



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.

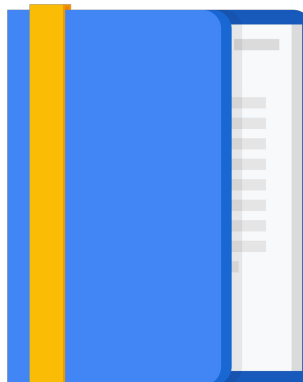
Learn how to ...

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

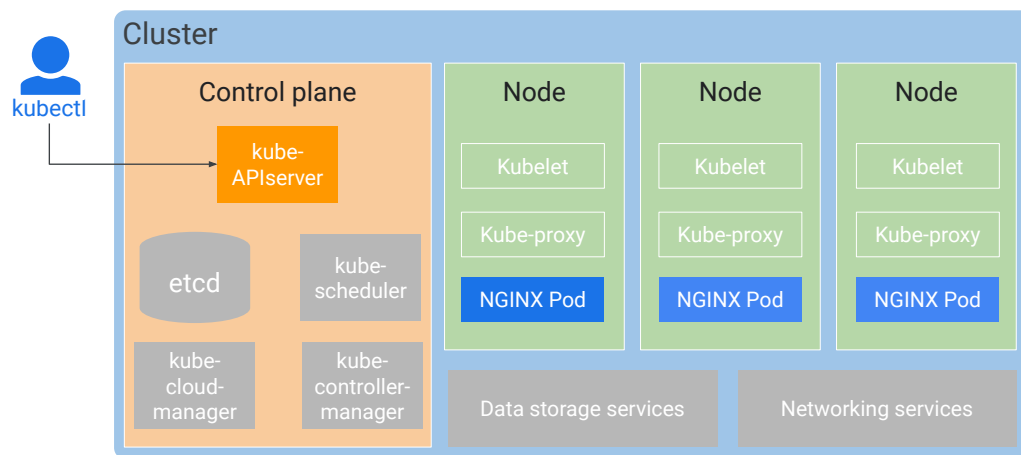
Quiz

Summary

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

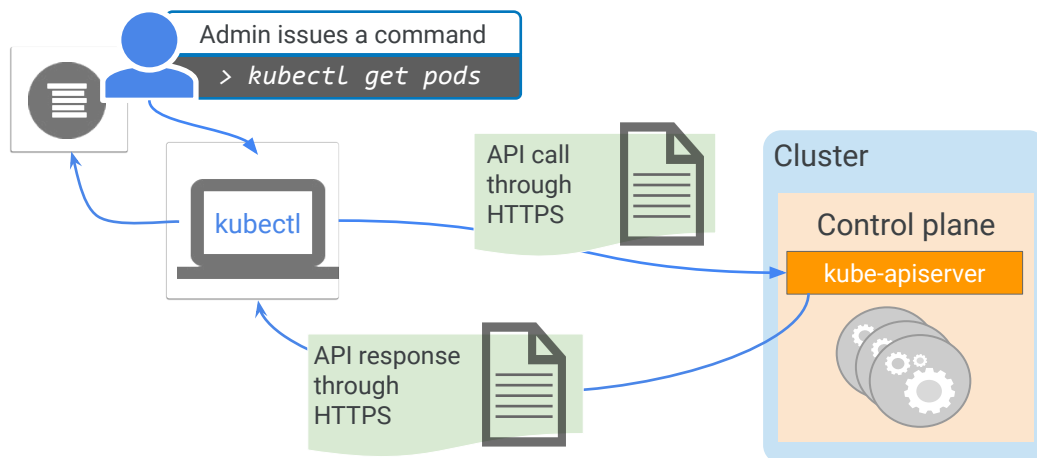
`Kubectl` is a utility used by administrators to control Kubernetes clusters. You use it to communicate with the Kube API server on your control plane.

Kubectl transforms your command-line entries into API calls



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]
```

Google Cloud

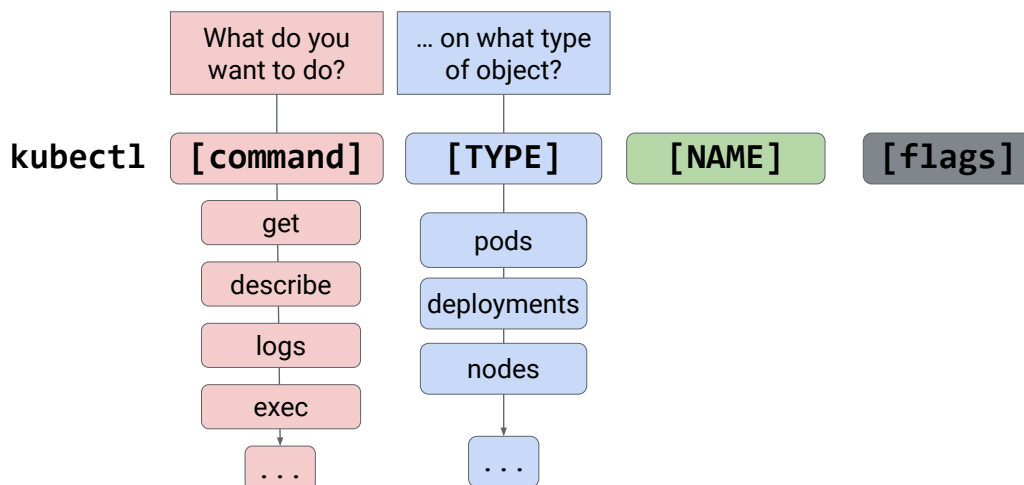
To connect to a GKE cluster with `kubectl`, retrieve your credentials for the specified cluster first. To do this, use the `get-credentials` `gcloud` command in any other environment where you've installed the `gcloud` command-line tool and `kubectl`. Both of these tools are installed by default in the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



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

Quiz

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

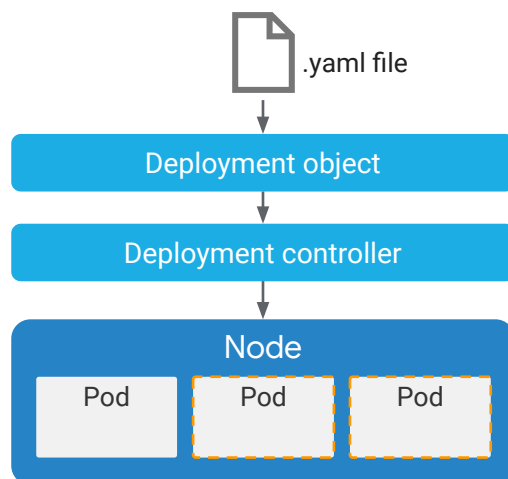
Every time you update the specification of the pods, for example, updating them to use a newer container image, a new ReplicaSet is created that matches the altered version of the Deployment. This is how deployments roll out updated Pods in a controlled manner: old Pods from the old ReplicaSet are replaced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

There are three ways to create a Deployment

The image displays three sequential screenshots of the Google Cloud 'Create a deployment' wizard, illustrating the steps to create a deployment.

Step 1: Container

- Edit container**
 - ☒ Existing container image
 - ☐ New container image
- Image path ***
 - nginx:latest
 - [SELECT](#)
- Environment variables**
 - [+ ADD ENVIRONMENT VARIABLE](#)
 - Initial command**
 - Overrides the default endpoint of the container image.
- [CANCEL](#) [DONE](#)
- [ADD CONTAINER](#)
- [CONTINUE](#)

Step 2: Configuration

- Application name ***
 - nginx-1
- Namespace ***
 - default
- Labels**

| Key * | Value |
|-------|---------|
| app | nginx-1 |
- [+ ADD KUBERNETES LABEL](#)

Step 3: Configuration YAML

- Configuration YAML**
 - Kubernetes deployments are defined declaratively using YAML files. The best practice is to store these files in version control, so you can track changes to your deployment configuration over time.
 - [VIEW YAML](#)
- Cluster**
 - Kubernetes Cluster
 - standard-cluster-1 (us-central1-a)
 - Cluster in which the deployment will be created.
 - [CREATE NEW CLUSTER](#)
- [DEPLOY](#)

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.

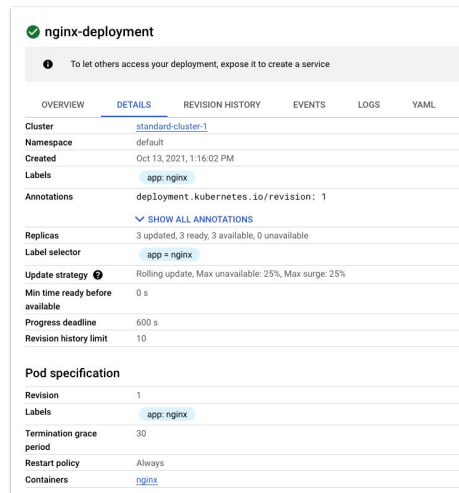
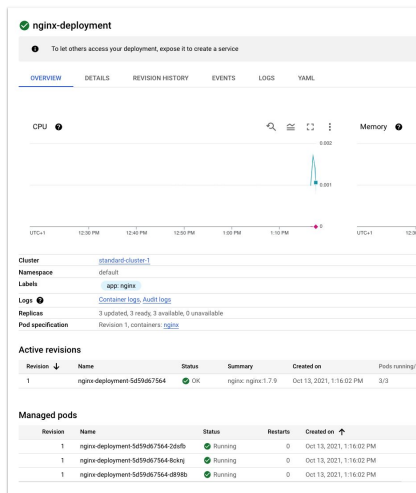
Use the 'describe' command to get detailed info

