



Introduction to Kubernetes Workloads



Welcome to the final module in Getting Started with GKE. So far on this course we have learnt about Kubernetes and containerised applications, how to create a cluster and populate it with nodes, how to manage your clusters, and how to move existing non containerised applications into Kubernetes.

But Kubernetes is a large topic, that we don't have time to cover fully in this course. In this final lesson, we will look at some of the Next Steps you might want to take in your journey with Kubernetes.

Learn how to ...

Work with the `kubectl` command.

Understand how Deployments are used in Kubernetes.

Understand the networking architecture of Pods.

Understand Kubernetes storage abstractions.



In this module, you'll learn how to:

- Understand and use the KubeCTL command
- Understand how deployments are used in Kubernetes
- Understand the networking architecture of Pods in your cluster
- And Understand Kubernetes storage abstractions.

Agenda

The `kubectl` command

Deployments

Lab: Creating Google Kubernetes
Engine Deployments

Pod Networking

Volumes

Lab: Configuring Persistent Storage
for Google Kubernetes Engine

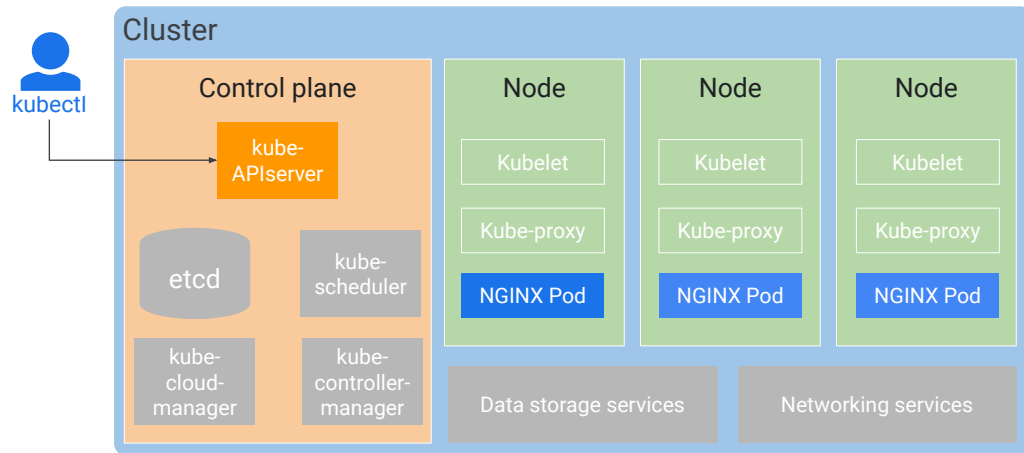
Summary



Let's start by discussing the `kubectl` command.

`Kubectl` is a utility used by administrators to control Kubernetes clusters. You use it to communicate with the Kube API server on your control plane.

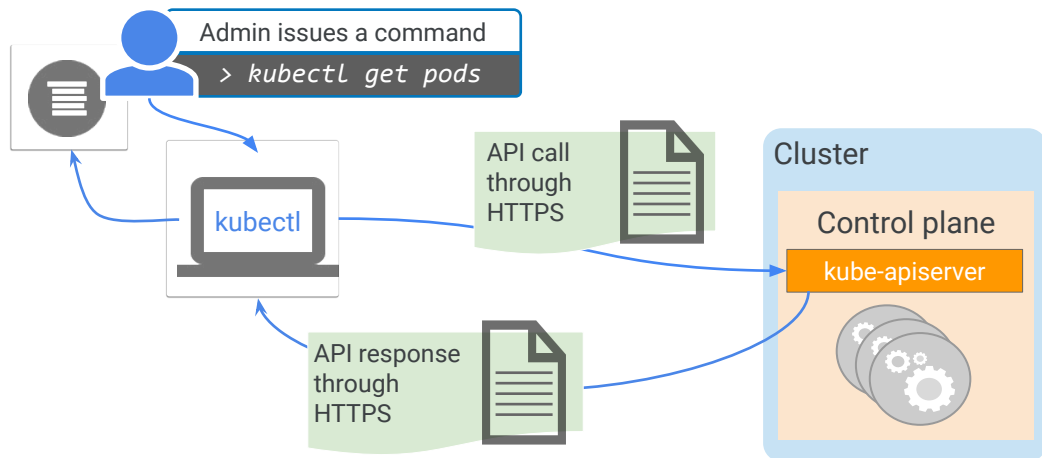
Kubectl transforms your command-line entries into API calls



 Google Cloud

Kubectl transforms your command-line entries into API calls that it sends to the Kube API server within your selected Kubernetes cluster. Before it can do any work for you, kubectl must be configured with the location and credentials of a Kubernetes cluster.

Use kubectl to see a list of Pods in a cluster



For example, take an administrator who wants to see a list of Pods in a cluster. After connecting `kubectl` to the cluster with proper credentials, the administrator can issue the `kubectl 'get pods'` command.

`Kubectl` converts this into an API call, which it sends to the Kube API server through HTTPS on the cluster's control plane server.

The Kube API server processes the request by querying `etcd`.

The Kube API server then returns the results to `kubectl` through HTTPS.

Finally, `kubectl` interprets the API response and displays the results to the administrator at the command prompt.

kubectl must be configured first

- Relies on a config file: `$HOME/.kube/config`.
- Config file contains:
 - Target cluster name
 - Credentials for the cluster
- Current config: `kubectl config view`.
- Sign in to a Pod interactively.



Before you can use `kubectl` to configure your cluster, you must configure it first.

`Kubectl` stores its configuration in a file in your home directory in a hidden folder named `.kube`.

The configuration file contains the list of clusters and the credentials that you'll use to attach to each of those clusters. You may be wondering where you get these credentials. For GKE, the service provides them to you through the `gcloud` command. I'll show you how that works in a moment.

To view the configuration, you can either open the config file or use the `kubectl` command: `'config view'`. Just to be clear here: `kubectl config view` tells you about the configuration of the `kubectl` *command itself*. Other `kubectl` commands tell you about the configurations of your cluster and workloads.

Connecting to a Google Kubernetes Engine cluster

```
$ gcloud container clusters \  
  get-credentials [CLUSTER_NAME] \  
  --zone [ZONE_NAME]
```



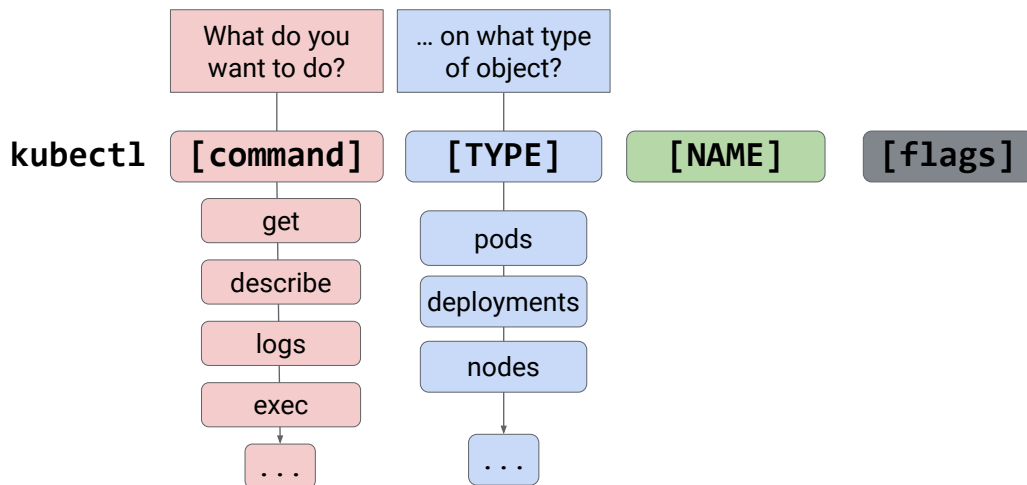
To connect to a GKE cluster with `kubectl`, retrieve your credentials for the specified cluster first. To do this, use the `get-credentials` `gcloud` command in any other environment where you've installed the `gcloud` command-line tool and `kubectl`. Both of these tools are installed by default in the Cloud Shell.

The `gcloud` `get-credentials` command writes configuration information into a config file in the `.kube` directory in the `$HOME` directory by default. If you rerun this command for a different cluster it'll update the config file with the credentials for the new cluster. You only need to perform this configuration process once per cluster in your Cloud Shell, because the `.kube` directory and its contents stay in your `$HOME` directory.

Can you figure out why the command is `gcloud get-credentials` rather than `kubectl get-credentials`? It's because the `kubectl` command requires credentials to work at all. The `gcloud` command is how authorized users interact with Google Cloud from the command line. The `gcloud get-credentials` command gives you the credentials you need to connect with a GKE cluster if you are authorized to do so.

