSet
Number of Pods = 10	Number of Pods = 5

Google Cloud

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

An example of a rolling update strategy

Deployment

Desired Pods = 10 Pods
**Max unavailable = 10% of desired Pods**
Max surge = 5 Pods
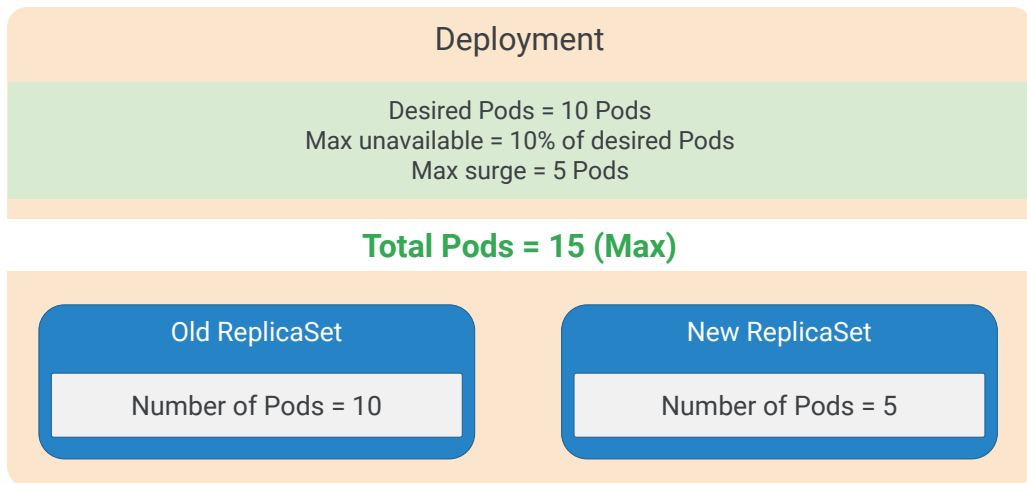
Total Pods = 15 (Max)

Old ReplicaSet	New ReplicaSet
Number of Pods = 10 - 6 = 4	Number of Pods = 5
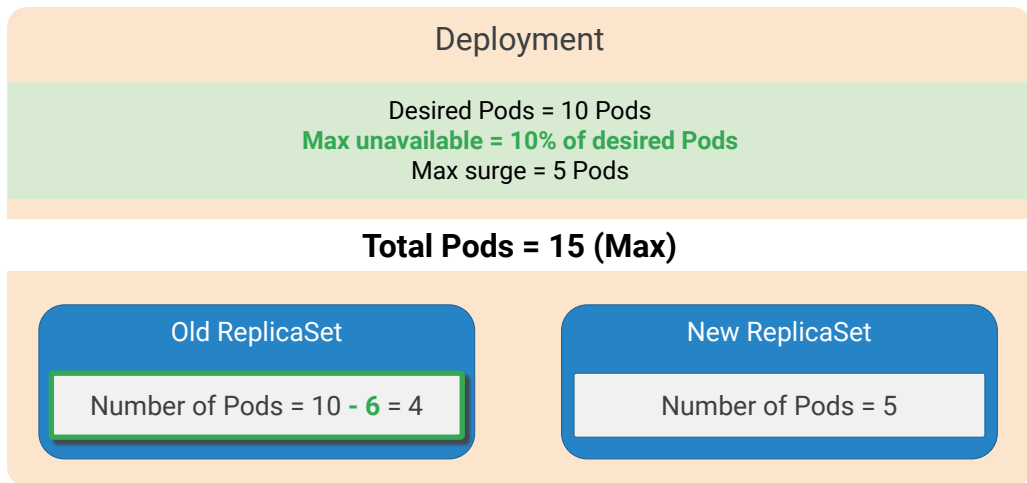
With max unavailable set to 10%, the minimum number of Pods running, regardless of old or new ReplicaSet, is 10 minus 10%, which equates to 9 Pods. So the rollout can remove old Pods until the overall total is no lower than 9 Pods. This means that 6 Pods can be removed from the old ReplicaSet at this stage.

An example of a rolling update strategy

Deployment

Desired Pods = 10 Pods
Max unavailable = 10% of desired Pods
Max surge = 5 Pods

Total Pods = 9 (Min)

Old ReplicaSet	New ReplicaSet
Number of Pods = 10 - 6 = **4**	Number of Pods = **5**

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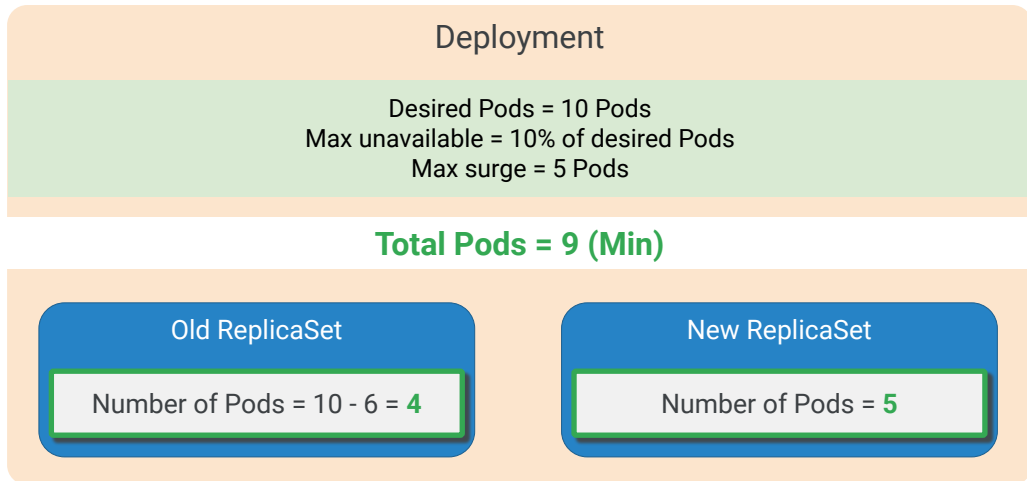
This keeps the minimum total at 9 Pods: 5 in the new ReplicaSet, and 4 in the old ReplicaSet.

An example of a rolling update strategy

Deployment

Desired Pods = 10 Pods
Max unavailable = 10% of desired Pods
Max surge = 5 Pods

Total Pods = 9 (Min)

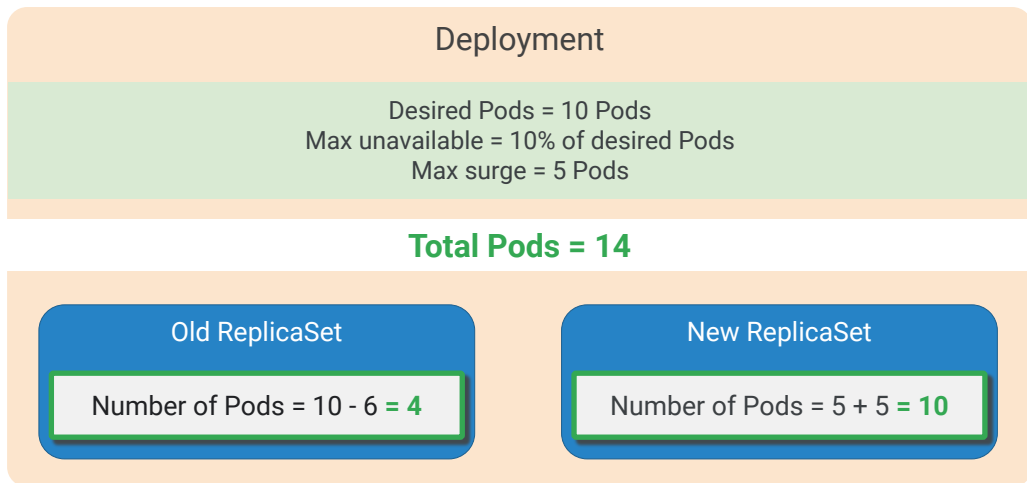Old ReplicaSet

Number of Pods = 10 - 6 = 4

New ReplicaSet

Number of Pods = 5 + 5 = 10

Google Cloud

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

An example of a rolling update strategy

Deployment

Desired Pods = 10 Pods
Max unavailable = 10% of desired Pods
Max surge = 5 Pods

Total Pods = 14

Old ReplicaSet	New ReplicaSet
Number of Pods = 10 - 6 **= 4**	Number of Pods = 5 + 5 **= 10**