```
$ 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



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 ▾ KUBE

- Autoscale
- Expose
- Rolling update
- Scale**
- Automated deployment

Edit default-pool

Node version
1.20.10-gke.301
[CHANGE](#)

Size

Number of nodes *
2

☐ Enable autoscaling ⓘ

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](#)

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.

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

0

Maximum surge

25%

Maximum unavailable

25%

Container images

Image of nginx *

nginx:1.7.9

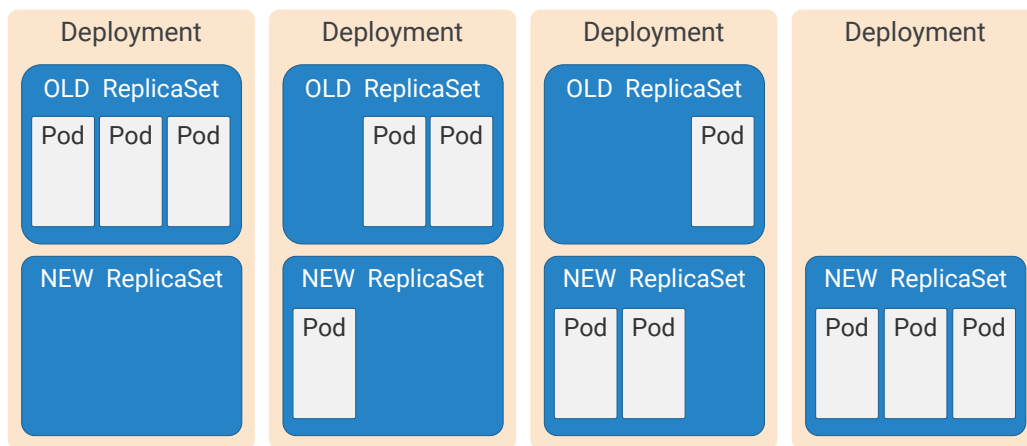
* Indicates required field

CANCEL

UPDATE

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.

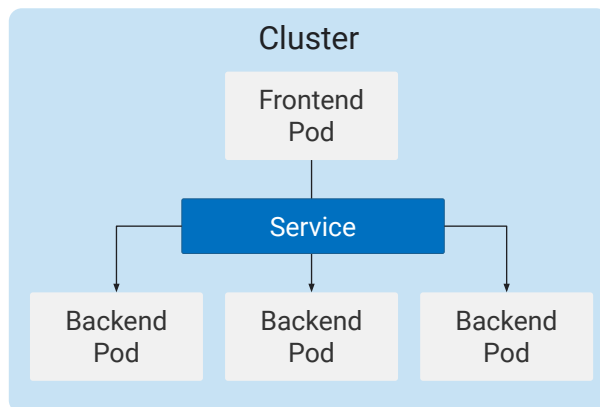
First, new Pods are launched in a new ReplicaSet.

Next, old Pods are deleted from the old ReplicaSet.

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

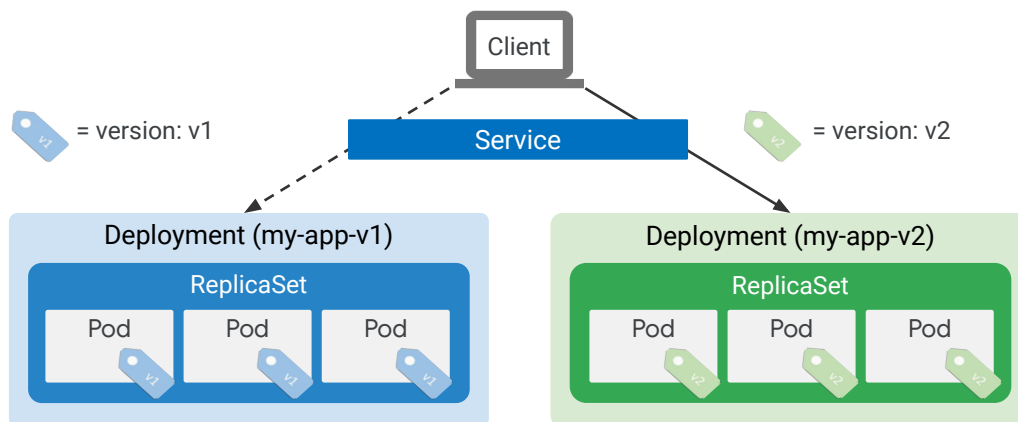
Its advantage is that updates are slowly released, which ensures the availability of the application. However, this process can take time, and there's no control over how the traffic is directed to the old and new Pods.

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



Google Cloud

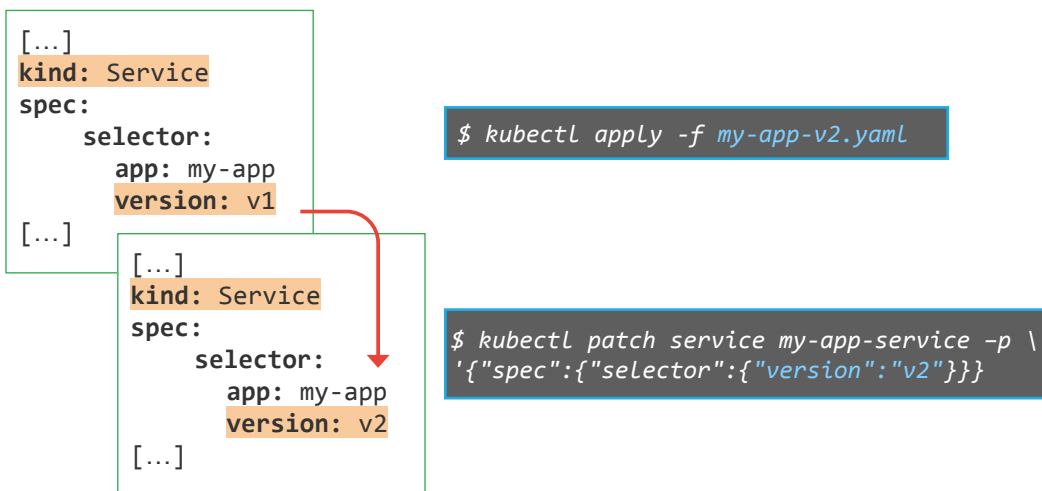
A blue/green deployment strategy is useful when you want to deploy a new version of an application and also ensure that application services remain available while the Deployment is updated.