In general, `kubectl` is a tool for administering the internal state of an existing cluster. But `kubectl` can't create new clusters or change the shape of existing clusters; for that you need the GKE control plane, which the `gcloud` command and the Cloud Console are your interfaces to.

The kubectl command syntax has several parts



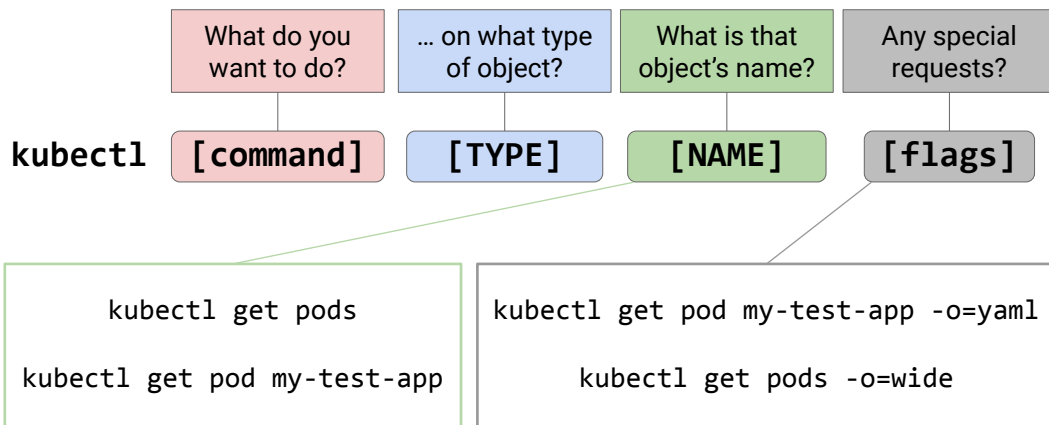
Once the config file in the `.kube` folder has been configured, the `kubectl` command automatically references this file and connects to the default cluster without prompting you for credentials. Now let's talk about how to use the `kubectl` command. Its syntax is composed of several parts: the command, the type, the name, and optional flags.

'Command' specifies the action that you want to perform, such as `get`, `describe`, `logs`, or `exec`. Some commands show you information, while others allow you to change the cluster's configuration.

'TYPE' defines the Kubernetes object that the 'command' acts upon. For example, you could specify `Pods`, `Deployments`, `nodes`, or other objects, including the cluster itself.

TYPE used in combination with 'command' tells `kubectl` what you want to do and the type of object you want to perform that action on.

The kubectl command syntax has several parts



'NAME' specifies the object defined in 'TYPE.' The Name field isn't always needed, especially when you're using commands that list or show you information. For example, if you run the command "kubectl get pods" without specifying a name, the command returns the list of all Pods. To filter this list you specify a Pod's name, such as "kubectl get pod my-test-app". kubectl then returns information only on the Pod named 'my-test-app'.

Some commands support additional optional flags that you can include at the end of the command.

Think of this as making a special request, like formatting the output in a certain way. You could view the state of a Pod by using the command "kubectl get pod my-test-app -o=yaml". By the way, telling kubectl to give you output in YAML format is a really useful tool. You'll often want to capture the existing state of a Kubernetes object in a YAML file so that, for example, you can recreate it in a different cluster.

You can also use flags to display more information than you normally see. For instance, you can run the command "kubectl get pods -o=wide" to display the list of Pods in "wide" format, which means you see additional columns of data for each of the Pods in the list. One noteworthy piece of extra information you get in wide format: which Node each Pod is running on.

The kubectl command has many uses

- Create Kubernetes objects
- View objects
- Delete objects
- View and export configurations



You can do many things with the kubectl command, from creating Kubernetes objects, to viewing them, deleting them, and viewing or exporting configuration files.

Just remember to configure kubectl first or to use the `--kubeconfig` or `--context` parameters, so that the commands you type are performed on the cluster you intended.

Agenda

The kubectl command

Deployments

Lab: Creating Google Kubernetes
Engine Deployments

Pod Networking

Volumes

Lab: Configuring Persistent Storage
for Google Kubernetes Engine

Summary



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.

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



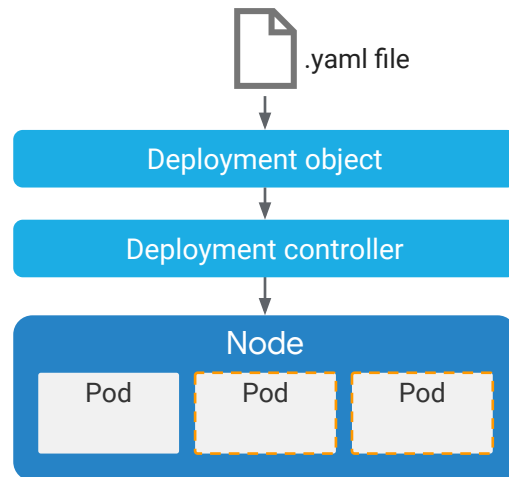
Every time you update the specification of the pods, for example, updating them to use a newer container image, a new ReplicaSet is created that matches the altered version of the Deployment. This is how deployments roll out updated Pods in a controlled manner: old Pods from the old ReplicaSet are replaced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



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

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



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

The Deployment named my-app is created with 3 replicated Pods.

In the spec.template section, a Pod template defines the metadata and specification of each of the Pods in this ReplicaSet.

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

Deployment has three different lifecycle states



Any Deployment has three different lifecycle states.

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

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

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

When you apply many small fixes across many rollouts, that translates to a large number of revisions, and to management complexity. You have to remember which small fix was applied with which rollout, which can make it challenging to figure out which revision to roll back to when issues arise. Remember, earlier 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
```



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.

←

Create a deployment

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

Deployment

Container

Container image

nginx:latest

Select Google Container Registry image

Environment variables

+ Add environment variable

Initial command (Optional)

Done

Cancel

+ Add container

3

Application name

nginx-1

Namespace

default

Labels

Key	Value
app	nginx-1

+ Add label

Cluster

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

The cluster will be named nginx-1-cluster

Zone

us-central1-a

Deploy

View YAML

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



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

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

- 'Desired' shows the desired number of replicas in the Deployment specification.
- 'Current' is the number of replicas currently running.
- 'Up-to-date' shows the number of replicas that are fully up to date as per the current Deployment specification.
- 'Available' displays the number of replicas available to the users.

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

Use the 'describe' command to get detailed info

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

```
master $ kubectl describe deployment nginx-deployment
Name:          nginx-deployment
Namespace:     default
CreationTimestamp:  Fri, 12 Oct 2018 15:23:46 +0000
Labels:        app=nginx
Annotations:   deployment.kubernetes.io/revision=1
Selector:      app=nginx
Replicas:      3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:  RollingUpdate
MinReadySeconds:  0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
    nginx:
      Image:   nginx:1.15.4
      Port:    80/TCP
      Host Port:  0/TCP
```



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

Or use the 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

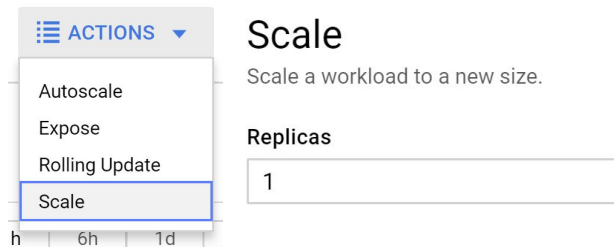
Revision	Name	Status	Restarts	Created on
1	nginx-1-7cb5b65464-2j97f	Running	0	Oct 12, 2018, 11:08:29 AM
1	nginx-1-7cb5b65464-gct8h	Running	0	Oct 12, 2018, 11:08:29 AM
1	nginx-1-7cb5b65464-tz85f	Running	0	Oct 12, 2018, 11:08:29 AM



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
```



The screenshot shows the 'Scale' section of the Google Cloud Console. On the left, there is a sidebar with a menu icon and the word 'ACTIONS' followed by a dropdown arrow. The dropdown menu is open, showing four options: 'Autoscale', 'Expose', 'Rolling Update', and 'Scale'. The 'Scale' option is highlighted with a blue border. Below the menu, there are time filters: 'h', '6h', and '1d'. To the right of the sidebar, the main area is titled 'Scale' in a large font. Below the title, there is a subtitle 'Scale a workload to a new size.' and a label 'Replicas' followed by a text input field containing the number '1'.