Google Cloud

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

Deployment

Desired Pods = 10 Pods
Max unavailable = 10% of desired Pods
Max surge = 5 Pods

Total Pods = 10

Old ReplicaSet

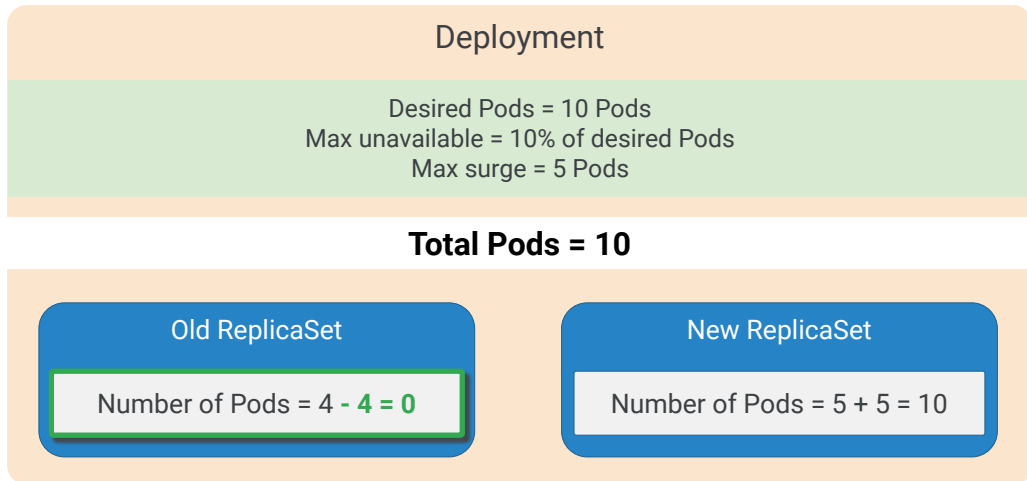Number of Pods = 4 - 4 = 0

New ReplicaSet

Number of Pods = 5 + 5 = 10

Google Cloud

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

Deployment

Desired Pods = 10 Pods
Max unavailable = 10% of desired Pods
Max surge = 5 Pods

Total Pods = 10

New ReplicaSet

Number of Pods = 5 + 5 = 10
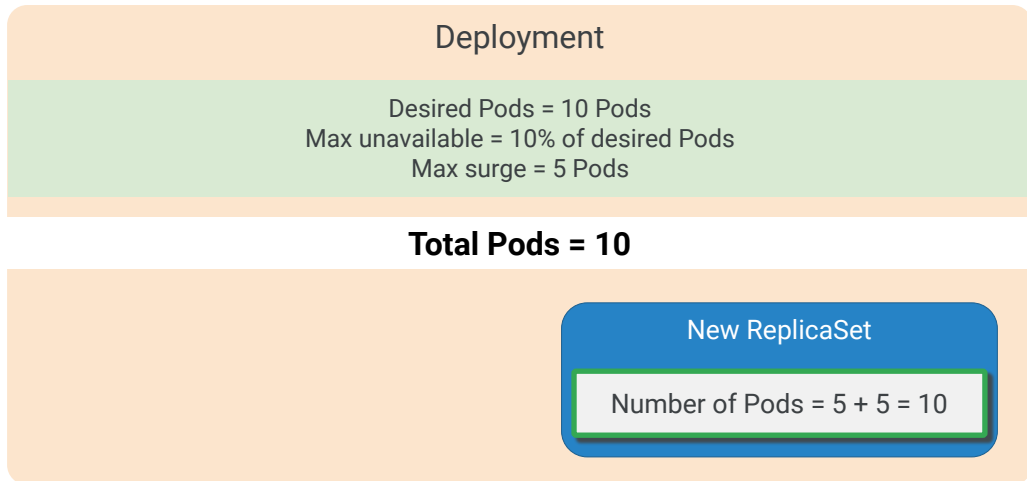
This leaves 10 Pods in a new ReplicaSet.

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.

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.

Service is a stable network representation of a set of pods
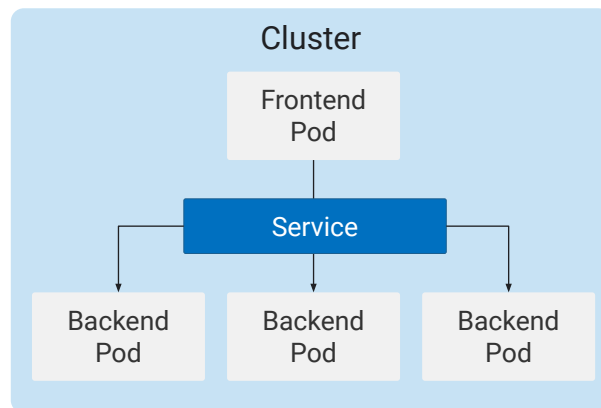
We haven't yet discussed how to locate and connect to the applications running in these Pods, especially as new Pods are created or updated by your Deployments. While you can connect to individual Pods directly, Pods themselves are transient. A Kubernetes Service is a static IP address that represents 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.

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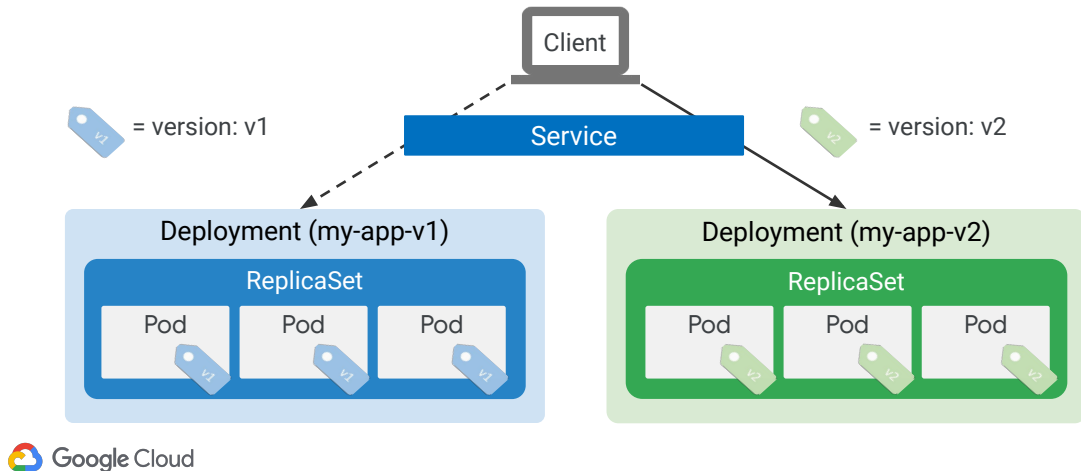
A blue/green deployment strategy is useful when you want to deploy a new version of an application and also ensure that application services remain available while the Deployment is updated.

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

```
[...]
kind: Service
spec:
    selector:
      app: my-app
      version: v1
[...]
```

```
[...]
kind: Service
spec:
    selector:
      app: my-app
      version: v2
[...]
```

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

```
$ kubectl patch service my-app-service -p \
'{"spec":{"selector":{"version":"v2"}}}
```

Google Cloud

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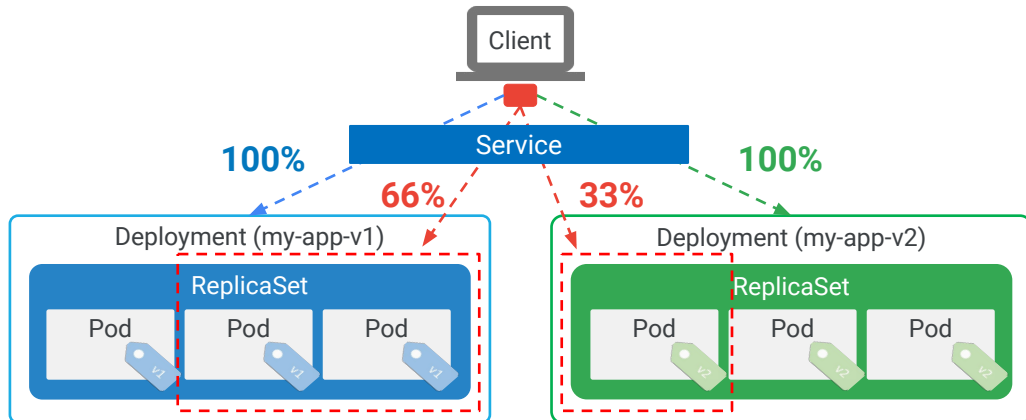
The canary method is another update strategy based on the blue/green method, but traffic is *gradually* shifted to the new version. The main advantages of using canary deployments are that you can minimize excess resource usage during the update, and because the rollout is gradual, issues can be identified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

Session affinity ensures that all client requests are sent to the same Pod
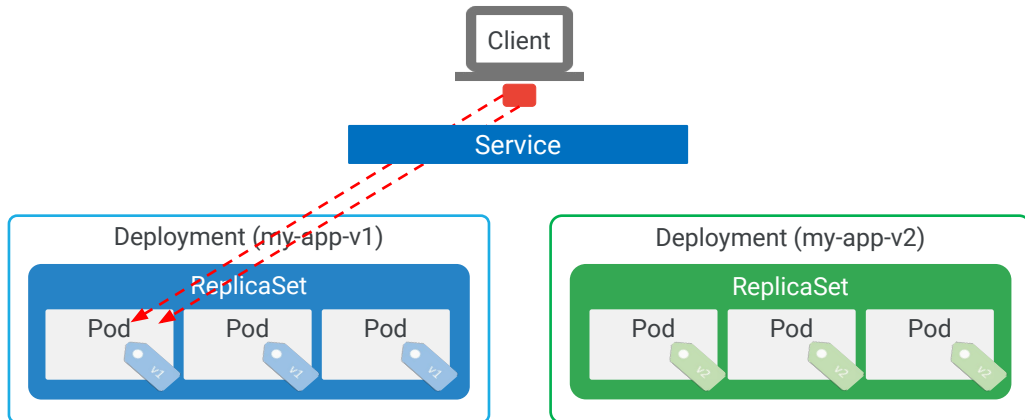
A Service configuration does not normally ensure that all requests from a single client will always connect to the same Pod. Each request is treated separately and can connect to any Pod deployment. This potential can cause issues if there are significant changes in functionality between Pods as may happen with a canary deployment. To prevent this you can set the sessionAffinity field to ClientIP in the specification of the service if you need a client's first request to determine which Pod will be used for all subsequent connections.

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

Google Cloud

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 Cloud Console doesn't have a direct rollback feature; however, you can start Cloud Shell from your Console and use these commands. The Cloud 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.