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

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

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

Applying a blue/green deployment strategy



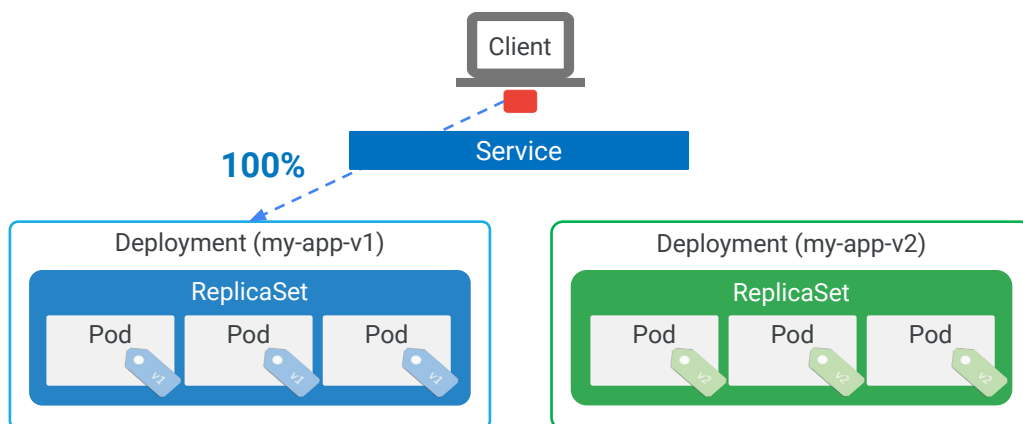
Here, in the Service definition, Pods are selected based on the label selector, where Pods in this example belong to my-app and to version v1.

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

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

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

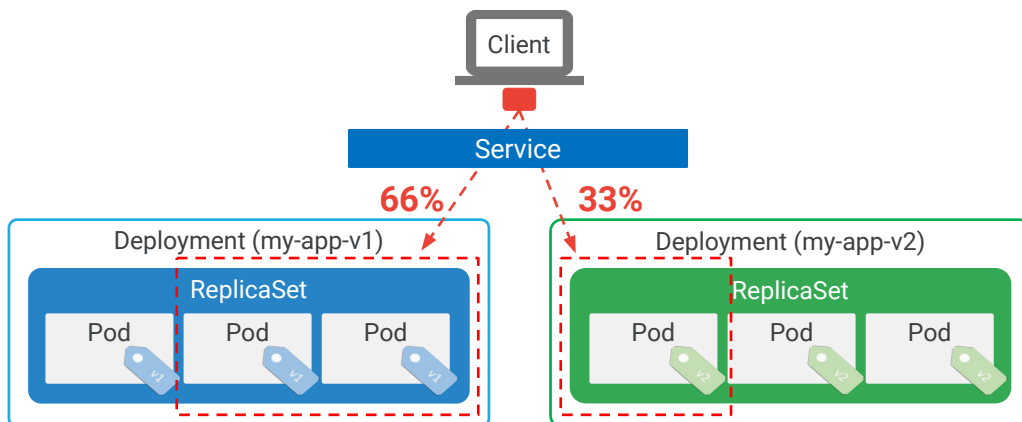
Canary deployment is an update strategy where traffic is gradually shifted to the new version



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

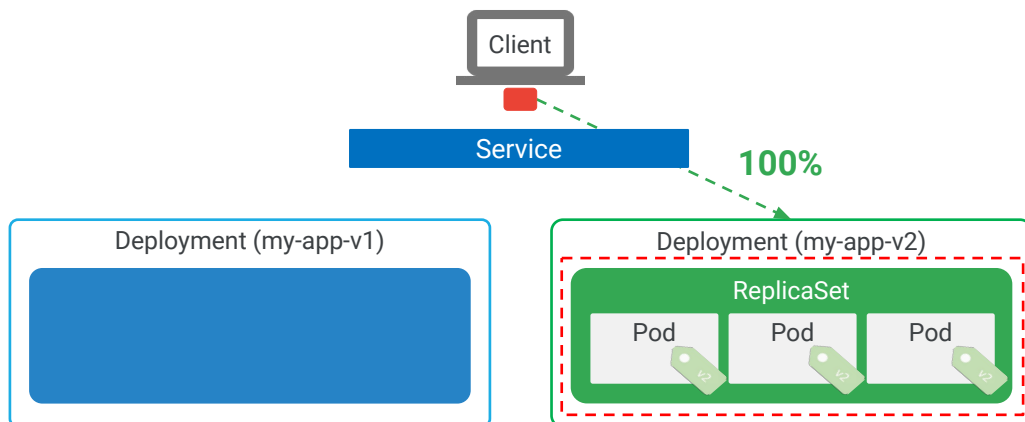
In this example, 100% of the application traffic is directed initially to my-app-v1 .

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.

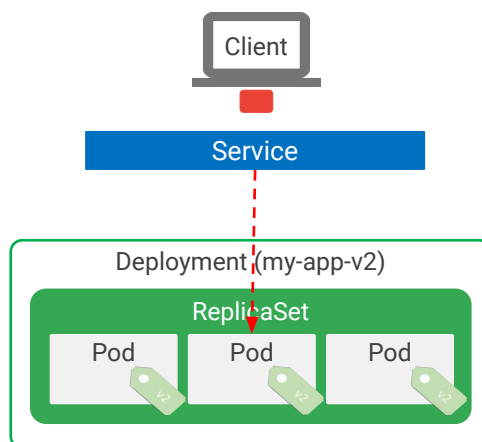
This setting allows the Service to select and direct the traffic to the Pods from both Deployments. Initially, the new version of the Deployment will start with zero replicas running. Over time, as the new version is scaled up, the old version of the Deployment can be scaled down and eventually deleted.

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

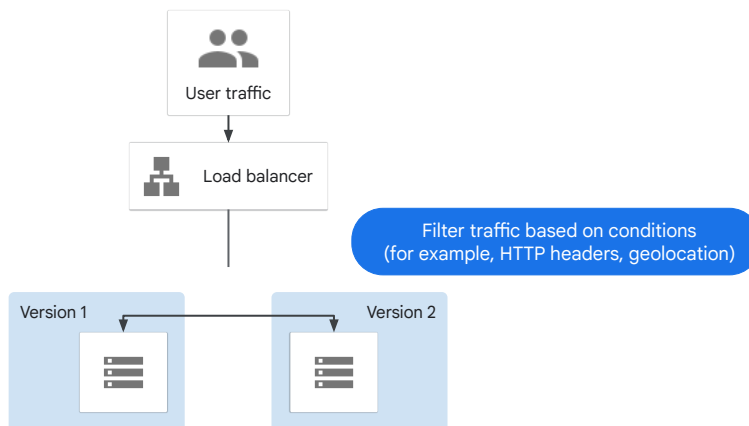
are other deployment strategies, such as A/B testing and shadow testing. These strategies are outside the scope of this course.

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.

A/B testing is used to measure the effectiveness of functionality in an application

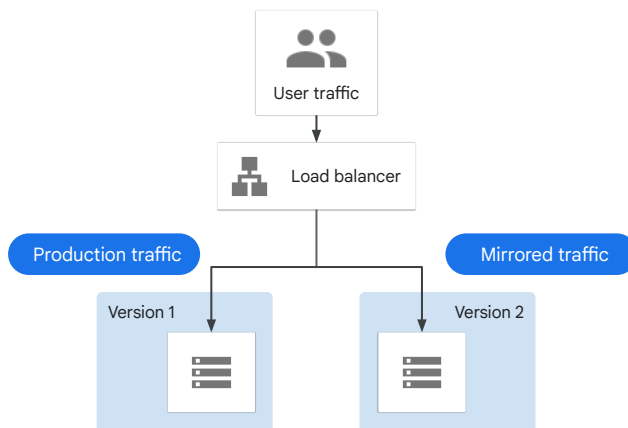


With A/B testing, you test a hypothesis by using variant implementations. A/B testing is used to make business decisions (not only predictions) based on the results derived from data.

When you perform an A/B test, you route a subset of users to new functionality based on routing rules as shown in the example here. Routing rules often include factors such as browser version, user agent, geolocation, and operating system. After you measure and compare the versions, you update the production environment with the version that yielded better results.