You now understand that a Deployment will maintain the desired number of replicas for an application. However, at some point you'll probably need to scale the Deployment. Maybe you need more web front end instances, for example. You can scale the Deployment manually using a `kubectl` command, or in the 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=1 --max=3  
--cpu-percent=80
```

Autoscale

Automatically scale the number of pods.

Minimum number of Pods (Optional)

Maximum number of Pods

Target CPU utilization in percent (Optional)

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



You can also autoscale the Deployment by specifying the minimum and maximum number of desired Pods along with a CPU utilization threshold. Again, you can perform autoscaling by using the `kubectl autoscale` command, or from the 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.

You can update a Deployment in different ways

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

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

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

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



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

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

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

Another way is to use a `kubectl 'edit'` command. This opens the specification file using the vim editor that allows you to make changes directly. Once you exit and save the file, `kubectl` automatically applies the updated file.

You can update a Deployment in different ways

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

Rolling update

Update workload Pods to a new application version.

Minimum seconds ready ⓘ (Optional)

Maximum surge ⓘ (Optional)

Maximum unavailable ⓘ (Optional)

Container name

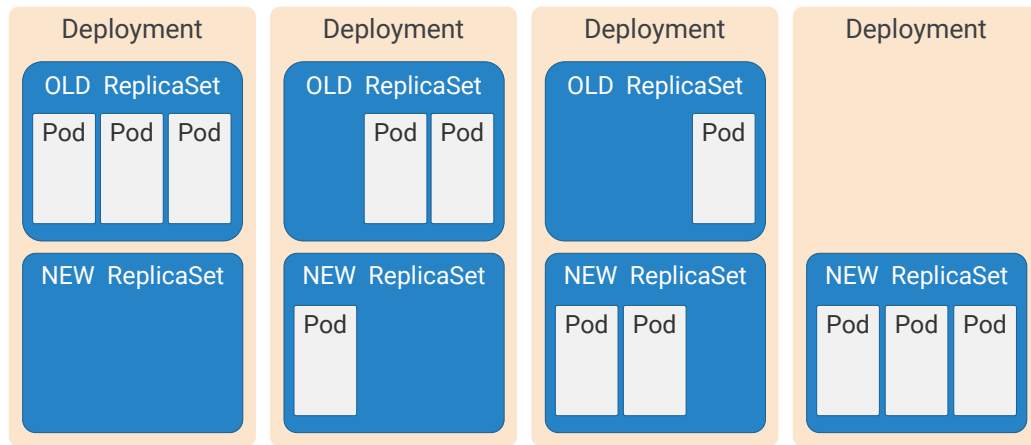
Image

nginx



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.

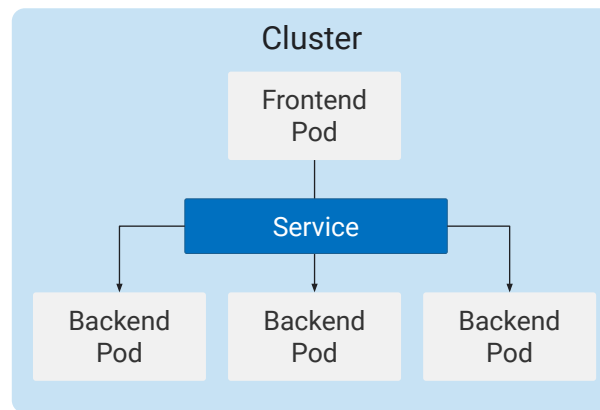
First, new Pods are launched in a new ReplicaSet.

Next, old Pods are deleted from the old ReplicaSet.

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

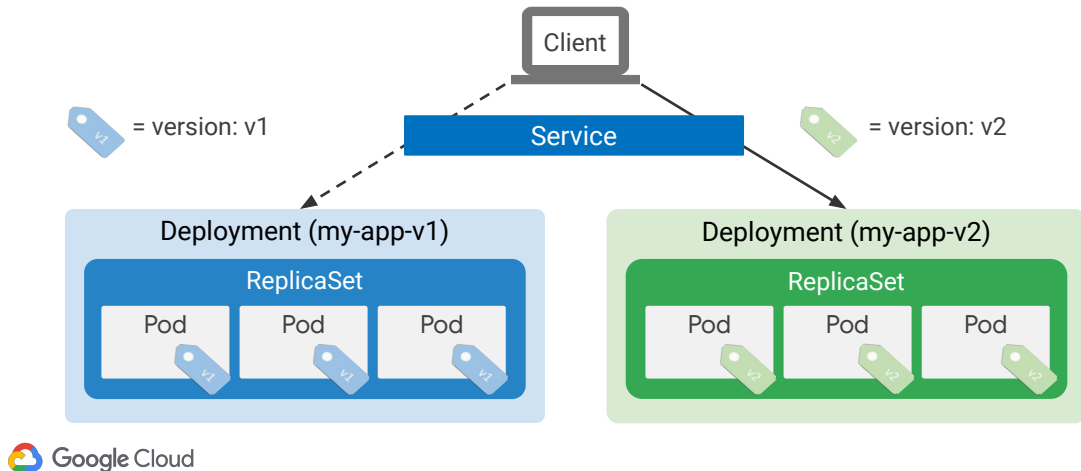
Its advantage is that updates are slowly released, which ensures the availability of the application. However, this process can take time, and there's no control over how the traffic is directed to the old and new Pods.

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.

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
[...]
```

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

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

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



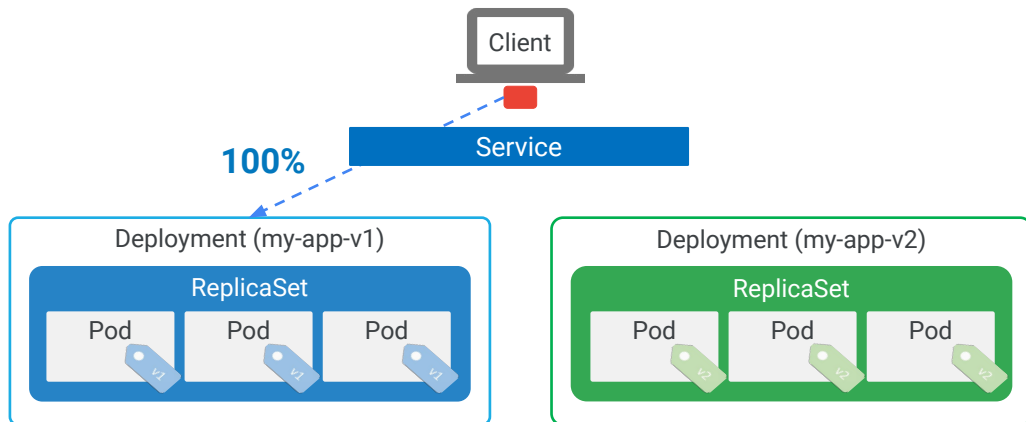
Here, in the Service definition, Pods are selected based on the label selector, where Pods in this example belong to my-app and to version v1.

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.

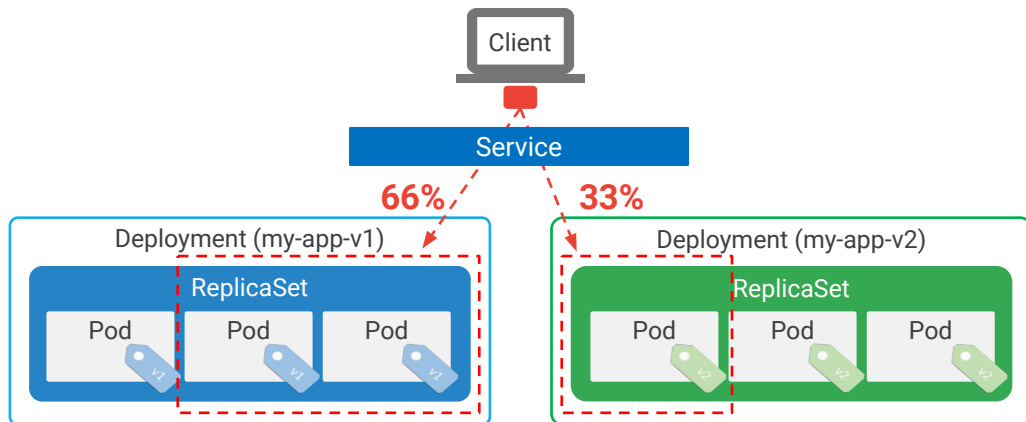
Canary deployment is an update strategy where traffic is gradually shifted to the new version



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.