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 rollout 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]
```

C 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

Google Cloud

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 Cloud Console. Either way, Kubernetes will delete all resources managed by the Deployment, especially running Pods.

Agenda

Deployments

Jobs and CronJobs

Lab: Deploying Jobs on Google Kubernetes Engine

Cluster Scaling

Controlling Pod Placement

Getting Software into Your Cluster
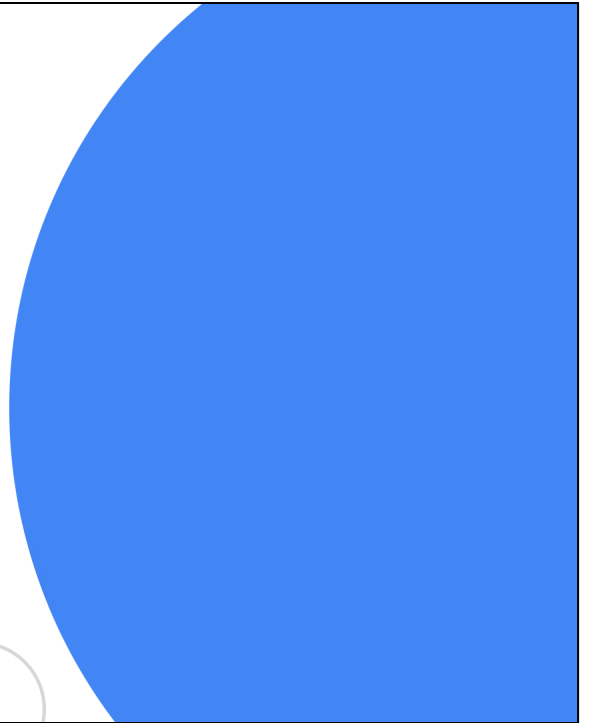
Lab: Configuring Pod Autoscaling and Node Pools

Summary

Google Cloud

Lab Intro
Creating Google Kubernetes
Engine Deployments

Google Cloud

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 Cloud 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

Google Cloud

In this lesson, we'll explore Jobs and CronJobs in more detail. Let's start with Jobs.

A Job is a Kubernetes object like a Deployment. A Job creates one or more Pods to run a specific task reliably. In its simplest form, a Job will create one Pod and track the task completion within that Pod. When the task is completed, it will terminate the Pod and report that the Job has successfully finished.

A scenario where Job provides the solution
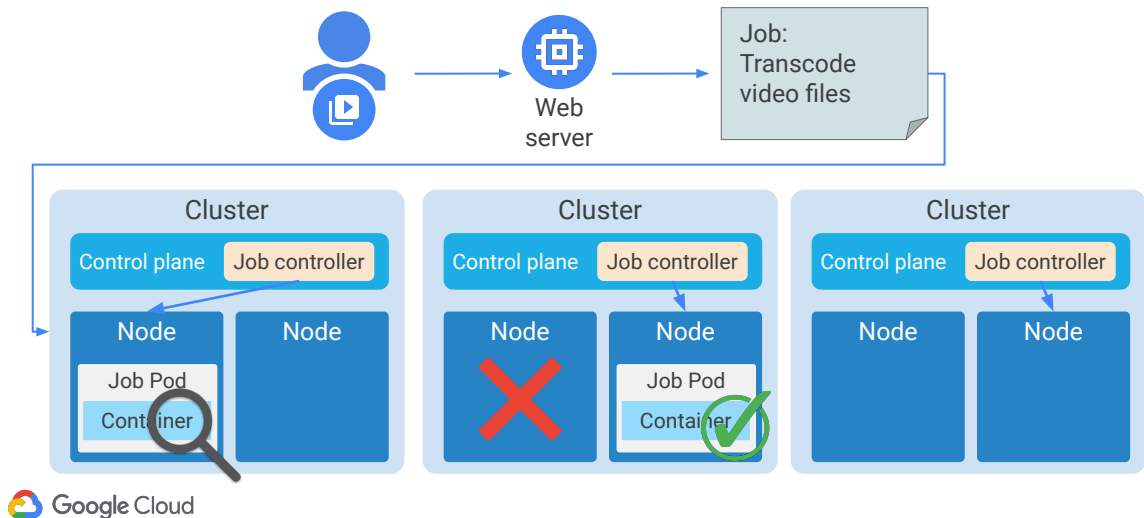
Consider a scenario where you're transcoding some video files using a single, standalone Pod. What if the node that Pod is running on suddenly shuts down? What will happen to the task that was being performed? It'll be lost. When the Pod is terminated for any reason, it won't be restarted. Jobs provide a mechanism to handle this type of failure. Unlike other Kubernetes controllers, Jobs manage a task up to its completion, rather than to an open-ended desired state. In a way, the desired state of a job is its completion. Kubernetes will make sure it reaches that state successfully.

- In this scenario, the first step in the process involves a user uploading a video file to a web server for conversion, or transcoding.
- The web server creates a Job object manifest for this transcoding task.
- A Job is created on the cluster.
- The Job controller schedules a Pod for the Job on a node.
- The Job controller monitors the Pod.
- If a node failure occurs and the Pod is lost, the Job Controller is aware that the task has not completed.
- The Job controller reschedules the Job Pod to run on a different node.
- The Job controller continues to monitor the Pod until the task completes.
- When the task has completed, the Job controller removes the finished Job and any Pods associated with it.

Jobs can be defined as non-parallel or parallel

```
apiVersion: batch/v1
kind: Job
metadata:
  name: my-app-job
spec:
  completions: 3
  template:
    spec:
[...]
```

```
apiVersion: batch/v1
kind: Job
metadata:
  name: my-app-job
spec:
  completions: 3
  parallelism: 2
  template:
    spec:
[...]
```

Google Cloud

There are two main ways to define a Job: non-parallel and parallel. Non-parallel Jobs create only one Pod at a time. Of course, that Pod is recreated if it terminates unsuccessfully. These Jobs are completed when the Pod terminates successfully, or—if a completion count is defined—when the required number of completions is reached.

Parallel Jobs are Jobs that have a parallelism value defined, where multiple Pods are scheduled to work on the Job at the same time. If they also have a completion count defined, they are used for tasks that must be completed more than once. Kubernetes considers Parallel Jobs complete when the number of Pods that have terminated successfully reaches the completion count.

A second type of parallel Job for processing work queues can also be defined, which you will see later. Let's start by looking at how to create a simple non-parallel Job.

A Job computing π to 2000 places

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

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

```
$ kubectl run pi --image perl --restart Never -- perl -Mbignum=bpi -wle 'print bpi(2000)'
```

Google Cloud

Here's a simple example of a Job which computes pi to 2000 decimal places.

A Job object is specified through its Kind.

Within a Job spec, there's a Pod template. This is where a Pod's specifications and its restartPolicy are defined.

In this example, the restartPolicy is set to Never. This means that if a container in a Pod fails for any reason, the entire Pod fails, and the Job controller will respond to this Pod failure by launching a new Pod. The other restartPolicy option that can be used for Jobs is to set the restartPolicy to OnFailure. In this case, the Pod remains on the node, but the Container is re-started.

Where application failures are possible or expected, the backOffLimit field can be used. backOffLimit specifies the number of retries before a Job is considered to have failed entirely. The default is 6. This allows you to halt Jobs that would otherwise get stuck in a restart loop. Failed Pods are recreated with an exponentially increasing delay—10 seconds, 20 seconds, 40 seconds, and so on—up to a maximum of 6 minutes. In this example, if the Pods continue to fail 4 times, the Job will fail, with BackoffLimitExceeded given as the reason. If the Pods succeed before the backOffLimit is reached, the counter is reset.

Like other Kubernetes objects, the Job object can be created using a kubectl 'apply' command.

Alternatively, a Job can be created using the kubectl 'run' command.

The role of a non-parallel Job
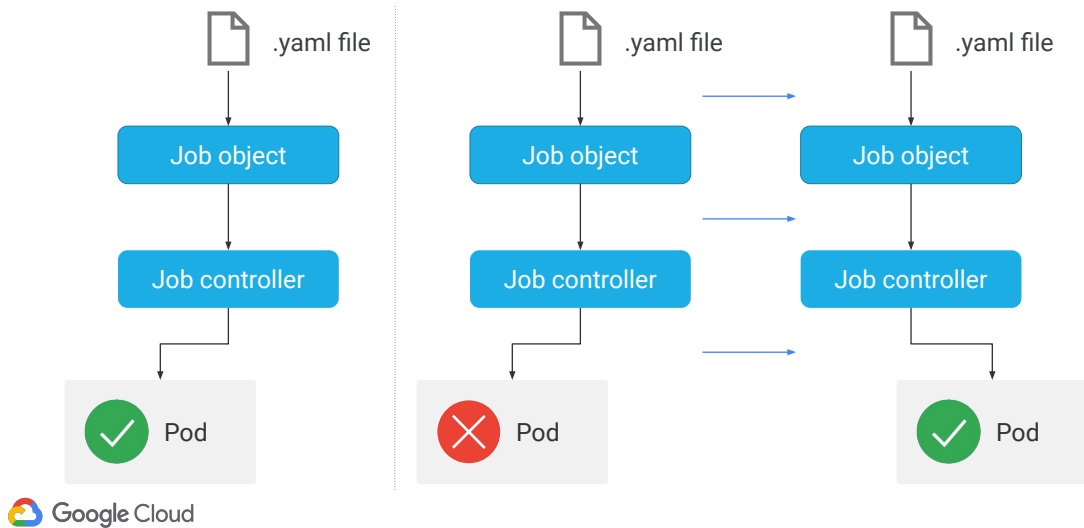
A non-parallel Job is where a single Pod is created to start the Job or task. The Pod is created as usual and the Job or task finishes successfully.

If the Pod fails for any reason it's restarted ensuring that the Job gets another opportunity to finish successfully.

Parallel Job with fixed completion count

```
apiVersion: batch/v1
kind: Job
metadata:
  name: my-app-job
spec:
  completions: 3
  parallelism: 2
  template:
    spec:
[...]
```

.yaml file

Job object

Job controller

Pod1

Pod2

Google Cloud
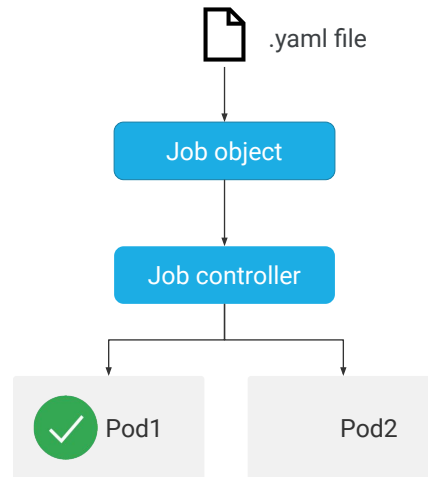
Now let's discuss parallel Jobs. The Parallel Job type creates multiple Pods that work on the same task at the same time. Parallel Job types are specified by setting the spec.parallelism value for a Job greater than 1. There are two types of parallel Jobs: one with a fixed task completion count, and the other which processes a work queue.

If you want to launch multiple Pods at the same time for parallel Jobs along with a fixed completion count, you can add completions along with parallelism. The Job controller will only launch the maximum number of Pods at the same time specified by the parallelism value and will continue restarting Pods until the completions count is reached.

In this example, the controller will launch up to two Pods in parallel to process tasks until three Pods have terminated successfully and the controller will wait for one of the running Pods to complete successfully,

Parallel Job with fixed completion count

```
apiVersion: batch/v1
kind: Job
metadata:
  name: my-app-job
spec:
  completions: 3
  parallelism: 2
  template:
    spec:
[...]
```

.yaml file

Job object

Job controller

Pod3

Pod2

before launching the next Pod.

Parallel Job with fixed completion count