A/B testing is best used to measure the effectiveness of functionality in an application. Use cases for the deployment patterns discussed earlier focus on releasing new software safely and rolling back predictably. In A/B testing, you control your target audience for the new features and monitor any statistically significant differences in user behavior.

Shadow testing allows you to run a new, hidden version



Google Cloud

Sequential experiment techniques like canary testing can potentially expose customers to an inferior application version during the early stages of the test. You can manage this risk by using offline techniques like simulation. However, offline techniques do not validate the application's improvements because there is no user interaction with the new versions.

With shadow testing, you deploy and run a new version alongside the current version, but in such a way that the new version is hidden from the users, as the diagram shown illustrates. An incoming request is mirrored and replayed in a test environment. This process can happen either in real time or asynchronously after a copy of the previously captured production traffic is replayed against the newly deployed service.

You need to ensure that the shadow tests do not trigger side effects that can alter the existing production environment or the user state.

Shadow testing has many key benefits. Because traffic is duplicated, any bugs in services that are processing shadow data have no impact on production. When used with tools such as Diffy, traffic shadowing lets you measure the behavior of your service against live production traffic. This ability lets you test for errors, exceptions, performance, and result parity between application versions. Finally, there is a reduced deployment risk. Traffic shadowing is typically combined with other approaches like canary testing. After testing a new feature by using traffic shadowing, you then test the user experience by gradually releasing the feature to an increasing number of users over time. No full rollout occurs until the application meets stability

and performance requirements.

Choosing the right strategy

| Deployment or testing pattern | Zero downtime | Real production traffic testing | Releasing to users based on conditions | Rollback duration | Impact on hardware and cloud costs |
|--|---------------|---------------------------------|--|-------------------------------------|---|
| Recreate Version 1 is terminated, and Version 2 is rolled out. | ✗ | ✗ | ✗ | Fast but disruptive due to downtime | No extra setup required |
| Rolling update Version 2 is gradually rolled out and replaces Version 1. | ✓ | ✗ | ✗ | Slow | Can require extra setup for surge upgrades |
| Blue/green Version 2 is released alongside version 1; the traffic is switched to Version 2 after it is tested. | ✓ | ✗ | ✗ | Instant | Need to maintain blue and green environments simultaneously |
| Canary Version 2 is released to a subset of users, followed by a full rollout. | ✓ | ✓ | ✗ | Fast | No extra setup required |
| A/B Version 2 is released, under specific conditions, to a subset of users. | ✓ | ✓ | ✓ | Fast | No extra setup required |
| Shadow Version 2 receives real-world traffic without impacting user requests. | ✓ | ✓ | ✗ | Does not apply | Need to maintain parallel environments in order to capture and replay user requests |

Google Cloud

You can deploy and release your application in several ways. Each approach has advantages and disadvantages. The best choice comes down to the needs and constraints of your business. You should consider the following when choosing the right approach.

What are your most critical considerations? For example, is downtime acceptable?

Do costs constrain you? Does your team have the right skills to undertake complex rollout and rollback setups? Do you have tight testing controls in place, or do you want to test the new releases against production traffic to ensure the stability of the release and limit any negative impact? Do you want to test features among a pool of users to cross-verify certain business hypotheses? Can you control whether targeted users accept the update? For example, updates on mobile devices require explicit user action and might require extra permissions.

Are microservices in your environment fully autonomous? Or, do you have a hybrid of microservice-style applications working alongside traditional, difficult-to-change applications? For more information, refer to [deployment patterns on hybrid and multi-cloud environments](#). Does the new release involve any schema changes? If yes, are the schema changes too complex to decouple from the code changes?

The table shown here summarizes the salient characteristics of the deployment and testing patterns. When you weigh the advantages and disadvantages of various deployment and testing approaches, consider your business needs and technological

resources, and then select the option that benefits you the most.

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]
```

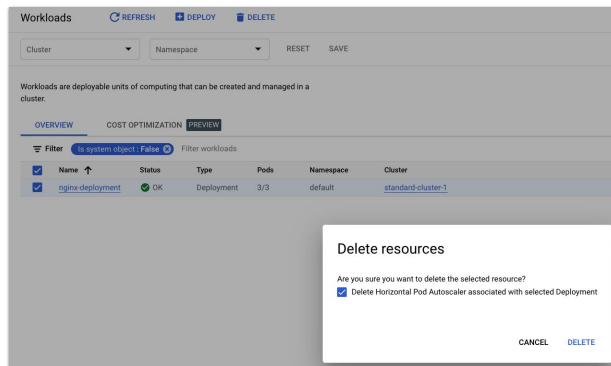
When you edit a deployment, your action normally triggers an automatic rollout. But if you have an environment where small fixes are released frequently, you'll have a large number of rollouts. In a situation like that, you'll find it more difficult to link issues with specific rollouts. To help, you 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]
```



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

Quiz

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

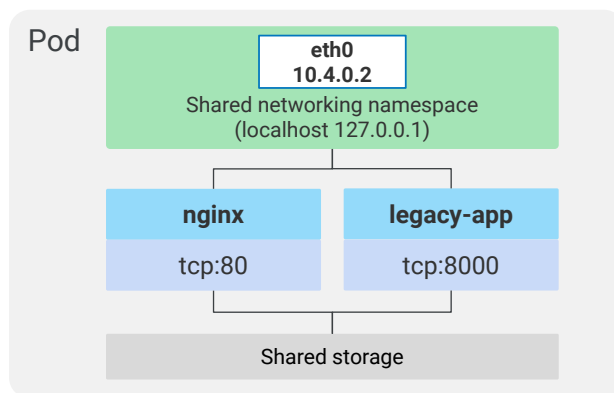
Lab: Configuring Persistent Storage
for Google Kubernetes Engine

Quiz

Summary

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

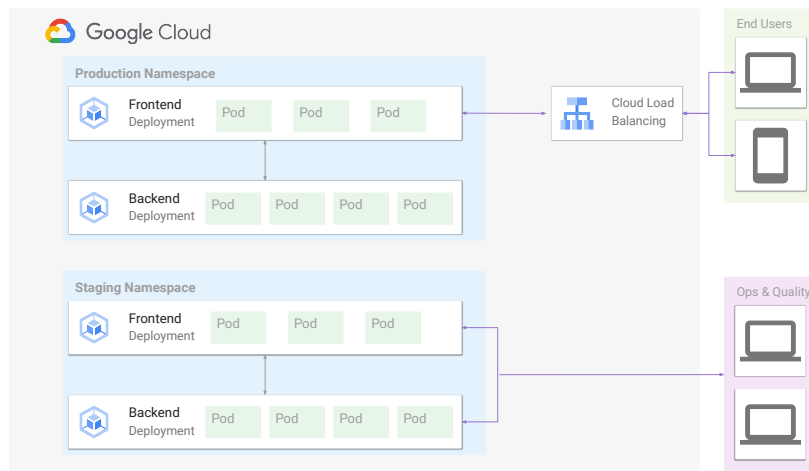
A Pod is a group of containers with shared storage and networking



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

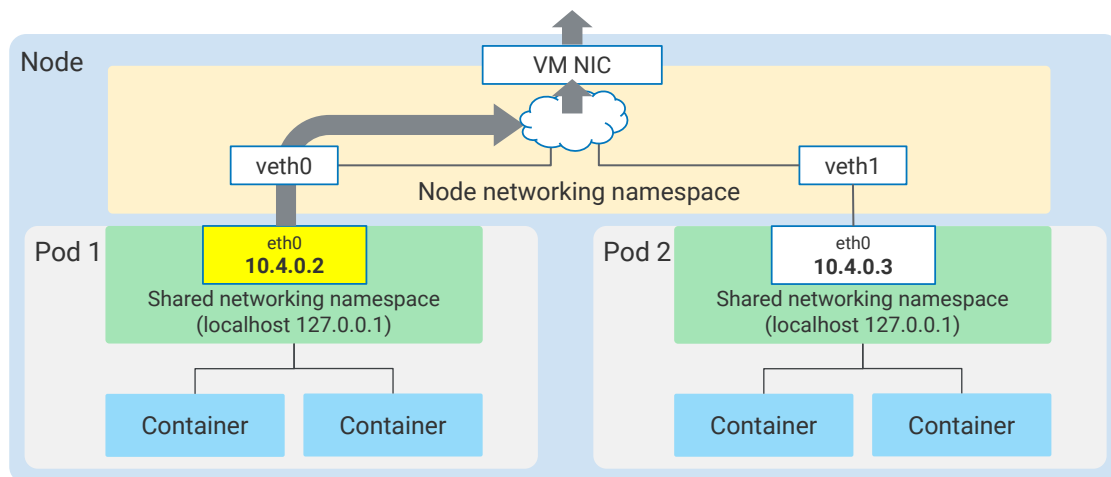


This works well for a single Pod, but your workload doesn't run in a single Pod.