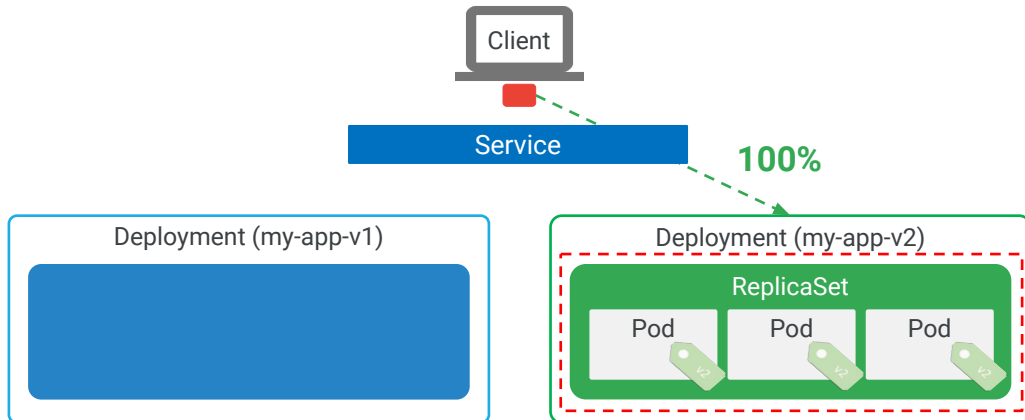
In this example, 100% of the application traffic is directed initially to my-app-v1 .

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



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.

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



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.

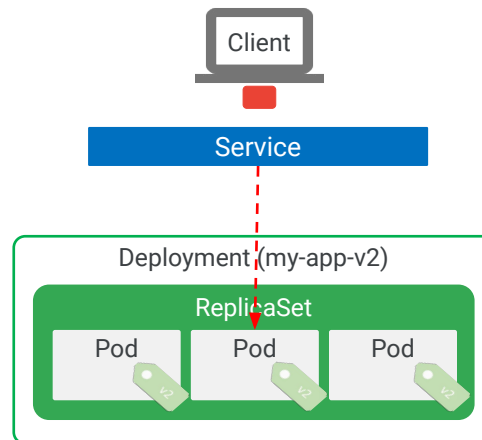
This setting allows the Service to select and direct the traffic to the Pods from both Deployments. Initially, the new version of the Deployment will start with zero replicas running. Over time, as the new version is scaled up, the old version of the Deployment can be scaled down and eventually deleted.