```
apiVersion: batch/v1
kind: Job
metadata:
  name: my-app-job
spec:
  completions: 3
  parallelism: 2
  template:
    spec:
[...]
```
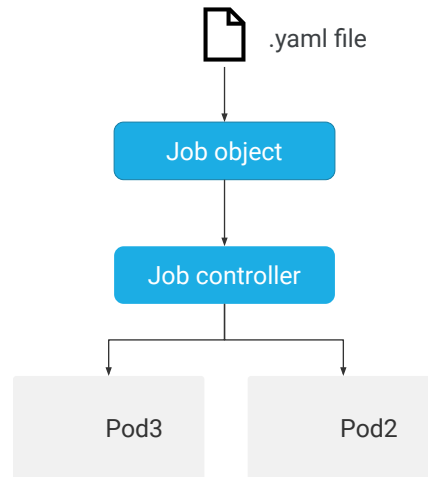

.yaml file

Job object

Job controller

Pod3

Pod2

Google Cloud
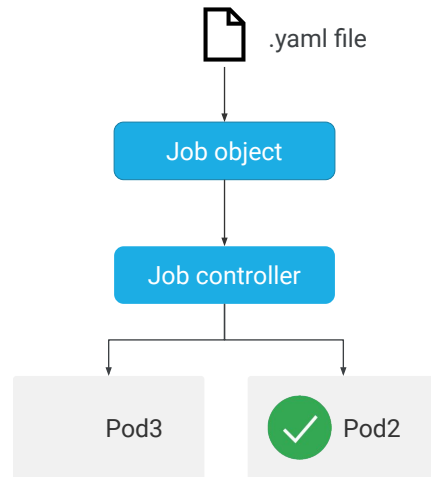
The Job controller will track successful completions.

Parallel Job with fixed completion count

```
apiVersion: batch/v1
kind: Job
metadata:
  name: my-app-job
spec:
  completions: 3
  parallelism: 2
  template:
    spec:
[...]
```

.yaml file

Job object

Job controller

Pod3

Job finished successfully

Google Cloud
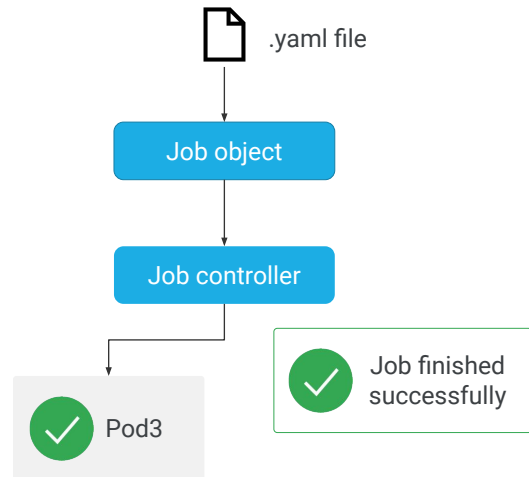
When the specified number of completions has been reached, the Job is considered complete. If the remaining number of completions is less than the parallelism value, the controller will not schedule new Pods as there are sufficient remaining Pods running at that point to complete the desired total for the Job.

Parallel Job with a worker queue

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: my-app-job
spec:
  parallelism: 3
  template:
    spec:
[...]
```
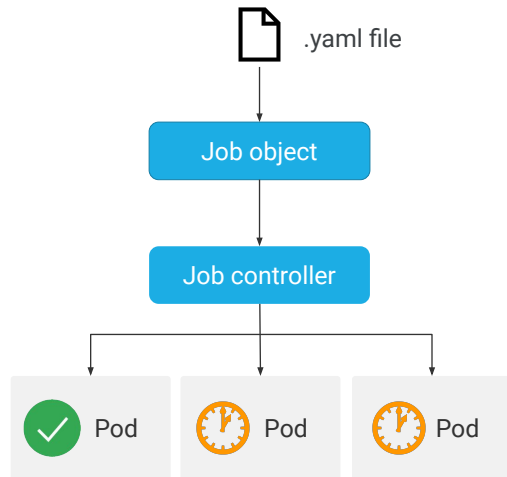
In a worker queue parallel Job, each Pod works on several items from a queue and then exits when there are no more items. Because the workers, the Pods themselves detect when the workqueue is empty, and the Job controller does not know about the workqueue. It relies on the workers to signal when they are done working by terminating. You create a parallel Job to process a worker queue by specifying a parallelism value and leaving spec.completions unset.

In this example, with parallelism set to 3, the Job controller will launch 3 Pods simultaneously. The Job will consider all tasks complete as soon as any of these three Pods successfully finishes its task.

At some point, one of the Pods terminates successfully. The applications running in the remaining Pods detect this completion state and finish, causing the remaining Pods to shut themselves down.

One Pod terminating the Job successfully is considered to be 'finished successfully.'

Inspecting a Job

```
$ kubectl describe job [JOB_NAME]
```

```
$ kubectl get pod -l [job-name=my-app-job]
```

Like other objects, Jobs can be inspected using a kubectl 'describe' command. The Pods can be filtered using the kubectl 'get' command and label selector. Job details can also be viewed from the Cloud Console.

Scaling a Job

```
$ kubectl scale job [JOB_NAME]
--replicas [VALUE]
```

Scale

Scale a workload to a new size.

Replicas

```
1
```

Jobs can be scaled either from a command-line or the Cloud Console. This is done by changing the parallelism value.

Failing a Job

```
apiVersion: batch/v1
kind: Job
metadata:
  name: my-app-job
spec:
  backoffLimit: 4
  activeDeadlineSeconds: 300
  template:
[...]
```

Pods can fail all the time. One way to limit Pod failure is through the 'backoffLimit' described earlier.

Another option is to use the activeDeadlineSeconds setting, where you set an active deadline period for the Job to finish. If the Job fails to complete within this limit the Job and all its Pods are terminated with a 'deadline exceeded' reason. The deadline count starts when the Job starts. Active deadline seconds has precedence over backoffLimit.

Deleting a Job

```
$ kubectl delete -f [JOB_FILE]
```

```
$ kubectl delete job [JOB_NAME]
```

```
$ kubectl delete job [JOB_NAME]
  --cascade false
```

Google Cloud

You can use either one of these kubectl 'delete' commands to delete a Job. When you delete a Job, all of its Pods are also deleted.

If you want to retain the Job Pods, set the cascade flag to false during the deletion.

Jobs can also be deleted directly from the Cloud Console.

Setting up a cron schedule under the CronJob spec

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: my-app-job
spec:
  schedule: "*/1 * * * *"
  jobtemplate:
    spec:
      template:
        spec:
[...]
```

Google Cloud

CronJob is a Kubernetes object that creates Jobs in a repeatable manner to a defined schedule. CronJobs are called that because they are named after cron, the standard Unix/Linux mechanism for scheduling a process.

The schedule field accepts a time in the Unix/Linux standard format for specifying a cron job. In this example, a cron schedule has been set up under the CronJob specification to run every 1 minute. The JobTemplate defines the Job specifications, just as I told you about in the previous lesson about Jobs.

Setting up a cron schedule under the CronJob spec

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: my-app-job
spec:
  schedule: "*/1 * * * *"
  startingDeadlineSeconds: 3600
  jobtemplate:
    spec:
      template:
        spec:
[...]
```

If the Job defined in this manifest is scheduled to start every minute, what happens if the Job isn't started at the scheduled time? By default, the CronJob looks at how many times the Job has failed to run since it was last scheduled. If that failure count exceeds 100, an error is logged and the Job is not scheduled, This will prevent an error with a CronJob resulting in an endless accumulation of failed attempts over time.

This behavior can be controlled using the startingDeadlineSeconds value. Instead of looking at the number of failed attempts to run the job since it was last successfully run, you can define the startingDeadlineSeconds attribute which defines the window of time to sum the number of failed attempts.

This changes the window that the controller examines. Now instead of looking back to the last time the job was scheduled, when startingDeadlineSeconds is set, the window starts at startingDeadlineSeconds before now.

Setting up a cron schedule under the CronJob spec

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: my-app-job
spec:
  schedule: "*/1 * * * *"
  startingDeadlineSeconds: 3600
  concurrencyPolicy: Forbid
  jobtemplate:
    spec:
      template:
        spec:
[...]
```

Google Cloud

Depending on how frequently Jobs are scheduled, and how long it takes to finish the defined task, the CronJob might end up executing more than one Job concurrently.

You use the concurrencyPolicy value to define whether concurrent executions are permitted, with the values 'Allow,' 'Forbid,' or 'Replace'. In the case of Forbid, if the existing Job hasn't finished, the CronJob won't execute a new Job. With Replace, the existing Job will be replaced by the new Job. This policy only applies to Jobs that were created using the same CronJob. Other CronJobs and their Jobs aren't considered or affected.

Setting up a cron schedule under the CronJob spec

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: my-app-job
spec:
  schedule: "*/1 * * * *"
  startingDeadlineSeconds: 3600
  concurrencyPolicy: Forbid
  suspend: True
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 1
  jobtemplate:
    spec:
[...]1
```

You can stop execution of individual Jobs by a CronJob by setting the suspend property to True. When this is set, all new Job executions are suspended. Suspended executions are still counted as missed Jobs.

The limit to the number of successful and failed Jobs to be retained in history is configured by the fields 'successfulJobsHistoryLimit' and 'failedJobsHistoryLimit.'

CronJobs can be managed using kubectl

Create a CronJob

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

Inspect a CronJob

```
$ kubectl describe cronjob [NAME]
```

Delete a CronJob

```
$ kubectl delete cronjob [NAME]
```

Google Cloud

CronJobs operate in the same manner as the Job itself. You can create, inspect, and delete CronJobs using the kubectl commands shown. You'll learn how to execute a CronJob in the Labs.

Agenda

Deployments

Lab: Creating Google Kubernetes
Engine Deployments

Jobs and CronJobs

Lab: Deploying Jobs on Google
Kubernetes Engine

Cluster Scaling

Controlling Pod Placement

Getting Software into Your Cluster

Lab: Configuring Pod Autoscaling
and Node Pools

Summary

Google Cloud

Lab Intro

Deploying Jobs on Google Kubernetes Engine

Google Cloud
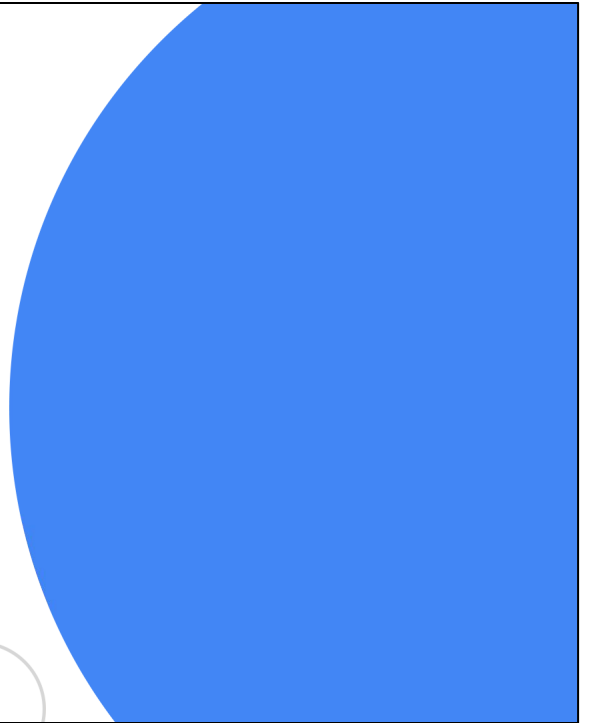
In this lab, you'll define and run Jobs and CronJobs. In GKE, a Job is a controller object that represents a finite task. Your first task will be to create a GKE cluster, create a Job, inspect its status, and then remove it.