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



Google Cloud

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.

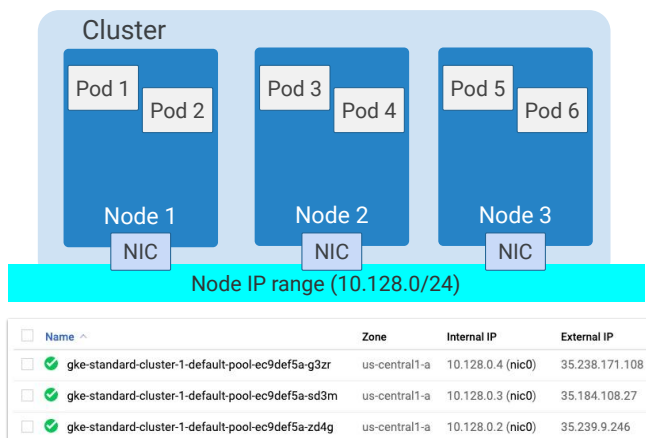
This allows the two Pods to communicate on the same node.

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



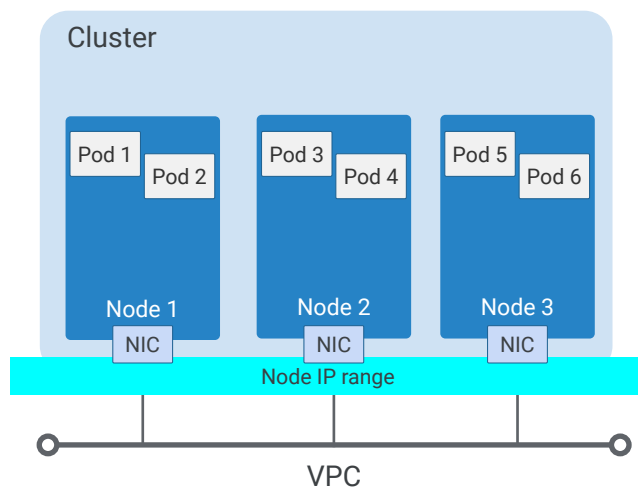
Google Cloud

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.

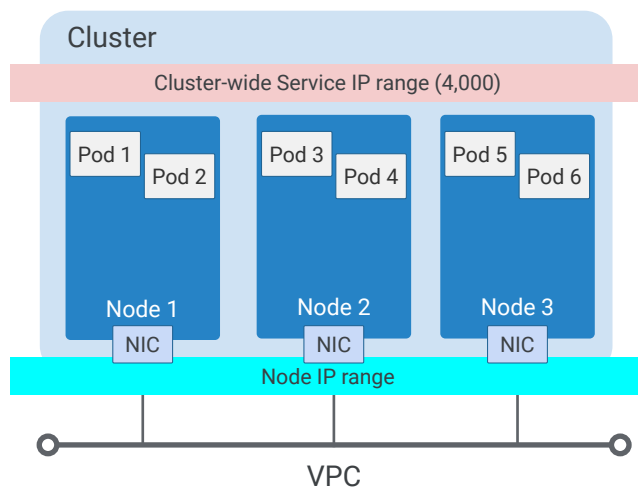
When you deploy GKE, you can select a VPC along with a region or zone. By default, a VPC has an IP subnet pre-allocated for each Google Cloud region in the world. The IP addresses in this subnet are then allocated to the compute instances that you deploy in the region.

Addressing the Pods



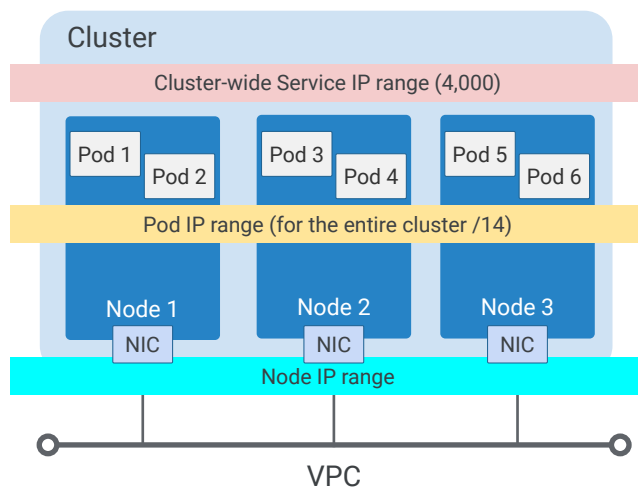
GKE cluster nodes are compute instances that GKE customizes and manages for you. These machines are assigned IP addresses from the VPC subnet that they reside in.

Addressing the Pods



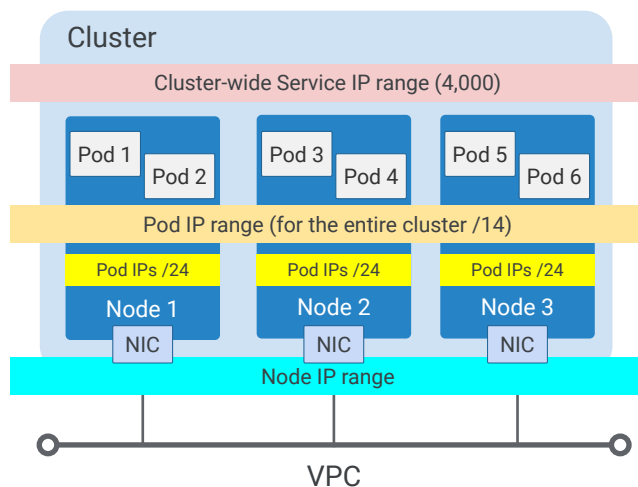
On Google Cloud, Alias IPs allow you to configure additional secondary IP addresses or IP ranges on your Compute Engine VM instances. VPC-Native GKE clusters automatically create an Alias IP range to reserve approximately 4,000 IP addresses for cluster-wide Services that you may create later. This mitigates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

Quiz

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

Are a directory which is accessible to all of the containers in a Pod.

Some Volumes are ephemeral.

Some Volumes are persistent.

PersistentVolumes

Manage durable storage in a cluster.

Are independent of the Pod's lifecycle.

Provisioned dynamically through PersistentVolumeClaims or explicitly created by a cluster admin.

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.

A volume is a directory which is accessible to all of the containers in 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.

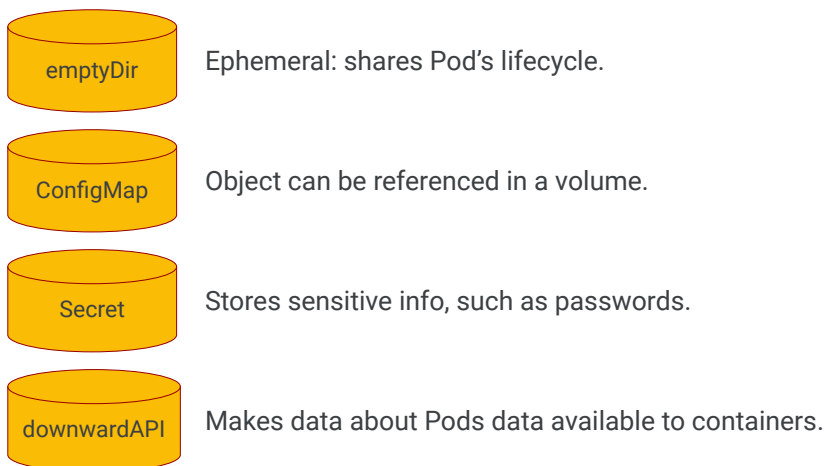
PersistentVolume resources are used to manage durable storage in a cluster. In GKE, a PersistentVolume is typically backed by a persistent disk. You can also use other storage solutions like NFS. Filestore is a NFS solution on Google Cloud. Unlike volumes, the PersistentVolume lifecycle is managed by Kubernetes.