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`



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]
```

 REFRESH

 EDIT

 DELETE

 ACTIONS ▾

 KUBECTL ▾

Delete

Delete a resource

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

The operation cannot be reverted.

☒ Delete horizontal pod autoscaler nginx-1

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



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

Agenda

The kubectl command

Deployments

[Lab: Creating Google Kubernetes Engine Deployments](#)

Pod Networking

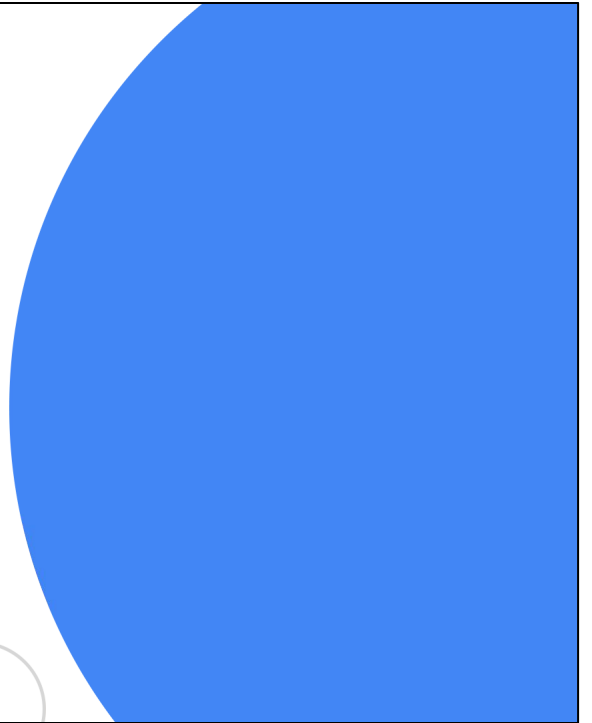
Volumes

Lab: Configuring Persistent Storage for Google Kubernetes Engine

Summary

Lab Intro

Creating Google Kubernetes
Engine Deployments



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

The first task that you'll learn to perform is to create a deployment manifest for a Pod inside the cluster. You'll then use both the 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

The kubectl command

Deployments

Lab: Creating Google Kubernetes
Engine Deployments

Pod Networking

Volumes

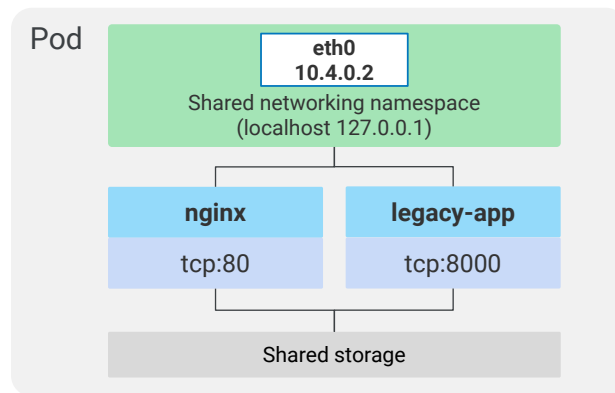
Lab: Configuring Persistent Storage
for Google Kubernetes Engine

Summary



The Kubernetes networking model relies heavily on IP addresses. Services, Pods, containers, and nodes communicate using IP addresses and ports.

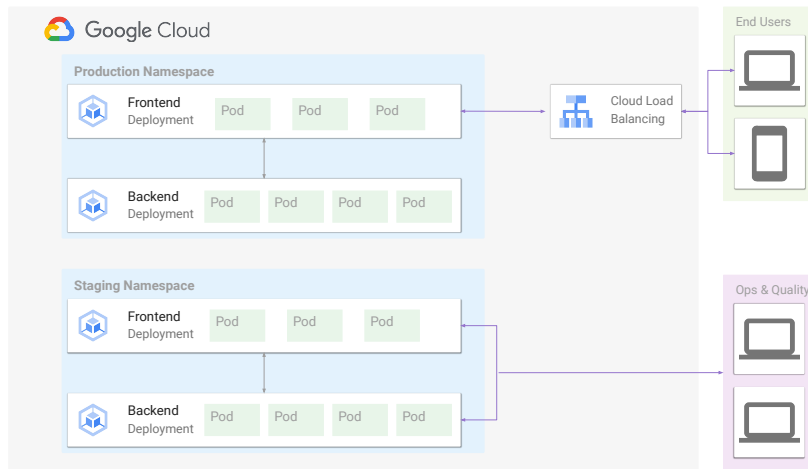
A Pod is a group of containers with shared storage and networking



Remember, a Pod is a group of containers with shared storage and networking. This is based on the “IP-per-pod” model of Kubernetes. With this model, each Pod is assigned a single IP address, and the containers within a Pod share the same network namespace, including that IP address.

For example, you might have a legacy application that uses nginx as a reverse-proxy for client access. The nginx container runs on TCP port 80, and the legacy application runs on TCP port 8000. Because both containers share the same networking namespace, the two containers appear as though they’re installed on the same machine. The nginx container will contact the legacy application by establishing a connection to “localhost” on TCP port 8000.

Your workload doesn't run in a single Pod



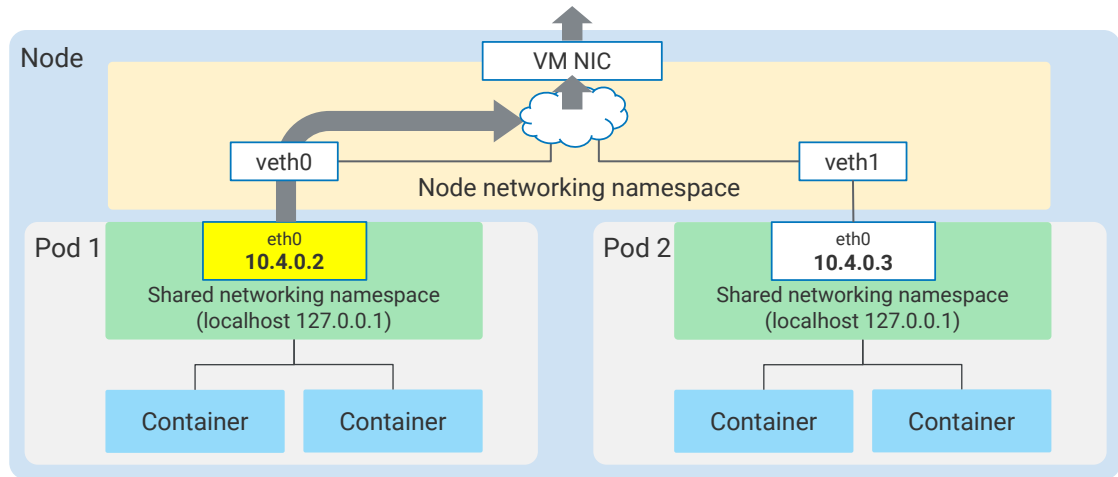
Google Cloud

This works well for a single Pod, but your workload doesn't run in a single Pod.

Your workload is composed of many different applications that need to talk to each other.

So how do Pods talk to other Pods?

Pod-to-Pod communication on the same node



Each Pod has a unique IP address, just like a host on the network.

On a node, the Pods are connected to each other through the node's root network namespace, which ensures that Pods can find and reach each other on that VM.

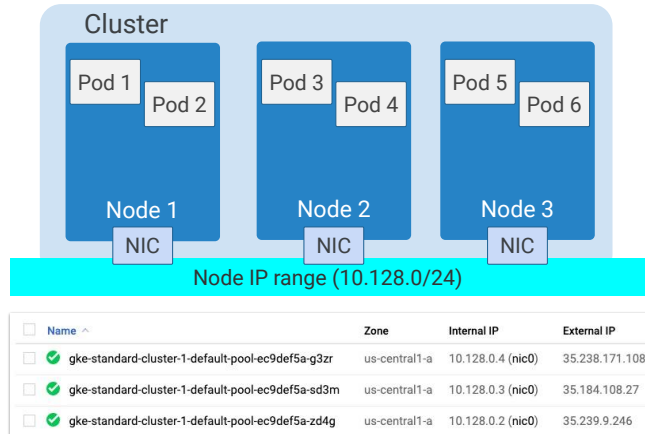
This allows the two Pods to communicate on the same node.

The root network namespace is connected to the Node's primary NIC.

Using the node's VM NIC, the root network namespace is able to forward traffic out of the node.

This means that the IP addresses on the Pods must be routable on the network that the node is connected to.

Nodes get Pod IP addresses from address ranges assigned to your Virtual Private Cloud

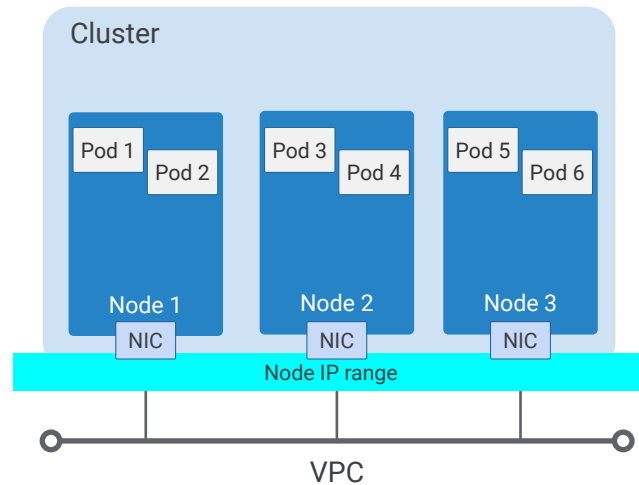


In GKE, the nodes will get the Pod IP addresses from address ranges assigned to your Virtual Private Cloud, or VPC.

VPCs are logically isolated networks that provide connectivity for resources you deploy within Google Cloud, such as Kubernetes Clusters, Compute Engine instances, and App Engine Flex instances. A VPC can be composed of many different IP subnets in regions all around the world.

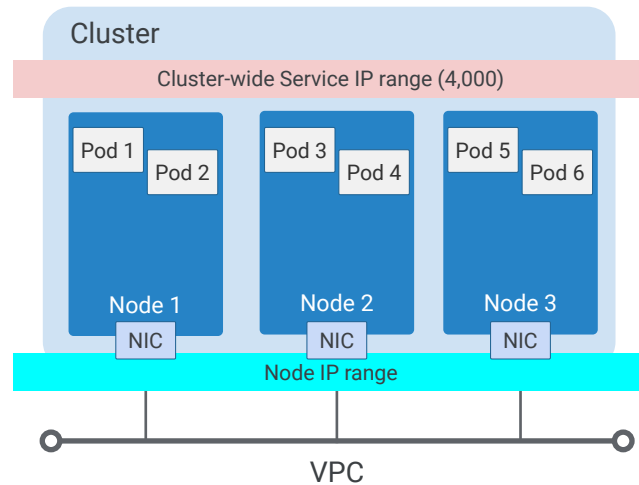
When you deploy GKE, you can select a VPC along with a region or zone. By default, a VPC has an IP subnet pre-allocated for each Google Cloud region in the world. The IP addresses in this subnet are then allocated to the compute instances that you deploy in the region.

Addressing the Pods



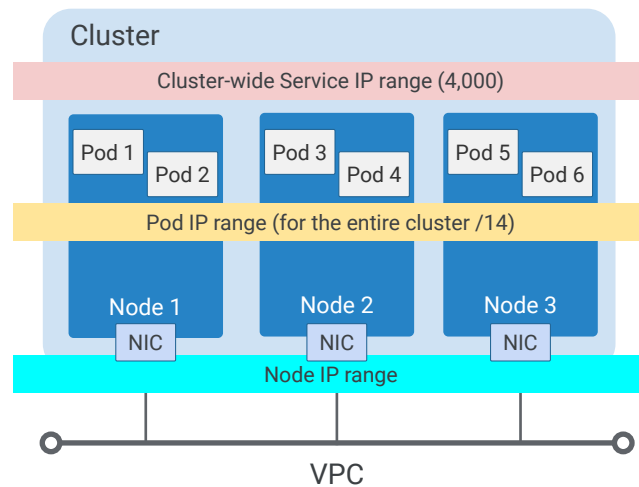
GKE cluster nodes are compute instances that GKE customizes and manages for you. These machines are assigned IP addresses from the VPC subnet that they reside in.

Addressing the Pods



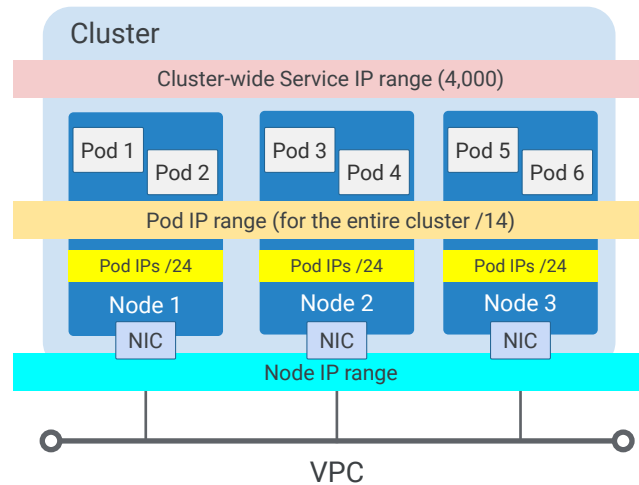
On Google Cloud, Alias IPs allow you to configure additional secondary IP addresses or IP ranges on your Compute Engine VM instances. VPC-Native GKE clusters automatically create an Alias IP range to reserve approximately 4,000 IP addresses for cluster-wide Services that you may create later. This mitigates the problem of unexpectedly running out of service IP addresses.

Addressing the Pods



VPC-Native GKE clusters also create a separate Alias IP range for your Pods. Remember, each Pod must have a unique address, so this address space will be large. By default the address range uses a /14 block, which contains over 250,000 IP addresses. That's a lot of Pods.

Addressing the Pods



In reality, Google doesn't expect you to run 250,000 Pods in a single cluster. Instead, that massive IP address range allows GKE to divide the IP space among the nodes. Using this large Pod IP range, GKE allocates a much smaller /24 block to each node, which contains about 250 IP addresses. This allows for 1000 nodes, with over running 100 pods each, by default.

The number of nodes you expect to use and the maximum number of pods per node are configurable, so you don't have to reserve a whole /14 for this.

Agenda

The kubectl command

Deployments

Lab: Creating Google Kubernetes
Engine Deployments

Pod Networking

Volumes

Lab: Configuring Persistent Storage
for Google Kubernetes Engine

Summary



Let's look at storage and the idea of Volumes. In this class you'll learn about the types of storage abstractions that Kubernetes provides, such as Volumes and PersistentVolumes. You'll learn about how these differ, and how they can be used to store and share information between Pods.

Kubernetes offers storage abstraction options

Volumes

Volumes are the method by which you attach storage to a Pod.

Some Volumes are ephemeral.

Some Volumes are persistent.

Persistent storage options

Are block storage, or networked file systems.

Provide durable storage outside a Pod.

Are independent of the Pod's lifecycle.

May exist before Pod creation and can be claimed.



Remember that Kubernetes uses objects to represent the resources it manages. This rule applies to storage as well as to Pods. All these objects function as useful abstractions, which means that you can manage the resources they represent without laborious attention to implementation details. Kubernetes provides storage abstractions as Volumes and PersistentVolumes.

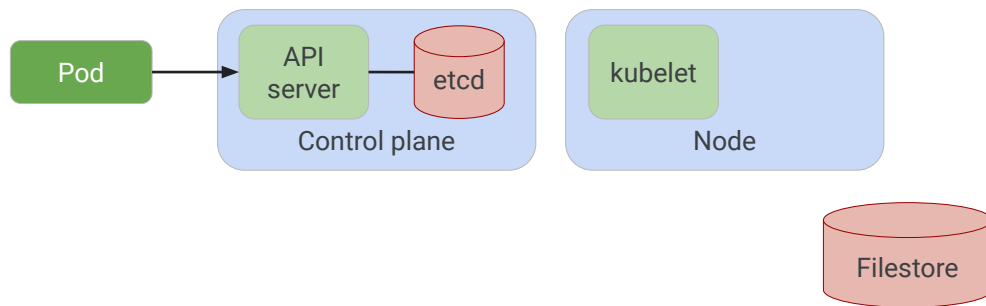
Volumes are the method by which you attach storage to a Pod. Some Volumes are ephemeral, which means they last only as long as the Pod to which they are attached. You will see examples of these types in this lesson, such as ConfigMap and emptyDir. And some Volumes are persistent, which means that they can outlive a Pod. Regardless of type, Volumes are attached to Pods, not containers. If a Pod isn't mapped to a node any more, the Volume isn't either.

Other Volume types can be used to provide storage that is persistent. This can be used for data that must outlive the lifecycle of an individual Pod. In Kubernetes clusters, you will frequently find these Volume types backed by NFS volumes, or Windows shares, or persistent disks from the underlying cloud provider. These types of Volumes embody block storage, or use networked file systems. On GKE, these Volumes are typically backed by Compute Engine Persistent Disks. They provide durable storage beyond the existence of a Pod. A failing node or Pod shouldn't affect these Volumes. If that happens, the volumes are simply unmounted from the failing Pod. Some of these Volumes might already exist before the creation of the Pod and

can be claimed and mounted.

A Volume is a directory that's accessible to the containers in a Pod