You can create CronJobs to perform finite, time-related tasks that run once or repeatedly at a time that you specify. For your second task, you'll create and run a CronJob. In order to stop the CronJob and clean up the Jobs associated with it, you will then delete the CronJob.

Agenda

Google Cloud

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.
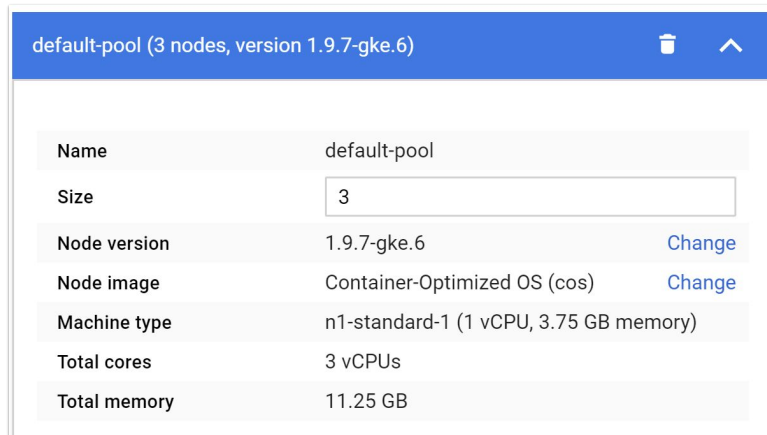
Scaling down a cluster using the gcloud command

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

You can also resize a cluster manually from the command line using the 'resize' gcloud command. As with the Cloud 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.

Scaling a cluster from the Cloud Console

default-pool (3 nodes, version 1.9.7-gke.6)

Name	default-pool
Size	3
Node version	1.9.7-gke.6 — Change
Node image	Container-Optimized OS (cos) — Change
Machine type	n1-standard-1 (1 vCPU, 3.75 GB memory)
Total cores	3 vCPUs
Total memory	11.25 GB

Google Cloud

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 Cloud Console. New nodes created during the process automatically self-register to the cluster within the Google Cloud 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 Cloud 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
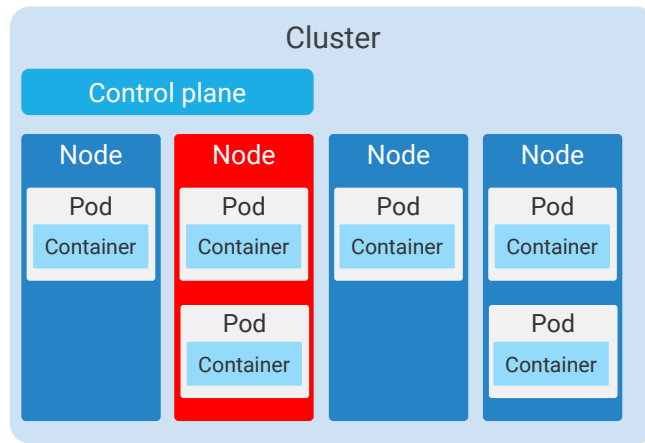
You can also manually decrease the cluster size. When you reduce the size of a cluster, the nodes to be removed are selected randomly. The resize process doesn't differentiate between nodes that are running Pods and ones that are empty.

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.

Scale up a cluster with autoscaling

Google Cloud
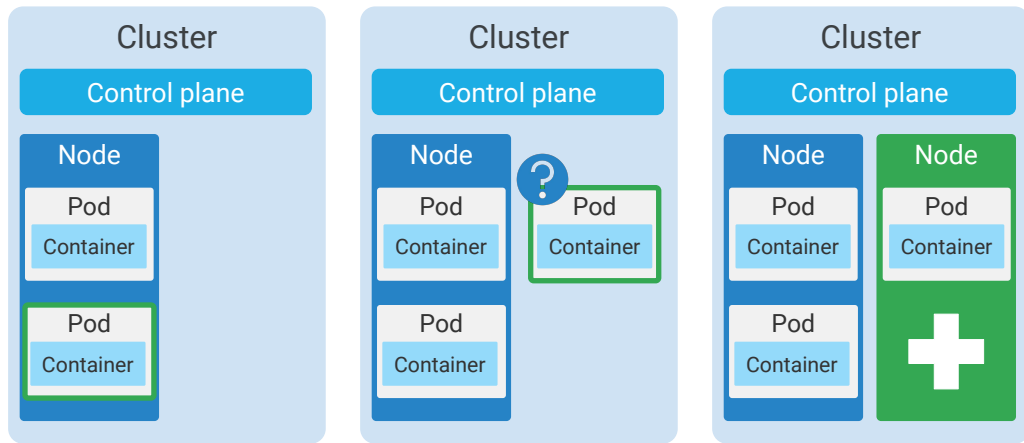
Cluster autoscaler controls the number of worker nodes in response to workload demands. GKE's cluster autoscaler can automatically resize a cluster based on the resource demands of your workload. By default, the cluster autoscaler is disabled. Cluster autoscaling allows you to pay only for resources 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

- There can be no scale-up events pending.
- 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.

The Autoscaler removes nodes that remain below 50% utilization
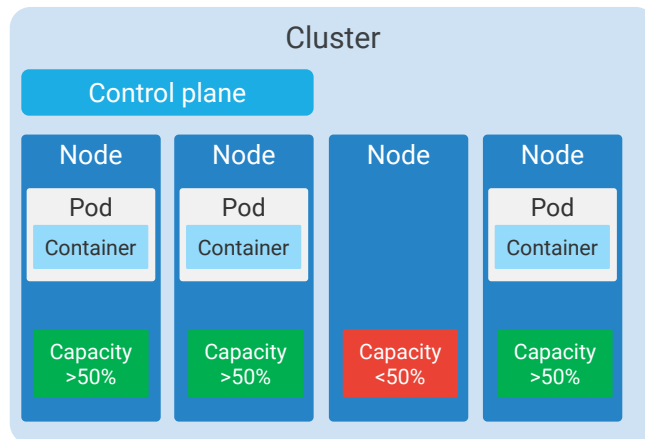
For each of the remaining nodes, the cluster autoscaler adds up the total CPU and memory requests for the running Pods. If this total is less than 50% of a node's allocatable capacity, the cluster autoscaler will monitor that node for the next 10 minutes. If the total remains below 50%, the node is deleted. After deleting a node, the cluster autoscaler re-analyzes the cluster to see whether additional nodes can be deleted.

Best practices for working with autoscaled clusters

❌	✅
Don't run Compute Engine autoscaling	Specify correct resource requests for Pods
Don't manually resize a node using the gcloud command	Use PodDisruptionBudget to maintain the app's availability
Don't manually modify a node	

Google Cloud

Let's look at some best practices for working with autoscaled clusters.

First, don't run Compute Engine autoscaling for managed instance groups on these nodes. The GKE autoscaler is separate from Compute Engine autoscaling.

Don't manually resize a node pool using a gcloud command when the cluster autoscaler is enabled. This might lead to cluster instability and result in the cluster having wrong node pool sizes.

Don't modify autoscaled nodes manually. All nodes in a node pool should have the same capacity, labels, and system Pods. In addition, if you change the labels for one node directly using kubectl, the changes won't be propagated to the other nodes in the node pool.

Do specify correct resource requests for Pods. This will allow Pods to work efficiently with the cluster autoscaler. If you don't know the resource needs of Pods, measure them under a test load.

Finally, do use PodDisruptionBudgets. It's expected that the Pods belonging to the controller can be safely terminated and relocated. If your application cannot tolerate such disruption, maintain your application's availability using PodDisruptionBudgets.

Setting a node pool size

Min

Max

Node pool = 0
Cluster size ≠ 0

MAX = 5,000 nodes x 30 Pods

Increase quota limits to avoid disruption

Google Cloud
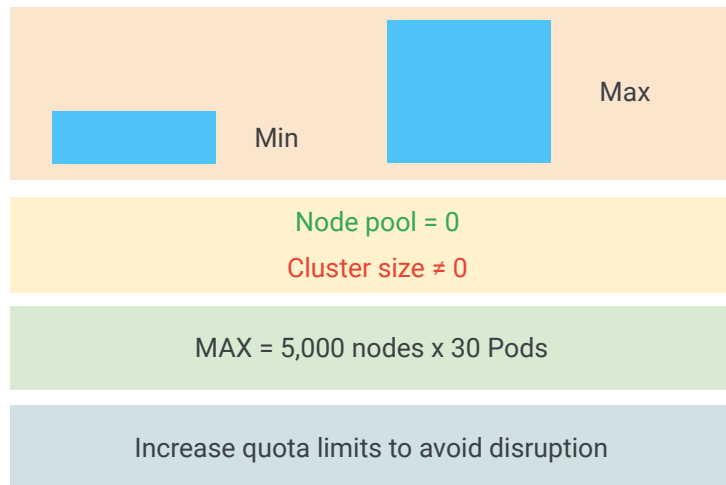
A cluster will contain one or more node pools. The node pool size can be set by specifying a minimum and maximum size.

You can scale down some of the node pools in a cluster to 0; however, the overall cluster size can never be scaled to 0. At least one node is required within a cluster to have system Pods running.

The cluster autoscaler has been tested up to a maximum of 5000 nodes, with each running 30 Pods.

However, standard Google Cloud quota limits for your total number of Compute Engine instances will apply. If you haven't increased your default quota, you'll eventually see disruption, and new VMs won't be started until the quota limit is raised.

gcloud commands for autoscaling

Create a cluster with autoscaling enabled	Enable autoscaling for an existing node pool
```gcloud container clusters create [CLUSTER_NAME] --num-nodes 30 \ --enable-autoscaling --min-nodes 15 --max-nodes 50 [--zone COMPUTE_ZONE]```	```gcloud container clusters update [CLUSTER_NAME] --enable-autoscaling \ --min-nodes 1 --max-nodes 10 --zone [COMPUTE_ZONE] --node-pool [POOL_NAME]```
Add a node pool with autoscaling enabled	Disable autoscaling for an existing node pool
```gcloud container node-pools create [POOL_NAME] --cluster [CLUSTER_NAME] --enable-autoscaling --min-nodes 15 --max-nodes 50 [--zone COMPUTE_ZONE]```	```gcloud container clusters update [CLUSTER_NAME] --no-enable-autoscaling \ --node-pool [POOL_NAME] [--zone [COMPUTE_ZONE] --project [PROJECT_ID]]```

Google Cloud

Here are some gcloud commands for autoscaling. You can start a cluster with autoscaling enabled for the default pool.