PersistentVolume resources are cluster resources that exist independently of Pods. This means that the disk and data represented by a PersistentVolume continue to exist as the cluster changes and as Pods are deleted and recreated.

PersistentVolume resources can be provisioned dynamically through

PersistentVolumeClaims, or they can be explicitly created by a cluster administrator
You do not have to manually create and delete the backing storage.

Ephemeral volume types explained



Google Cloud

- An emptyDir volume is simply an empty directory that allows the containers within the Pod to read and write to and from it. It's created when a Pod is assigned to a node, and it exists as long as the Pod exists. However, it'll be deleted if the Pod is removed from a node for any reason. So don't use emptyDir volumes for data of lasting value. Applications usually use emptyDir for short-term purposes.

Kubernetes creates emptyDir volumes from the node's local disk, or by using a memory-backed file system.

- The configMap resource provides a way to inject application configuration data into Pods from Kubernetes. The data stored in a ConfigMap object can be referenced in a volume, as if it were a tree of files and directories. Your applications can then consume the data. For instance, if you were using a Web server in a Pod, you might use a configMap to set that Web server's parameters.
- Secrets are similar to ConfigMaps. You should use Secrets to store sensitive information, such as passwords, tokens, and ssh keys. Just like ConfigMap, a Secret Volume is created to pass sensitive information to the Pods. These Secret Volumes are backed by in-memory file systems, so the Secrets are

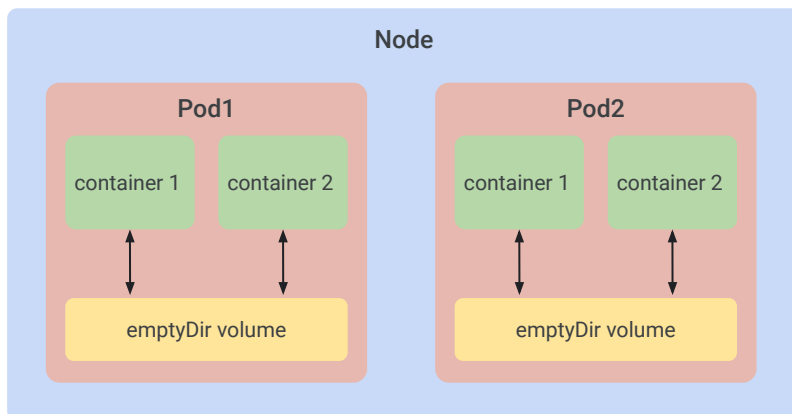
- never written to non-volatile storage. And it's a common practice to obfuscate the values that go into secrets using the familiar base64 encoding.

Beyond that, though, you should not assume that Secrets are secret just because of the way they are configured. Having a differentiation between ConfigMaps and Secrets allows you to manage non-sensitive and sensitive Pod configuration data differently. You will probably apply different permissions to each.

- The downwardAPI Volume type is used to make downwardAPI data available to applications. And what's the downward API? It's a way containers can learn about their Pod environment. For example, suppose your containerized application needs to construct an identifier that's guaranteed to be unique in the cluster. Wouldn't it be easiest to base that identifier on the name of this Pod, since Pod names are unique in the cluster too? The downwardAPI is how your application can fetch the name of the Pod it's running on, if you choose to make it available.

All these volume types are fundamentally similar. ConfigMap, Secret, and downwardAPI are essentially EmptyDir Volumes pre-configured to contain configurations from the GKE API. ConfigMap and Secret are discussed later in this module, while downwardAPI is out of this course's scope.

An emptyDir volume is first created when a Pod is assigned to a node



An emptyDir volume is first created when a Pod is assigned to a node, and exists as long as that Pod is running on that node. The emptyDir volume is initially empty. All containers in the Pod can read and write the same files in the emptyDir volume, though that volume can be mounted at the same or different paths in each container. When a Pod is removed from a node for any reason, the data in the emptyDir is deleted permanently.

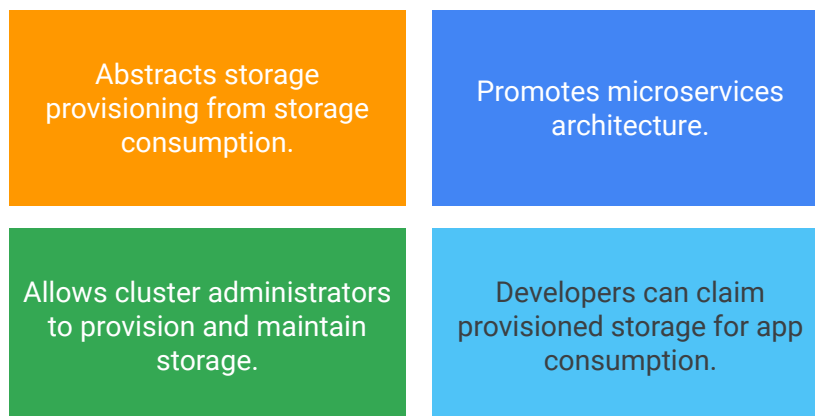
It should be noted that a container crashing does not remove a Pod from a node. The data in an emptyDir volume is safe across container crashes.

Creating a Pod with an emptyDir volume

```
apiVersion: v1
kind: Pod
metadata:
  name: web
spec:
  containers:
  - name: web
    image: nginx
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

Here, we're creating a Pod with an emptyDir volume. Some uses for an emptyDir are a scratch space, such as for a disk-based merge sort, checkpointing a long computation for recovery from crashes or holding files that a content-manager container fetches while a web server container serves the data.

The benefits of PersistentVolumes



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.

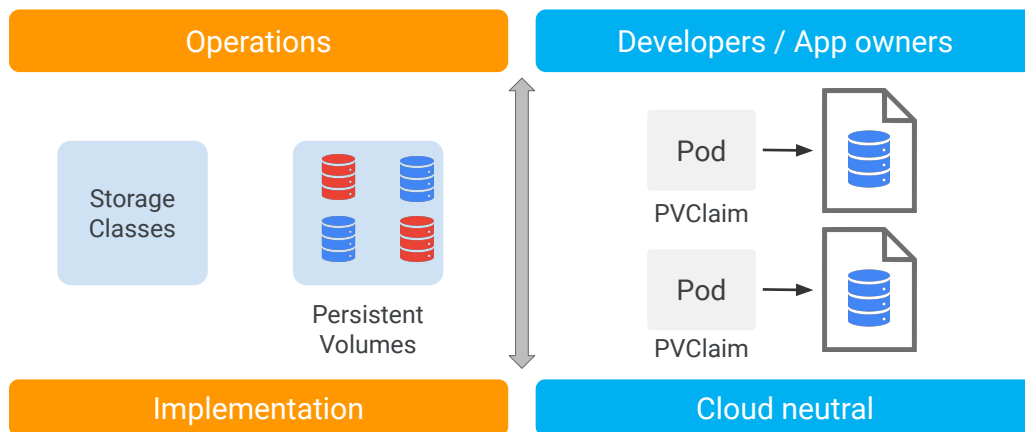
The two job roles can work independently of each other.

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.

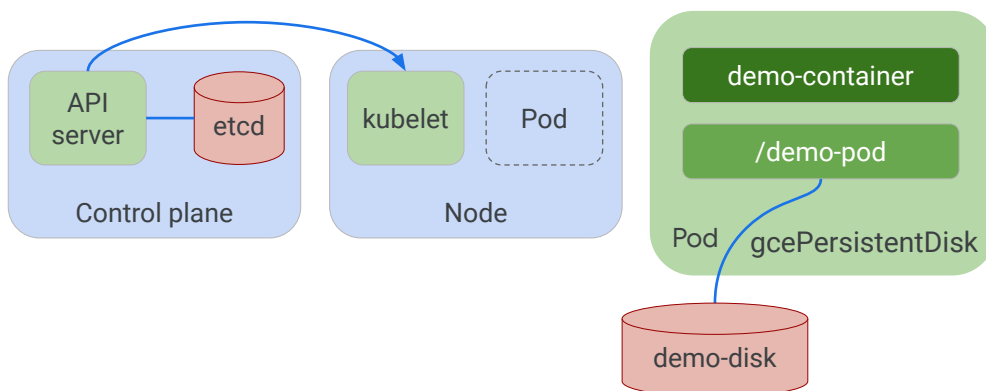
This allows the operations team to manage the cloud services they wish to use and allows the application owners to focus on what their application requires rather than the specific implementation detail.

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
```

Google Cloud

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.