```
$> kubectl apply -f pod.yaml
```



At its core, a Volume is just a directory that is accessible to the containers in a Pod. How that directory is created, the medium that backs it, and its contents are determined by the particular Volume type used.

In this case, we'll create a Pod with an NFS Volume. The NFS server that backs this could be anywhere. In Google Cloud, the lowest-overhead way to serve NFS volumes is the Filestore managed service.

In this example, we'll create a Pod using the `kubectl apply` command using a Pod manifest that includes an NFS Volume type.

Creating a Pod with an NFS Volume (1/3)

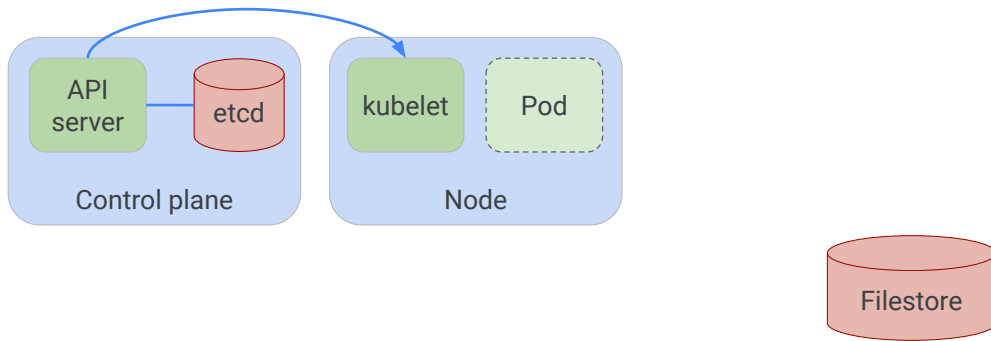
```
apiVersion: v1
kind: Pod
metadata:
  name: web
spec:
  containers:
    - name: web
      image: nginx
      volumeMounts:
        - mountPath: /mnt/vol
          name: nfs
  volumes:
    - name: nfs
      server: 10.1.1.2
      path: "/"
      readOnly: false
```



Here, we're creating a Pod with an NFS Volume. A Volume section is added under the Pod's spec. The Volume is named and fields are configured with the access details for an existing NFS share. This creates an NFS Volume when the Pod is created.

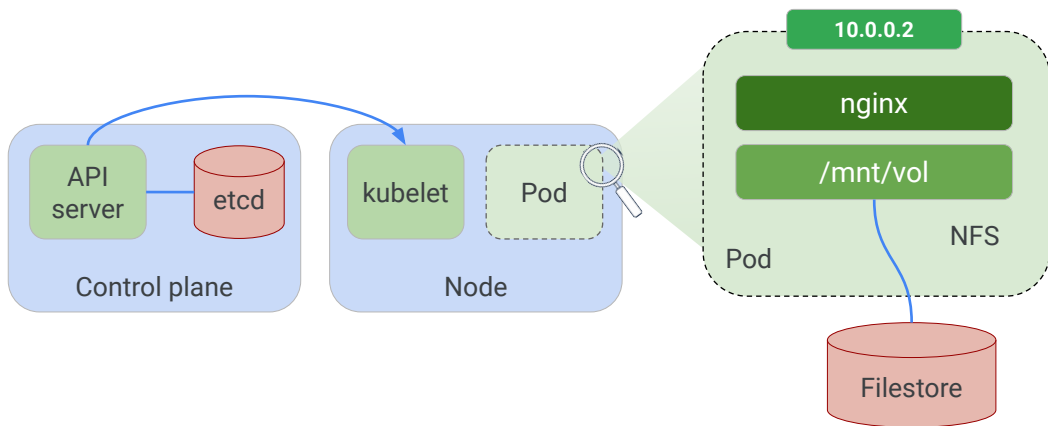
In order to mount the Volume to a container, the Volume name and mount path must be specified under the volumeMounts field for the container. The container will only start when all the volumes are ready to be mounted.

Creating a Pod with an NFS Volume (2/3)



The Volume is created during Pod creation. After the Volume is created, it's available to any container in the Pod before the containers are brought online.

Creating a Pod with an NFS Volume (3/3)

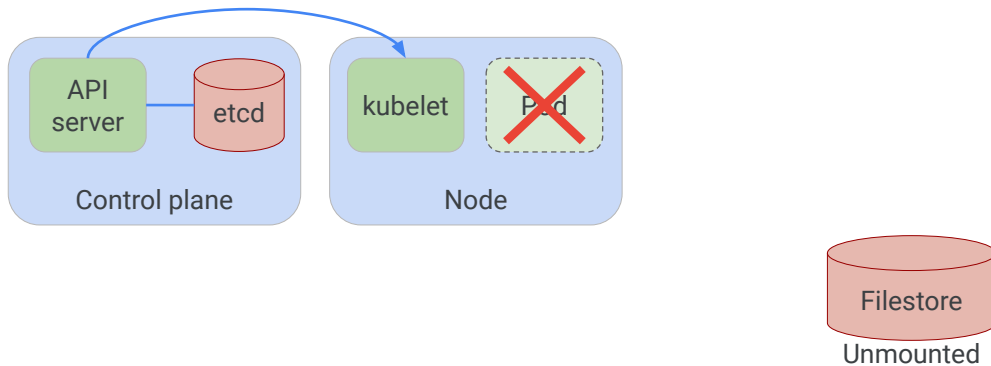


The Volume is created during Pod creation. After the Volume is created, it's available to any container in the Pod before the containers are brought online. After the Volume is attached to a container, the data in the Volume is mounted into the container's file system. In this example, the files stored on the NFS share are made available on the `/mnt/vol` directory inside the container.

With the data mounted, the containers in the Pod are brought online and the rest of the Pod initialization happens as before.

In this example, you have an NGINX Pod with an NFS Volume attached. The Volume is attached at the mount Volume directory when the Pod is run, and then the NGINX container can see that `/mnt/vol` directory and get data from it. For example, maybe the content that NGINX serves is located in that share.

Data saved on NFS Volumes will outlive the Pod



Unlike the contents of emptyDir Volumes, which are erased when a Pod is deleted, the data saved on NFS Volumes will outlive the Pod. When the Pod is deleted, the NFS Volume will be removed. However, the data isn't erased; it's just unmounted and can be remounted on new Pods if needed.

The benefits of PersistentVolumes

Abstracts storage provisioning from storage consumption.

Promotes microservices architecture.

Allows cluster administrators to provision and maintain storage.

Developers can claim provisioned storage for app consumption.



Kubernetes PersistentVolume objects abstract storage provisioning from storage consumption.

Remember, Kubernetes enables microservices architecture where an application is decoupled into components that can be scaled easily. Persistent storage makes it possible to deal with failures and allow for dynamic rescheduling of components without loss of data. However, should application developers be responsible for creating and maintaining separate Volumes for their application components? Also, how can developers test applications before deploying into production without modifying the Pod manifests for their applications? Whenever you have to reconfigure things to go from test to production, there's a risk of error. Kubernetes' PersistentVolume abstraction resolves both of these issues.

Using PersistentVolumes, a cluster administrator can provision a variety of Volume types. The cluster administrator can simply provision storage and not worry about its consumption.

And application developers can easily claim and use provisioned storage using PersistentVolumeClaims without creating and maintaining storage volumes directly. Notice the separation of roles? It's the job of administrators to make persistent volumes available, and the job of developers to use those volumes in applications.

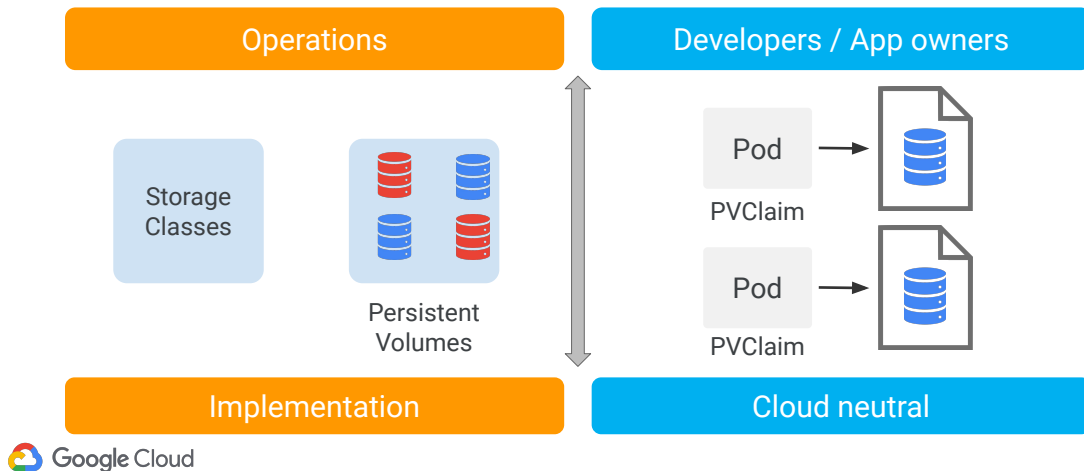
The two job roles can work independently of each other.

When application developers use PersistentVolumeClaims, they don't need to worry about where the application is running: provisioned storage capacity can be claimed by the application regardless of whether it is running on a local site, Google Cloud, or any other cloud provider.

Let's look at what is required for an application owner to consume Compute Engine persistent disk storage, first using Pod-level Volumes and then using Cluster-level PersistentVolumes and PersistentVolumeClaims in Pod manifests. You will see that the second way is more manageable.

In GKE the default StorageClass is configured to dynamically provision gcePersistentDisk based PersistentVolumes by default so PersistentVolumeClaims can be used for standard persistent volumes without any additional preparation.

PersistentVolumeClaims and PersistentVolumes separate storage consumption from provisioning



In order to use PersistentVolumes the operations team that owns the specific cloud implementation define the storage classes and manage the actual implementation details of the Persistent Volumes.

The developers and application owners use PersistentVolumeClaims to request the quantity of storage and storage class, which determines the type of storage.

This allows the operations team to manage the cloud services they wish to use and allows the application owners to focus on what their application requires rather than the specific implementation detail.

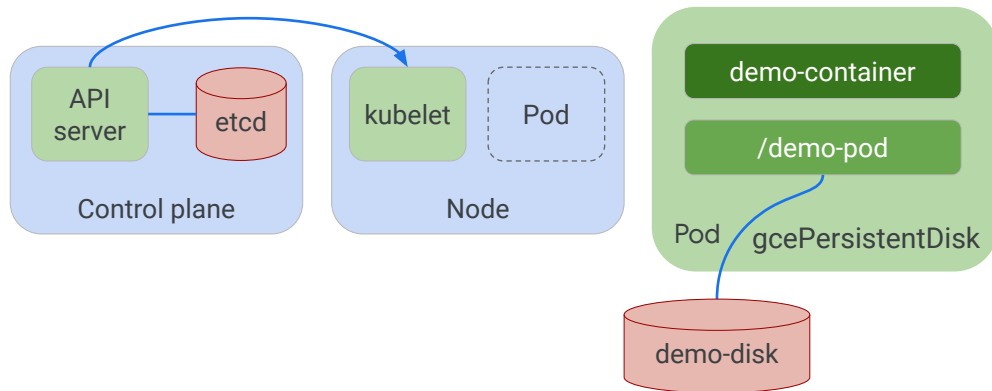
Creating a Compute Engine persistent disk using a gcloud command