You can add a new pool with autoscaling enabled, or enable autoscaling for an existing node pool.

You can also disable autoscaling for an existing node pool. If autoscaling is disabled, the node pool size will be fixed at the cluster's current node pool size.

You can carry out all of these actions from the Cloud Console also.

By default with zonal clusters all resources (nodes and the control plane) are created in the same zone. If you enable secondary zones then all node pools are duplicated in the secondary zone similar to the way pools are duplicated for regional clusters.

Agenda

Deployments

Lab: Creating Google Kubernetes
Engine Deployments

Jobs and CronJobs

Lab: Deploying Jobs on Google
Kubernetes Engine

Cluster Scaling

Controlling Pod Placement

Getting Software into Your Cluster

Lab: Configuring Pod Autoscaling
and Node Pools

Summary

Google Cloud

In this lesson, we'll look at controlling Pod placement. In Kubernetes, Pod placement can be controlled with labels and taints on nodes, and node affinity rules and tolerations in the deployment specifications.

Controlled scheduling
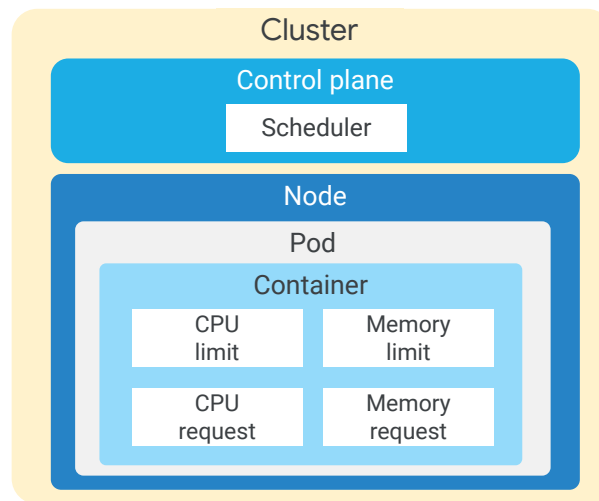
When you specify a Pod, you can optionally specify how much CPU and memory (RAM) each Container needs. When Containers have resource requests specified, the scheduler can make better decisions about which nodes to place Pods on. And when Containers have their limits specified, contention for resources on a node can be handled in a specified manner.

A Pod sums each container's resource requests/limits and sets up it own requests and limits, depending on the number of containers it's running.

A scheduler assigns a Pod to a node based on resource requests and limits set by the containers within the Pod.

The scheduler ensures that a Pod's requests and limits are within a node's capacity. It also spreads Pods across nodes automatically. These nodes can be set up across different compute zones.

When nodes are started, the kubelet automatically adds labels to them with zone information.

Kubernetes will automatically spread the pods in a replication controller or service across nodes in a single-zone cluster, to reduce the impact of failures. With multiple-zone clusters, this spreading behavior is extended across zones to reduce the impact of zone failures.

But what if you want to run certain types of applications on a specific node? For example, if you want run a database server on a node with an SSD.

Nodes must match all the labels present under the nodeSelector field

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mysql
  labels:
    env: test
spec:
  containers:
  - name: mysql
    image: mysql
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
[…]
```

```yaml
apiVersion: v1
kind: Node
metadata:
  name: node1
  labels:
    disktype: ssd
[…]
```

Google Cloud

For a Pod to run on a specific node, that node must match all the labels present under the nodeSelector field in a Pod. NodeSelector is a Pod specification field that specifies one or more labels.

Nodes must match all the labels present under the nodeSelector field

```
apiVersion: v1
kind: Pod
metadata:
  name: mysql
  labels:
    env: test
spec:
  containers:
  - name: mysql
    image: mysql
    imagePullPolicy: IfNotPresent
  nodeSelector:
    cloud.google.com/gke-nodepool=ssd
[...]
```

Google Cloud

The node labels may be automatically assigned, for example the label cloud.google.com/gke-nodepool is automatically created by GKE and contains the name of the node pool, or you may need to add labels directly to identify nodes that meet specific criteria.

For example, this Pod will only run on a node that has a label that indicates the node is a member of the GKE node pool called ssd. If the node's labels are changed, running Pods are not affected. NodeSelector is only used during Pod scheduling.

Node affinity is conceptually similar to nodeSelector

```
apiVersion: v1
kind: Pod
metadata:
   name: with-node-affinity
spec:
   affinity:
     nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
           nodeSelectorTerms:
              - matchExpressions:
                 - key: beta.kubernetes.io/instance-type
                 operator: In
                 values:
                    - n1-highmem-4
                 - n1-highmem-8
[…]
```

Next, let's look at node affinity and anti-affinity features.

Like NodeSelectors, node affinity also allows you to constrain which nodes your Pod can be scheduled on, based on labels, but the features are more expressive and can be used to constrain against the labels of both nodes and other Pods running on nodes.

Unlike NodeSelector, where a Pod won't be scheduled if the NodeSelector requirements aren't met, it's possible to define node affinity and node anti-affinity rules as preferences rather than requirements so that they won't prevent a Pod from being launched if the preferences cannot be met. Think of these as allowing you to set soft preferences in addition to hard requirements.

Affinity and anti-affinity rules

```
apiVersion: v1
kind: Pod
metadata:
   name: with-node-affinity
spec:
   affinity:
      nodeAffinity:
         requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
               - matchExpressions:
                  - key: accelerator-type
                  operator: In
                  values:
                     - gpu
                  - tpu
```

Google Cloud

Affinity and anti-affinity rules are denoted by the 'requiredDuringSchedulingIgnoredDuringExecution' and 'preferredDuringSchedulingIgnoredDuringExecution' rules.

The keywords contain the string 'IgnoredDuringExecution' part to remind us that if the labels change, Pods already running aren't affected.

The 'requiredDuringScheduling' rule shown here is the hard requirement, similar to NodeSelector, that must be met for the Pod to be scheduled. The 'preferredDuringScheduling' format of this rule defines a soft preference that is covered in the next slide.

In this hard-requirements example, the Pod can only run on nodes with a label whose key is accelerator-type, and whose value is either gpu or tpu. These can be any labels you want to use. In this case, the Pod runs an application that can benefit from either a GPU or TPU accelerator. Those labels must have been added to compute nodes previously.

Note that a single NodeSelectorTerms is used here. You can use multiple NodeSelectorTerms, but only one is required for scheduling.

In this example, only one matchExpression is mentioned; but you can add multiple matchExpressions. The node must satisfy all the listed matchExpressions in each nodeSelector. Logically they are joined using boolean AND.

With the In operator you can have multiple values, but only one is required to match. Two values are noted here for a single key. Logically the In operator acts as a boolean OR.

You can use other operators, such as NotIn, Exists, DoesNotExist, Gt (for greater-than) and Lt (for less than). For example, if the NotIn operator was used in this example, you would be configuring a node anti-affinity rule.

Defining the intensity of preference

```
apiVersion: v1
kind: Pod
metadata:
   name: with-node-affinity
spec:
   affinity:
     nodeAffinity:
       preferredDuringSchedulingIgnoredDuringExecution:
         - weight: 90
         preference:
           - matchExpressions:
             - key: cloud.google.com/gke-nodepool
             operator: In
             values:
               - n1-highmem-4
             - n1-highmem-8
```

Google Cloud

Specifying 'preferredDuringSchedulingIgnoredDuringExecution' creates a soft preference rule. The value for the weight of this preference ranges from 1, the weakest preference level, to 100 for the highest preference. In this example, the soft preference is also given a weight set to 1. What makes it a soft preference, rather than a hard requirement, is our use of the "preferred" keyword instead of the "required" keyword.

When the scheduler evaluates these preferences, each node that the Pod might be scheduled on receives a total weight score based on all the requirements it meets, such as resource requests, resource limits, and other nodeAffinity rules, such as 'RequiredDuringSchedulingIgnoredDuringExecution.' The weight of 'preferredDuringSchedulingIgnoredDuringExecution' is also added to this total score. The scheduler then assigns the Pod to the node with the highest total score. The weight defines the intensity of preference.

In the example shown here a pod with a heavy duty workload is configured with node affinity rules that configure a strong preference for nodes containing labels that indicate they are members of GKE node pools called either n1-highmem-4 or n1-highmem-8. The node pools could have any name but it is useful to give them names that reflect the type of compute instance used to create the nodes since all members of a node-pool are identical and this allows you to create preferences for

specific types of hardware using the node-pool names, which are automatically configured as labels on all nodes by GKE.

Inter-pod affinity and anti-affinity features are built on the node affinity concept