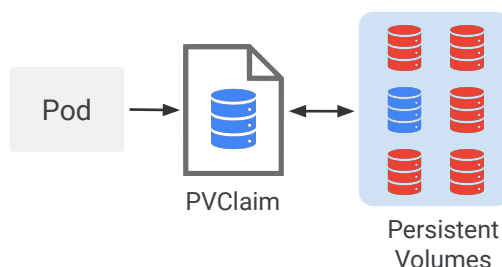
Whenever you need to reconfigure an application to move it from one environment to another, there's a risk of error. To address this problem, Kubernetes provides an abstraction called Persistent Volumes. This abstraction lets a Pod claim a Volume of a certain size, or of a certain name, from a pool of 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.

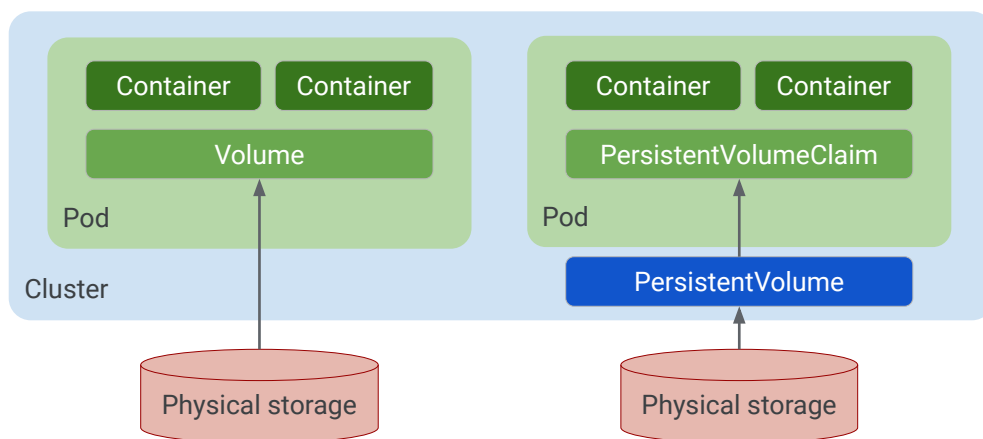
GKE can use Compute Engine Persistent Disks as PersistentVolumes.

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

Google Cloud

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

| <input type="checkbox"/> | Name ↑ | Provisioner | Type | Zone | Cluster |
|--------------------------|--------------|-----------------------|-------------|------|--------------------|
| <input type="checkbox"/> | premium-rwo | pd.csi.storage.gke.io | pd-ssd | | standard-cluster-1 |
| <input type="checkbox"/> | standard | kubernetes.io/gce-pd | pd-standard | | standard-cluster-1 |
| <input type="checkbox"/> | standard-rwo | pd.csi.storage.gke.io | pd-balanced | | 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

Quiz

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

Quiz

Summary

Question #1

Question

Which control plane component does the `kubectl` command interact with?

- A. etcd
- B. kubelet
- C. kube-apiserver
- D. The GKE API

Question #1

Answer

Which control plane component does the `kubectl` command interact with?

- A. etcd
- B. kubelet
- C. kube-apiserver
- D. The GKE API



A: Incorrect.

Feedback: Only kube-apiserver interacts with etcd.

B: Incorrect.

Feedback: Only kube-apiserver interacts with the kubelets on the node.

C: Correct.

D: Incorrect.

Feedback: kubectl is not aware of the GKE API.

Question #2

Question

You want to use `kubectl` to configure your cluster, but first you must configure it. Where does the `kubectl` command store its configuration file?

- A. `kubectl` always prompts the user for credentials before executing commands.
- B. `kubectl` uses the same authorization and credential tokens as the `gcloud` CLI utilities.
- C. The configuration information is stored in environment variables in the current shell when required.
- D. The configuration information is stored in the `$HOME/.kube/config` file.

Question #2

Answer

You want to use `kubectl` to configure your cluster, but first you must configure it. Where does the `kubectl` command store its configuration file?

- A. `kubectl` always prompts the user for credentials before executing commands.
- B. `kubectl` uses the same authorization and credential tokens as the `gcloud` CLI utilities.
- C. The configuration information is stored in environment variables in the current shell when required.
- D. The configuration information is stored in the `$HOME/.kube/config` file.



A: Incorrect.

Feedback: Review the lesson on the `kubectl` command.

B: Incorrect.

Feedback: Review the lesson on the `kubectl` command.

C: Incorrect.

Feedback: Review the lesson on the `kubectl` command.

D: Correct.

Question #3

Question

You want to use a `kubectl get` command to identify which Node each Pod is running on. Which command do you need to execute?

- A. `kubectl get nodes`
- B. `kubectl get nodes -o=yaml`
- C. `kubectl get pods`
- D. `kubectl get pods -o=wide`

Question #3

Answer

You want to use a `kubectl get` command to identify which Node each Pod is running on. Which command do you need to execute?

- A. `kubectl get nodes`
- B. `kubectl get nodes -o=yaml`
- C. `kubectl get pods`
- D. `kubectl get pods -o=wide`



A: Incorrect.

Feedback: Review the lesson on the `kubectl` command.

B: Incorrect.

Feedback: Review the lesson on the `kubectl` command.

C: Incorrect.

Feedback: Review the lesson on the `kubectl` command.

D: Correct.

Question #4

Question



What is the purpose of a Service? Choose all that are true (2 correct answers)

- A. To allow you to choose how Pods are exposed.
- B. To allow you to put constraints on Pods' resource consumption.
- C. To provide a load-balancing network endpoint for Pods.
- D. To provide a way to inspect and diagnose code running in a Pod.

Question #4

Answer

What is the purpose of a Service? Choose all that are true (2 correct answers)

- A. To allow you to choose how Pods are exposed. 
- B. To allow you to put constraints on Pods' resource consumption.
- C. To provide a load-balancing network endpoint for Pods. 
- D. To provide a way to inspect and diagnose code running in a Pod.

- A: Correct.**
- B: Incorrect.**
- C: Correct.**
- D: Incorrect.**

Question #5

Question

After a Deployment has been created and its component Pods are running, which component is responsible for ensuring that a replacement Pod is launched whenever a Pod fails or is evicted?

- A. DaemonSet
- B. Deployment
- C. ReplicaSet
- D. StatefulSet

Question #5

Answer

After a Deployment has been created and its component Pods are running, which component is responsible for ensuring that a replacement Pod is launched whenever a Pod fails or is evicted?

- A. DaemonSet
- B. Deployment
- C. ReplicaSet
- D. StatefulSet



A: Incorrect.

Feedback: Review the Deployments lesson.

B: Incorrect.

Feedback: Review the Deployments lesson.

C: Correct.

D: Incorrect.

Feedback: Review the Deployments lesson.

Question #6

Question


What is the relationship between Deployments and ReplicaSets?