```
$ gcloud compute disks create  
--size=100GB  
--zone=us-central1-a demo-disk
```



Google Cloud's Compute Engine service uses Persistent Disks for virtual machines' disks, and Kubernetes Engine uses the same technology for PersistentVolumes. Persistent Disks are network-based block storage that can provide durable storage. First a 100-GB Compute Engine Persistent Disk is created using a gcloud command. Before this Persistent Disk can be used by any Pod, someone or some process must create it, and that person or process must have Compute Engine administration rights.

Creating a Compute Engine Persistent Disk



When the Pod is created, Kubernetes uses the Compute Engine API to attach the Persistent Disk to the node on which the Pod is running. The Volume is automatically formatted and mounted to the container. If this Pod is moved to another node, Kubernetes automatically detaches the Persistent Disk from the existing node and re-attaches it to the newer node.

Configuring Volumes in Pods makes portability difficult

```
Volumes:
  pd-volume:
    Type:          GCEPersistentDisk
    PDName:        demo-disk
    FSType:        ext4
    Partition:     0
```

```
Volumes:
  pd-volume:
    Type:          vsphereVolume
    PDName:        demo-disk
    FSType:        ext4
    Partition:     0
```



You can confirm that a Volume was mounted successfully using the `kubectl describe Pod` command. Don't forget that a Persistent Disk must be created before it can be used. The Persistent Disk can have pre-populated data that can be shared by the containers within a Pod. That's very convenient. However, the application owner must include the specific details of the Persistent Disk inside the Pod manifest for their application and must confirm that the Persistent Disk already exists. That's not very convenient.

Hard coding Volume configuration information into Pod specifications in this way means you may have difficulty porting data from one cloud to another. On GKE, Volumes are usually configured to use Compute Engine Persistent Disks. In your on-premises Kubernetes cluster, they might be VMware vSphere volume files, for example, or even physical hard drives.

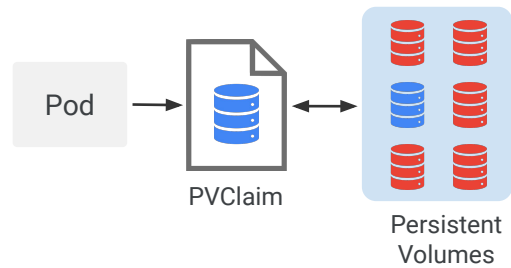
Whenever you need to reconfigure an application to move it from one environment to another, there's a risk of error. To address this problem, Kubernetes provides an abstraction called Persistent Volumes. This abstraction lets a Pod claim a Volume of a certain size, or of a certain name, from a pool of storage without forcing you to define the storage type details inside the Pod specification.

PersistentVolumes abstraction has two components

PersistentVolume (PV)

- Independent of a Pod's lifecycle.
- Managed by Kubernetes.
- Manually or dynamically provisioned.
- Persistent Disks are used by GKE as PersistentVolumes.

PersistentVolumeClaim (PVC)



Let's take a closer look at how PersistentVolumes make the use of network storage like this more manageable. The PersistentVolume abstraction has two components: PersistentVolume (PV) and PersistentVolumeClaim (PVC).

PersistentVolumes are durable and persistent storage resources managed at the cluster level. Although these cluster resources are independent of the Pod's lifecycle, a Pod can use these resources during its lifecycle. However, if a Pod is deleted, a PersistentVolume and its data continue to exist.

These Volumes are managed by Kubernetes and can be manually or dynamically provisioned.

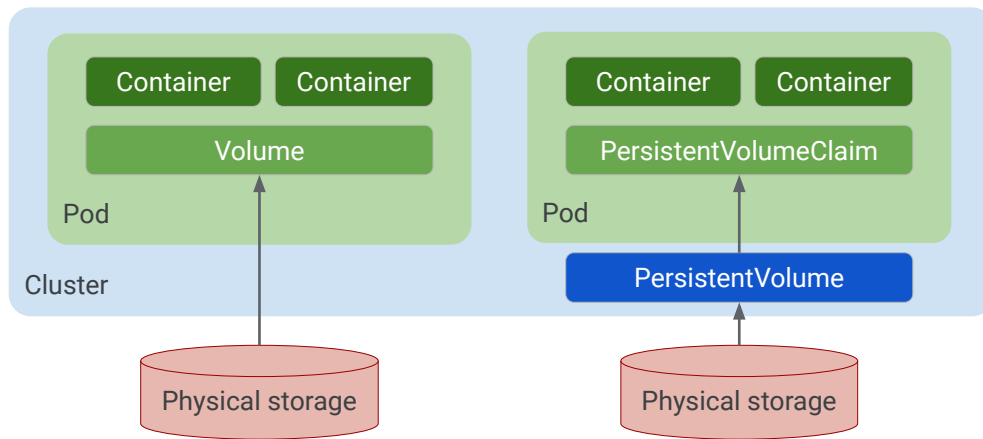
GKE can use Compute Engine Persistent Disks as PersistentVolumes.

PersistentVolumeClaims are requests and claims made by Pods to use PersistentVolumes. Within a PersistentVolumeClaim object, you define a Volume size, access mode, and StorageClass. What's a StorageClass? It's a set of storage characteristics that you've given a name to.

A Pod uses this PersistentVolumeClaim to request a PersistentVolume. If a PersistentVolume matches all the requirements defined in a PersistentVolumeClaim,

the PersistentVolumeClaim is bound to that PersistentVolume. Now, a Pod can consume storage from this PersistentVolume.

PersistentVolumes must be claimed



What's the critical difference between using Pod-level Volumes and Cluster-level PersistentVolumes for storage? PersistentVolumes provide a level of abstraction that lets you decouple storage administration from application configuration. The storage in a PersistentVolume must be bound with a PersistentVolumeClaim in order to be accessed by a Pod.

Creating a PersistentVolume manifest

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pd-volume
spec:
  storageClassName: "standard"
  capacity:
    storage: 100G
  accessModes:
    - ReadWriteOnce:
  gcePersistentDisk:
    pdName: demo-disk
    fsType: ext4
```