```
[...]
metadata:
    name: with-pod-affinity
spec:
    affinity:
        podAntiAffinity:
            preferredDuringSchedulingIgnoredDuringExecution:
                - weight: 100
                podAffinityTerm:
                    labelSelector:
                        - matchExpressions:
                            - key: app
                            operator: In
                            values:
                                - webserver
                topologyKey: failure-domain.beta.kubernetes.io/zone
```

Google Cloud

Inter-pod affinity and anti-affinity features extend the node affinity concept to include rules based on Pod labels that are already running on the node, instead of only on labels on the nodes themselves.

A Pod that is required or prefers to run on the same node as other Pods can be configured with podAffinity rules. Pods that must not, or should not, be scheduled on the same node as other Pods can be configured with podAntiAffinity rules. You can see this is similar to node affinity with the addition of some extra fields. Notice in this case that this Pod strongly prefers not to be scheduled on the same node as Pods with label key: value of app:webserver. This is a strong but still soft preference, because the weight of this podAntiAffinity rule has the highest possible value of 100, but the rule is still preferredDuringScheduling.

Using topologyKeys, you can also specify affinity and non-affinity rules at a higher level than just specific nodes. For example, to ensure that Pods are not co-located in the same zone, not just the same node, you define a topologyKey to specify that a podAntiAffinity rule should apply at the zone topology level.
You can use topologyKey to specify topology domains, such as node, zone, and region.

The Pod shown here has a podAntiAffinity rule with topologyKey set so that it prefers

not to be scheduled in the same zone that's already running at least one Pod with label key: and value of app:webserver.

Combining inter-pod affinity and anti-affinity

Pod (#1)
(Label – app:webserver)

Requirement: No app:webserver
on the same zone

Preference: Prefer app:cache
on the same node

Pod (#2)
(Label – app:webserver)

Requirement: No app:webserver
on the same zone

Preference:Prefer app:cache
on the same node

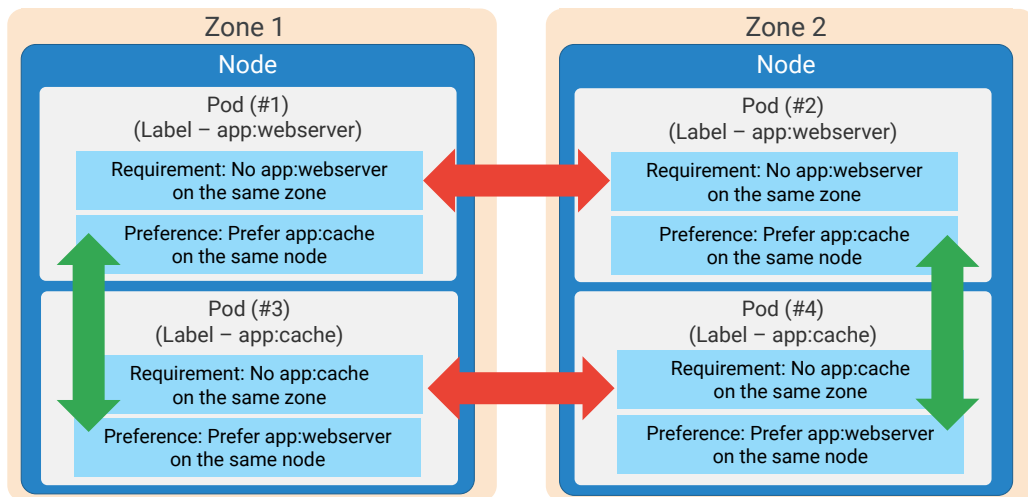Pod (#3)
(Label – app:cache)

Requirement: No app:cache
on the same zone

Preference: Prefer app:webserver
on the same node

Pod (#4)
(Label – app:cache)

Requirement No app:cache
on the same zone

Preference: Prefer app:webserver
on the same node

Google Cloud

Here's an example that combines Pod affinity and anti-affinity. Pod number 1, with label app:webserver, has both a hard requirement and a soft preference. In this Pod, the hard requirement has an anti-affinity rule saying that other Pods with the label app:webserver aren't allowed in the same zone. On the other hand, the soft preference has an affinity rule where it prefers to have other Pods with label app:cache on the same node.

The cache Pods, however, prefer to be scheduled on the same node as a web server but have a hard requirement that prevents multiple cache Pods from being deployed in the same zone.

The resulting distribution has the two web server Pods repelling each other at the zone level, and attracting cache Pods at the node level. Both cache Pods also repel each other at the zone level and attract web server Pods at the node level.

Combining inter-pod affinity and anti-affinity

Zone 1

Node

Pod (#1)
(Label − app:webserver)

Requirement: No app:webserver
on the same zone

Preference: Prefer app:cache
on the same node

Pod (#3)
(Label − app:cache)

Requirement: No app:cache
on the same zone

Preference: Prefer app:webserver
on the same node

Zone 2

Node

Pod (#2)
(Label − app:webserver)

Requirement: No app:webserver
on the same zone

Preference: Prefer app:cache
on the same node

Pod (#4)
(Label − app:cache)

Requirement: No app:cache
on the same zone

Preference: Prefer app:webserver
on the same node

Google Cloud

Pod #1, which is a web server, is already running on a node in zone 1.

Pod #2, a second web server, is scheduled.

Pods #1 and #2 repel each other based on the hard requirement.

Therefore, Pod #2 is scheduled to run in a different zone: zone 2...

And on a different node.

Pod #3, which is a cache server, has a soft preference to run on the same node as web server Pods; so it can be scheduled on the same node as either of the web servers in Pod #1 or Pod #2.

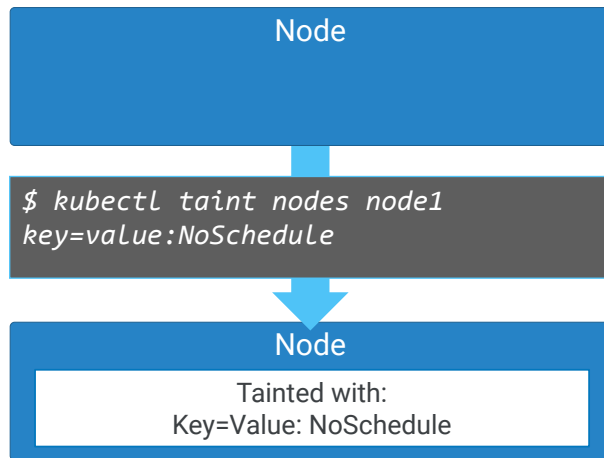It's scheduled to run alongside the web server in Pod #1.

Finally, the the second cache server in Pod #4 has to be scheduled.

It will be repelled from the cache server in Pod #3 with its hard requirement, ...

But can run on a different zone along with the web server in Pod #2. Additionally, there is a soft preference for the cache server Pods to run on the same node as a web server Pod, instead of on a different node.

This type of pattern allows you to specify affinity and anti-affinity rules to co-locate Pods at different topological layers and control the distribution of Pods across topological layers.

Taints allow a node to repel Pods


```
$ kubectl taint nodes node1
key=value:NoSchedule
```

Node
Tainted with:
Key=Value: NoSchedule

Google Cloud

Node affinity attracts Pods, and anti-affinity repels them. You can also use taints to prevent Pods from being scheduled on specific nodes. You are probably wondering, "Why do we need both Taints and affinity settings?" Having the choice gives you more management flexibility. You configure NodeSelector, affinity, and anti-affinity rules on Pods. By contrast, you configure Taints on nodes, and they apply to all Pods in the cluster. You should use whichever mechanism lets you express the behavior you want most economically.

To taint a nodes, use the kubectl taint command.
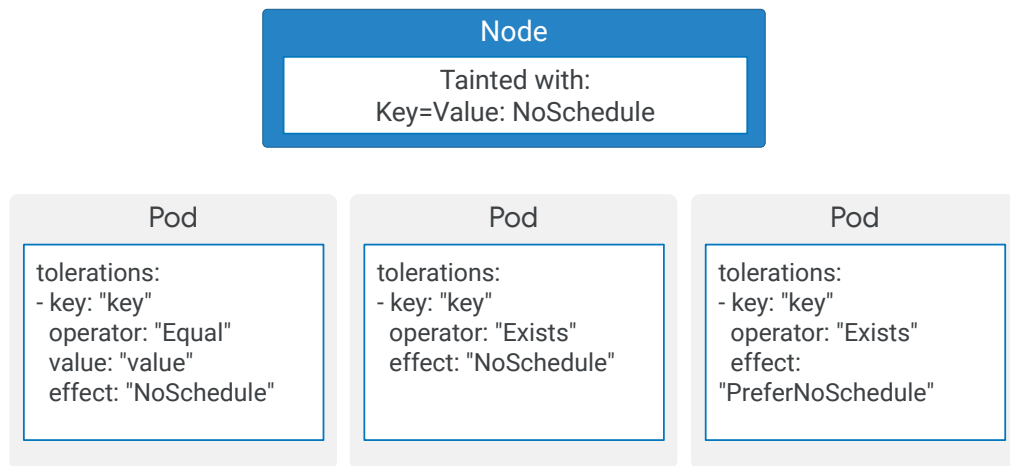
The taint has a key with a value and a taint effect.

This taint with the NoSchedule effect limits all Pods from scheduling on this particular node.

You can apply multiple taints to a node

Node

Tainted with:
Key=Value: NoSchedule
Key2=Value2: NoExecute

Google Cloud
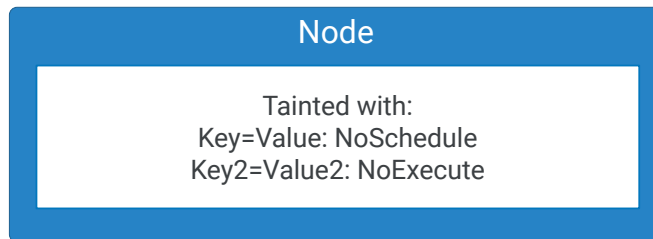
You can apply multiple taints to a node. In this case, no Pods can be scheduled, and all running Pods will be evicted.

You can configure Pods to tolerate taints

Node
Tainted with:
Key=Value: NoSchedule

Pod
tolerations:
- key: "key"
  operator: "Equal"
  value: "value"
  effect: "NoSchedule"

Pod
tolerations:
- key: "key"
  operator: "Exists"
  effect: "NoSchedule"

Pod
tolerations:
- key: "key"
  operator: "Exists"
  effect:
  "PreferNoSchedule"

Google Cloud

So how can you schedule a node here? Tolerations enable this.

Tolerations are applied to Pods. A Toleration is a mechanism that allows a Pod to counteract the effect of a taint that would otherwise prevent the Pod from being scheduled or continue to run on a Node. A toleration field consists of a key, value, effect, and operator. A Pod's toleration will match a taint if the keys in the toleration and the taint are the same, the effects are the same, and the operator accepts the values.

The operator field allows some flexibility. When an operator field is Equal, the value must also be equal.
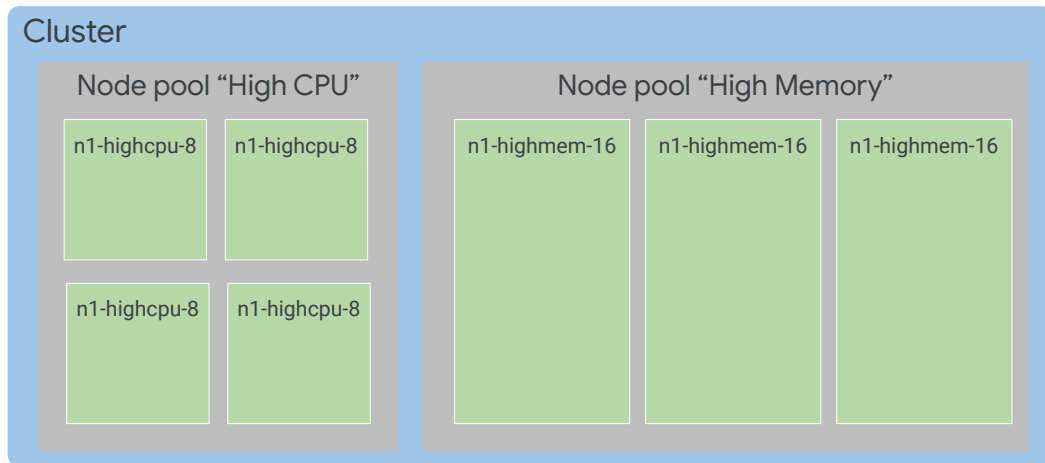However, when an operator field is set to Exists, only the keys and effects must match for the Toleration to apply, even if the value field isn't specified.

Three effect settings can be applied:
- NoSchedule is a scheduling hard limit that prevents scheduling of Pods unless there is a Pod toleration with the NoSchedule effect that matches.
- PreferNoSchedule tells the scheduler to try to not place a Pod that doesn't have a matching Pod toleration for the node's taint, but this is only a soft limit, and the Pod might still be scheduled.
- NoExecute will evict running Pods from the nodes unless the Pods each have

- at least one matching toleration with the NoExecute effect.

Use node pools to manage different kinds of nodes

Cluster

Node pool "High CPU"

n1-highcpu-8	n1-highcpu-8

n1-highcpu-8	n1-highcpu-8

Node pool "High Memory"

n1-highmem-16	n1-highmem-16	n1-highmem-16

Google Cloud

You've learned a lot of techniques for constraining Pods to particular nodes. You may be wondering, "What's the simplest way to manage this situation?" In GKE, the concept of node pools allows us to abstract away a lot of the complexity, because one of the most common reasons to manage where Pods run is to get them onto the right hardware. Remember that the nodes in a node pool always have the same hardware. So, for example, you can use NodeSelectors to direct Pods to the right node pool.

In this example, this GKE cluster has two node pools. One node pool is called "High CPU" because the nodes that make it up are machines with a high CPU to memory ratio, with 8 virtual CPUs and 7.2 gigabytes of memory. The other pool is called "High Memory" because its virtual machines have a much higher ratio of memory with with 16 virtual CPUs and 104 gigabytes of memory in each node.

Direct Pods to desired nodes using nodeSelectors

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeSelector:
    cloud.google.com/gke-nodepool: small
[...]
```