- A. A Deployment configures a ReplicaSet controller to create and maintain a specific version of the Pods that the Deployment specifies.
- B. A Deployment configures a ReplicaSet controller to create and maintain all the Pods that the Deployment specifies, regardless of their version.
- C. A ReplicaSet configures a Deployment controller to create and maintain a specific version of the Pods that the Deployment specifies.
- D. There is no relationship; in modern Kubernetes, Replication Controllers are typically used to maintain a set of Pods in a running state.

Question #6

Answer

What is the relationship between Deployments and ReplicaSets?

- A. A Deployment configures a ReplicaSet controller to create and maintain a specific version of the Pods that the Deployment specifies. 
- B. A Deployment configures a ReplicaSet controller to create and maintain all the Pods that the Deployment specifies, regardless of their version.
- C. A ReplicaSet configures a Deployment controller to create and maintain a specific version of the Pods that the Deployment specifies.
- D. There is no relationship; in modern Kubernetes, Replication Controllers are typically used to maintain a set of Pods in a running state.

Google Cloud

A: Correct.

B: Incorrect.

Feedback: Review the lesson on Deployments.

C: Incorrect.

Feedback: Review the lesson on Deployments.

D: Incorrect.

Feedback: Review the lesson on Deployments.

Question #7

Question

What type of application is suited for use with a Deployment?

- A. Batch
- B. Stateful
- C. Stateless
- D. Written in Go

Question #7

Answer

What type of application is suited for use with a Deployment?

- A. Batch
- B. Stateful
- C. Stateless
- D. Written in Go



A: Incorrect.

Feedback: Review the lesson on Deployments.

B: Incorrect.

Feedback: Review the lesson on Deployments.

C: Correct.

D: Incorrect.

Feedback: Kubernetes does not care what languages the applications it controls are written in.

Question #8

Question


You have made a number of changes to your deployment and applied those changes. Which command should you use to rollback the environment to the deployment identified in the deployment history as revision 2?

- A. Run `'kubectl apply -f DEPLOYMENT_FILE --to-revision=2'`
- B. Run `'kubectl rollout undo deployment'` twice.
- C. Run `'kubectl rollout undo deployment --to-revision=2'`
- D. Select the desired revision from the revision history list in the Cloud Console.

Question #8

Answer

You have made a number of changes to your deployment and applied those changes. Which command should you use to rollback the environment to the deployment identified in the deployment history as revision 2?

- A. Run `'kubectl apply -f DEPLOYMENT_FILE --to-revision=2'`
- B. Run `'kubectl rollout undo deployment'` twice.
- C. Run `'kubectl rollout undo deployment --to-revision=2'` 
- D. Select the desired revision from the revision history list in the Cloud Console.

A: Incorrect.

Feedback: Review the lesson on Deployments.

B: Incorrect.

Feedback: Review the lesson on Deployments.

C: Correct.

D: Incorrect.

Feedback: Review the lesson on Deployments.

Question #9

Question

You are resolving a range of issues with a Deployment and need to make a large number of changes. Which command can you execute to group these changes into a single rollout, thus avoiding pushing out a large number of rollouts?

- A. `kubectl delete deployment`
- B. `kubectl rollout pause deployment`
- C. `kubectl rollout resume deployment`
- D. `kubectl stop deployment`

Question #9

Answer

You are resolving a range of issues with a Deployment and need to make a large number of changes. Which command can you execute to group these changes into a single rollout, thus avoiding pushing out a large number of rollouts?

- A. `kubectl delete deployment`
- B. `kubectl rollout pause deployment`
- C. `kubectl rollout resume deployment`
- D. `kubectl stop deployment`



A: Incorrect.

Feedback: Review the lesson on Deployments.

B: Correct.

C: Incorrect.

Feedback: Review the lesson on Deployments.

D: Incorrect.

Feedback: Review the lesson on Deployments.

Question #10

Question

In GKE, what is the source of the IP addresses for Pods?

- A. Address ranges assigned to your Virtual Private Cloud
- B. Arbitrary network addresses per cluster
- C. Loopback network addresses

Question #10

Answer

In GKE, what is the source of the IP addresses for Pods?

- A. Address ranges assigned to your Virtual Private Cloud
- B. Arbitrary network addresses per cluster
- C. Loopback network addresses



A: Correct.

B: Incorrect.

Feedback: Review the lesson on Pod networking.

C: Incorrect.

Feedback: Review the lesson on Pod networking.

Question #11

Question

Your Pod has been rescheduled and the IP address that was assigned to the Pod when it was originally scheduled is no longer accessible. What is the reason for this?

- A. The new Pod IP address is blocked by a firewall.
- B. The new Pod has received a different IP address.
- C. The old Pod IP address is blocked by a firewall.
- D. The Pod IP range for the cluster is exhausted.

Question #11

Answer

Your Pod has been rescheduled and the IP address that was assigned to the Pod when it was originally scheduled is no longer accessible. What is the reason for this?

- A. The new Pod IP address is blocked by a firewall.
- B. The new Pod has received a different IP address.
- C. The old Pod IP address is blocked by a firewall.
- D. The Pod IP range for the cluster is exhausted.



A: Incorrect.

Feedback: Review the lesson on Pod networking.

B: Correct.

C: Incorrect.

Feedback: Review the lesson on Pod networking.

D: Incorrect.

Feedback: Review the lesson on Pod networking.

Question #12

Question

What happens if a Pod fails while it is using a persistent volume?

- A. The volumes are deleted, and their contents are lost.
- B. The volumes are unmounted from the failing Pod, and their contents are deleted.
- C. The volumes are unmounted from the failing Pod, and their contents revert to what they had before the Pod was attached to it.
- D. The volumes are unmounted from the failing Pod, and they continue to exist with their last contents.

Question #12

Answer

What happens if a Pod fails while it is using a persistent volume?

- A. The volumes are deleted, and their contents are lost.
- B. The volumes are unmounted from the failing Pod, and their contents are deleted.
- C. The volumes are unmounted from the failing Pod, and their contents revert to what they had before the Pod was attached to it.
- D. The volumes are unmounted from the failing Pod, and they continue to exist with their last contents.



A: Incorrect.

Feedback: Review the lesson on volumes.

B: Incorrect.

Feedback: Review the lesson on volumes.

C: Incorrect.

Feedback: Review the lesson on volumes.

D: Correct.

Question #13

Question

How can a Pod request persistent storage without specifying the details of how that storage is to be implemented?

- A. By using a `gcePersistentDisk`
- B. By using a `PersistentVolume`
- C. By using a `PersistentVolumeClaim`
- D. By using an `emptyDir`

Question #13

Answer

How can a Pod request persistent storage without specifying the details of how that storage is to be implemented?

- A. By using a `gcePersistentDisk`
- B. By using a `PersistentVolume`
- C. By using a `PersistentVolumeClaim`
- D. By using an `emptyDir`



A: Incorrect.

Feedback: Review the video on `PersistentVolumes` and `PersistentVolumeClaims`.

B: Incorrect.

Feedback: Review the video on `PersistentVolumes` and `PersistentVolumeClaims`.

C: Correct.

D: Incorrect.

Feedback: Review the video on `PersistentVolumes` and `PersistentVolumeClaims`.

Question #14

Question

An application owner has created a Pod manifest using a `PersistentVolumeClaim` with a `StorageClassName` value of `standard`. What type of storage is used for this volume in a GKE Cluster?

- A. Google Persistent Disk
- B. Local volume on the node
- C. Memory Backed
- D. NFS Storage

Question #14

Answer

An application owner has created a Pod manifest using a `PersistentVolumeClaim` with a `StorageClassName` value of `standard`. What type of storage is used for this volume in a GKE Cluster?

- A. Google Persistent Disk
- B. Local volume on the node
- C. Memory Backed
- D. NFS Storage



A: Correct.

B: Incorrect.

Feedback: Review the lesson on Kubernetes Volumes.

C: Incorrect.

Feedback: Review the lesson on Kubernetes Volumes.

D: Incorrect.

Feedback: Review the lesson on Kubernetes Volumes.

Agenda

The kubectl command

Deployments

Lab: Creating Google Kubernetes
Engine Deployments

Pod Networking

Volumes

Lab: Configuring Persistent Storage
for Google Kubernetes Engine

Quiz

[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.