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: standard
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  replication-type: none
```

PVC StorageClassName
must match the
PV StorageClassName



Here's how you create a PersistentVolume manifest for the same storage. Let's take a closer look at how this is used to make managing storage configuration for Pods easier.

First, you specify the Volume capacity.

Then the storageClassName. StorageClass is a resource used to implement PersistentVolumes. Note that the PVC uses the StorageClassName when you define the PVC in a Pod, and it must match the PV StorageClassName for the claim to succeed.

GKE has a default StorageClass named 'standard' to use the Compute Engine Standard Persistent Disk type, as shown here on the right. In this example, the PV definition on the left matches the GKE default StorageClass. In GKE clusters, a PVC with no defined StorageClass will use this default StorageClass and provide storage using a standard Persistent Disk.

Create a new StorageClass to use an SSD Persistent Disk

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pd-volume
spec:
  storageClassName: "ssd"
  capacity:
    storage: 100G
  accessModes:
    - ReadWriteOnce:
  gcePersistentDisk:
    pdName: demo-disk
    fsType: ext4
```

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: ssd
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```



If you want to use an SSD Persistent Disk, you can create a new StorageClass, such as this example named `ssd`. A PVC that uses this new StorageClass named `ssd` will only use a PV that also has a StorageClass named `ssd`. In this instance, an SSD Persistent Disk will be used.

The modern, easier-to-manage way is to use the PersistentVolume abstraction

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
spec:
  containers:
    - name: demo-container
      image: gcr.io/hello-app:1.0
      volumeMounts:
        - mountPath: /demo-pod
          name: pd-volume
  volumes:
    - name: pd-volume
      PersistentVolumeClaim:
        claimName: pd-volume-claim
```

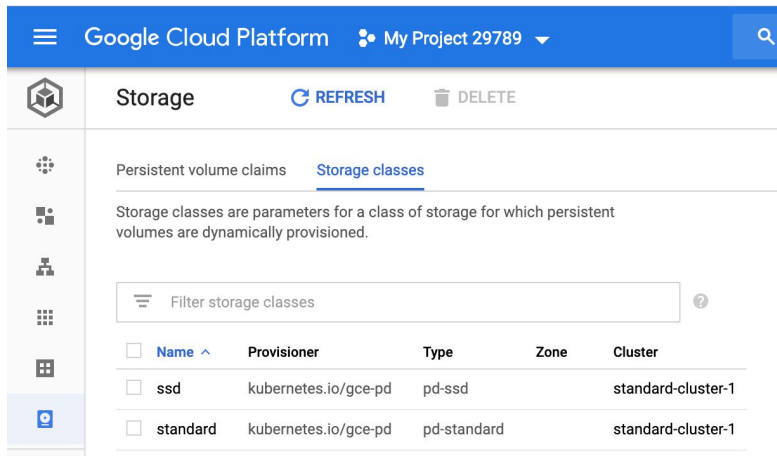
```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pd-volume-claim
spec:
  storageClassName: "standard"
  accessModes:
    - ReadWriteOnce:
  resources:
    requests:
      storage: 100G
```



Add a PersistentVolumeClaim to the Pod, as shown here on the left. In our example, the PersistentVolumeClaim named 'pd-volume-claim' has the 'standard' storageClassName, the 'ReadWriteOnce' accessMode, and a requested capacity of 100 gigabytes. When this Pod is started, GKE will look for a matching PV with the same storageClassName and accessModes and sufficient capacity. The specific cloud implementation doesn't really matter, the specific storage used to deliver this storage class is something the cluster administrators, not the application developers, control.

What could go wrong with this? Well, what if application developers claim more storage than has already been allocated to PersistentVolumes? PersistentVolumes are managed by cluster administrators, but application developers make the PersistentVolumeClaims, and this could lead to storage allocation failures.

Viewing the new storage class in the Cloud Console



The screenshot shows the Google Cloud Platform Storage console. The top bar displays 'Google Cloud Platform' and 'My Project 29789'. The left sidebar contains navigation icons. The main content area is titled 'Storage' and includes a 'REFRESH' button and a 'DELETE' button. Below this, there are tabs for 'Persistent volume claims' and 'Storage classes'. The 'Storage classes' tab is active, showing a description: 'Storage classes are parameters for a class of storage for which persistent volumes are dynamically provisioned.' Below the description is a search bar labeled 'Filter storage classes'. A table lists the storage classes:

<input type="checkbox"/>	Name ^	Provisioner	Type	Zone	Cluster
<input type="checkbox"/>	ssd	kubernetes.io/gce-pd	pd-ssd		standard-cluster-1
<input type="checkbox"/>	standard	kubernetes.io/gce-pd	pd-standard		standard-cluster-1



Once you create the new storage class with a `kubectl apply` command, you can view it in the Cloud Console.

By the way, don't confuse Kubernetes StorageClasses with Storage Classes that Google Cloud Storage makes available. Although the features have the same name, they are unrelated, because they come from different services and govern different features. Google Cloud Storage is object storage for the web, while Kubernetes StorageClasses are choices for how PersistentVolumes are backed.

Agenda

The kubectl command

Deployments

Lab: Creating Google Kubernetes
Engine Deployments

Pod Networking

Volumes

[Lab: Configuring Persistent Storage
for Google Kubernetes Engine](#)

Summary

Lab Intro

Configuring Persistent Storage for
Google Kubernetes Engine



In this lab, you'll set up `PersistentVolumes` and `PersistentVolumeClaims`. `PersistentVolumes` are storage that is available to a Kubernetes cluster. `PersistentVolumeClaims` enable Pods to access `PersistentVolumes`. Without `PersistentVolumeClaims`, Pods are mostly ephemeral, so you should use `PersistentVolumeClaims` for any data that you expect to survive Pod scaling, updating, or migrating.

The tasks that you'll perform include creating manifests for PVs and PVCs for Compute Engine persistent disks, mounting Compute Engine persistent disk PVCs as volumes in Pods, and using manifests to create `StatefulSets`. You'll also mount Compute Engine persistent disk PVCs as Volumes in `StatefulSets` and verify the connection of pods in `StatefulSets` to particular PVs as the Pods are stopped and restarted.

Agenda

The kubectl command

Deployments

Lab: Creating Google Kubernetes
Engine Deployments

Pod Networking

Volumes

Lab: Configuring Persistent Storage
for Google Kubernetes Engine

[Summary](#)

Summary

Work with the kubectl command.

Understand how Deployments are used in Kubernetes.

Understand the networking architecture of Pods.

Understand Kubernetes storage abstractions.



That concludes the introduction to Kubernetes Workloads. In this module you learned how to understand and work with Kubernetes using the kubectl command. You looked at how to create and use Deployments. You learned about Pod networking and how to create Services to expose applications running within Pods, allowing them to communicate with each other, and the outside world. Lastly you learnt about storage abstractions which will allow you configure and select the appropriate storage type for your applications.