Google Cloud

GKE automatically labels the nodes in each node pool with the node pool name you supply. You can use the full logical power of nodeSelectors to direct Pods to nodes using these names. For example, if you have a fleet of front-end Web servers that do little processing themselves, it might be most economical to constrain those Pods to smaller nodes. In this example, the nginx Pod named by the YAML template must run on a node in the "small" node pool.

Or use node pool names with affinity, anti-affinity rules

```
apiVersion: v1
kind: Pod
metadata:
  name: rendering-engine
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
            - key: cloud.google.com/gke-nodepool
              operator: NotIn
              values:
              - small
```

If nodeSelectors are not expressive enough, you can also use the affinity specifications we learned in this module with node pool names. In this example, we've specified that this pod, which performs a compute-intensive activity, can be scheduled onto any node pool other than the one named "small."

In this lesson, you saw how to control the scheduling of Pods based on node selector, node affinity, Pod affinity, Pod anti-affinity, and taints with tolerations. Using these, Pods can be distributed across the zones, co-located with Pods they need to remain close to, scheduled on nodes with certain features, or prevented from running on nodes that are unsuitable.

Agenda

Google Cloud

You have now learned several useful and expressive ways to manage your workloads
in your Kubernetes clusters. But what about the software that those workloads run?
Where do you get it?

How to get software

- Build it yourself, and supply your own YAML.
- Use Helm to install software into your cluster.

An earlier module in this course series discussed packaging software into containers, with tools such as Cloud Build. You can use Container Registry to store private container images, and Kubernetes can fetch these images from the registry and run them in Pods. But it's up to you to define the deployment patterns and services that make them run reliably and usefully. You'll create your own YAML manifests and maintain them yourself in a source-control system of your choice.

You are probably thinking, "Writing YAML files is not much fun. And haven't other people already figured out the best patterns for deploying popular open-source software into Kubernetes clusters? Why do I have to figure all that out starting from scratch?" You don't. This problem is what Helm helps us solve.

Organize Kubernetes objects in packages and deploy complex packages

Charts

Create   Version   Share   Publish   Download   Configure   Release

Google Cloud
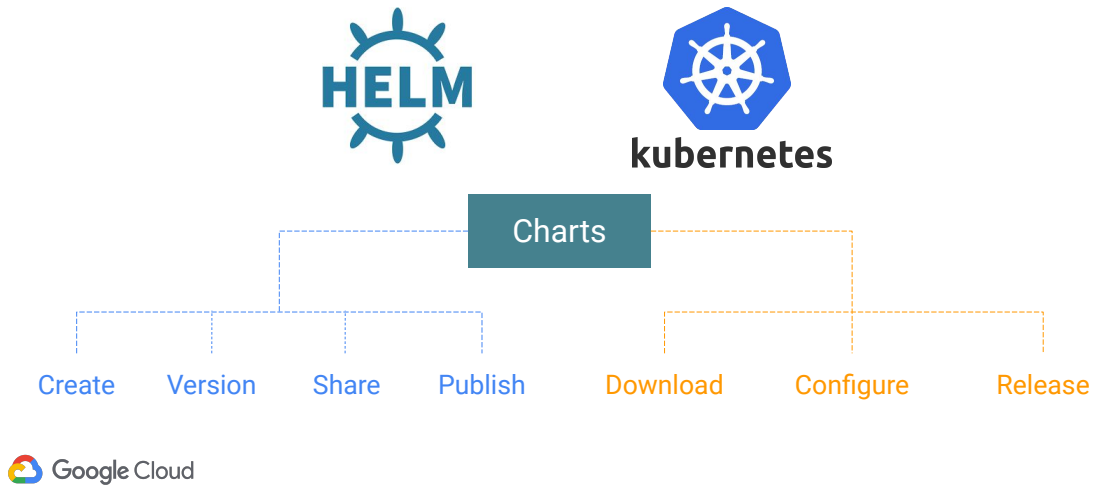
Helm is an open-source package manager for Kubernetes in the same way that apt-get and yum are package managers for Linux.
Helm lets developers organize Kubernetes objects in packages called "charts." Charts are easily created, versioned, shared, and published.

Charts manage the deployment of complex applications. You can think of a chart as a parameterized YAML template. Helm charts know what parameters are needed to make them work. For example, how many instances of each component do you need, and what resource constraints should they work under? Developers of Helm charts define these parameters, and even supply helpful defaults.

When you install a Helm chart, Helm fills in the parameters you supply and deploys a release, which is a specific instance of the chart, to the cluster.

There are two components in the Helm architecture
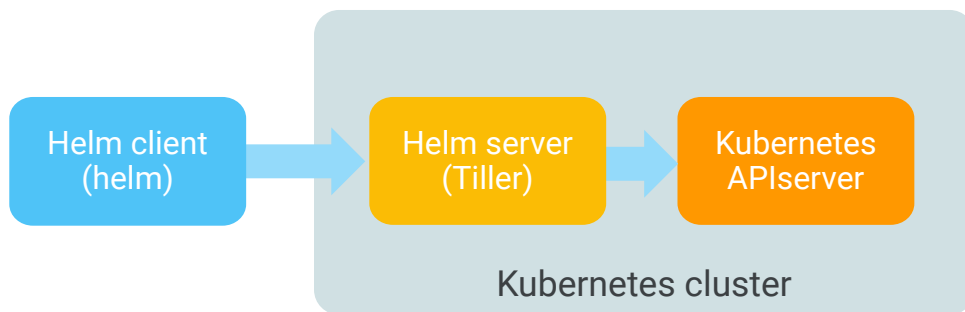
First, there's a command-line client, which is also called helm, that allows you to develop new charts and manage chart repositories.

The second component is the Helm server, called Tiller, which runs within the Kubernetes cluster.

Tiller interacts with the Kubernetes APIserver to install, upgrade, query, and remove Kubernetes resources. It also stores the objects that represent Helm chart releases.

How to get software

- Build it yourself, and supply your own YAML.
- Use Helm to install software into your cluster.
- Use Google Cloud Marketplace to install both open-source and commercial software.

Using Helm lets us deploy open-source software into our Kubernetes cluster with less work and less risk of error. But you still must manage Helm itself, and you still must find the tools you want to use. What if there were something even simpler?

Google Cloud Marketplace offers ready-to-go development stacks, solutions, and services to accelerate development for popular open-source and commercial software packages. Marketplace contains many Kubernetes-based tools that are ready to install into your existing GKE clusters. You don't have to set up Helm to use it. Instead, you just click a package you want, specify any needed parameters, and launch its installation into your cluster. Google Cloud Marketplace uses kubectl commands or Helm charts to automate the installation.

Agenda

Deployments

Lab: Creating Google Kubernetes
Engine Deployments

Jobs and CronJobs

Lab: Deploying Jobs on Google
Kubernetes Engine

Cluster Scaling

Controlling Pod Placement

Getting Software into Your Cluster

Lab: Configuring Pod Autoscaling
and Node Pools

Summary

Google Cloud

Lab Intro

Configuring Pod Autoscaling and
Node Pools

Google Cloud

In this lab, you'll set up an application in GKE, and then use a
HorizontalPodAutoscaler to autoscale the web application. You'll then work with
multiple node pools of different types, and apply taints and tolerances to control the
scheduling of Pods with respect to the underlying node pool. Your first task will be to
create a GKE cluster and a deployment manifest for a set of Pods within the cluster
that you'll then use to test DNS resolution of Pod and service names. In the next task
, you will configure the cluster to automatically scale the sample application that you
deployed earlier. Finally, you'll create a new pool of nodes using preemptible
instances, and then you'll constrain the web deployment to run only on the
preemptible nodes.

Agenda

Deployments

Lab: Creating Google Kubernetes
Engine Deployments

Jobs and CronJobs

Lab: Deploying Jobs on Google
Kubernetes Engine

Cluster Scaling

Controlling Pod Placement

Getting Software into Your Cluster

Lab: Configuring Pod Autoscaling
and Node Pools

Summary

Google Cloud

Summary

Create and use Deployments.

Create and run Jobs and CronJobs.

Use Helm Charts.

Scale clusters manually and automatically.

Configure Node and Pod affinity.

Google Cloud

That completes Deployment, Jobs and Scaling. In this module you learned how to create and use Deployments, create and run Jobs and CronJobs, use Helm Charts, how to scale clusters manually and automatically, and how to configure node and Pod affinity.