Welcome to the Kubernetes Architecture module of our course. It helps to know the parts of Kubernetes and to understand the philosophy it implements. How does Kubernetes expect you to tell it what to do? And what choices do you have for describing your workloads? That's the theme of this module.

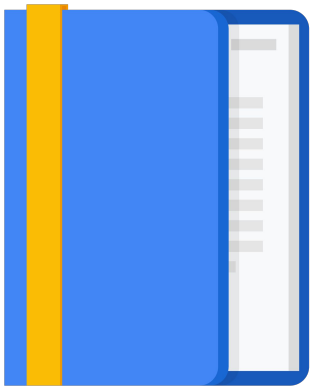
Learn how to ...

Understand Kubernetes objects and the Kubernetes control plane.

Deploy a Kubernetes cluster using Google Kubernetes Engine (GKE).

Deploy Pods to a GKE cluster.

View and manage Kubernetes objects.



Google Cloud

In this module you'll learn how to understand how the Kubernetes architecture is laid out, deploy a Kubernetes cluster using Google Kubernetes Engine, deploy Pods to a GKE cluster, and, view and manage several very useful kinds of Kubernetes objects.

Agenda

Kubernetes Concepts

Kubernetes Components

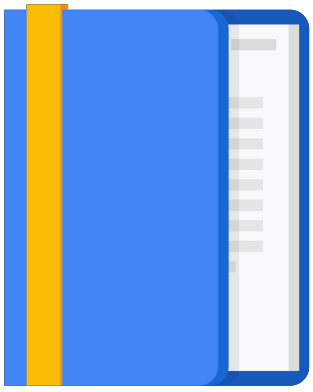
Google Kubernetes Engine Concepts

Object Management

Lab: Deploying Google Kubernetes Engine

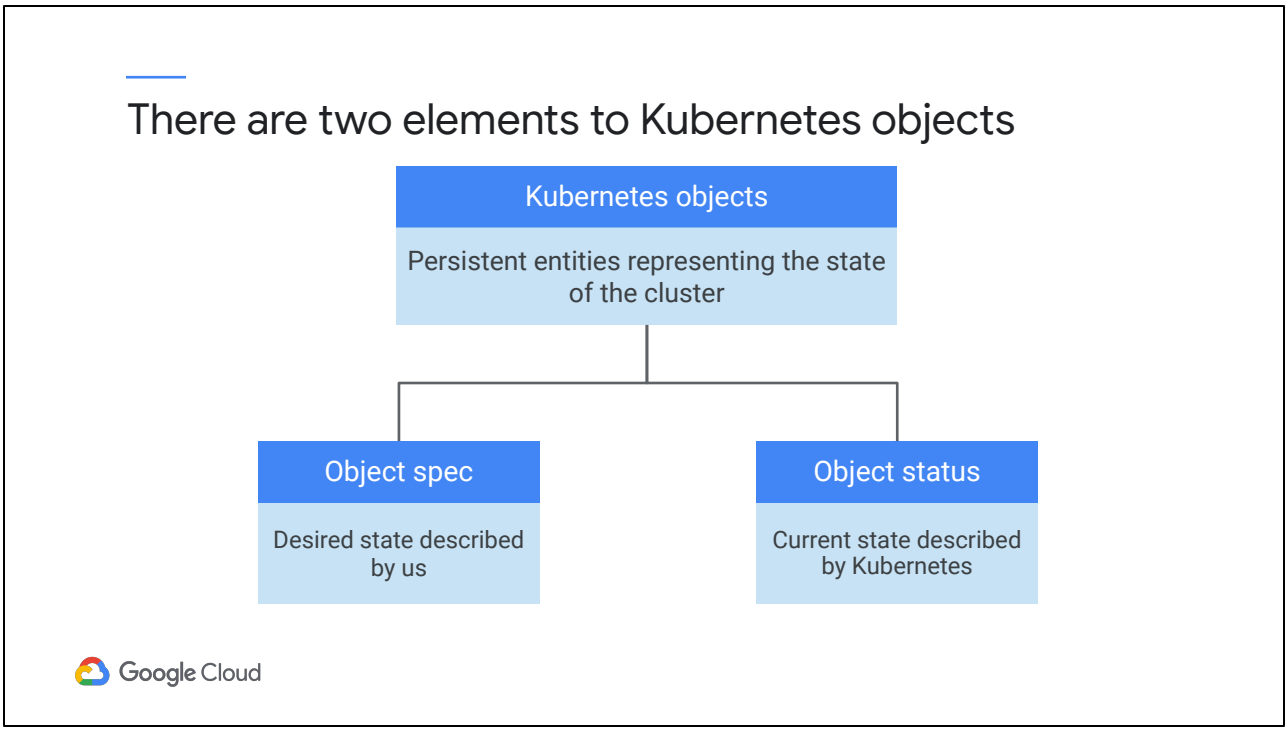
Migrate for Anthos

Summary



Google Cloud

In this lesson, we'll lay out the fundamental components of the Kubernetes operating philosophy.
To understand how Kubernetes works, there are two related concepts you need to understand. The first is the Kubernetes object model. Each thing Kubernetes manages is represented by an object, and you can view and change these objects' attributes and state. The second is the principle of declarative management. Kubernetes expects you to tell it what you want the state of the objects under its management to be; it will work to bring that state into being and keep it there. How does it do that? By means of its so-called "watch loop."

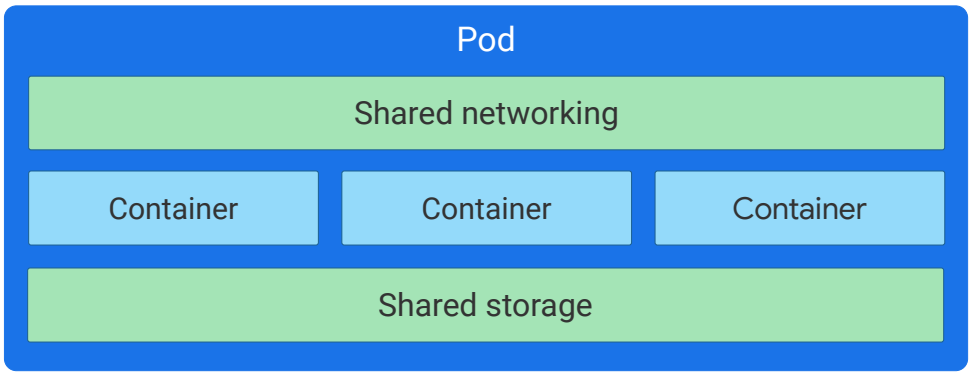


Formally, a Kubernetes object is defined as a persistent entity that represents the state of something running in a cluster: its desired state and its current state. Various kinds of objects represent containerized applications, the resources that are available to them, and the policies that affect their behavior. Kubernetes objects have two important elements.

You give Kubernetes an Object *spec* for each object you want to create. With this spec, you define the desired state of the object by providing the characteristics that you want.

The Object *status* is simply the current state of the object provided by the Kubernetes control plane. By the way, we use this term “Kubernetes control plane” to refer to the various system processes that collaborate to make a Kubernetes cluster work. You’ll learn about these processes later in this module.

Containers in a Pod share resources



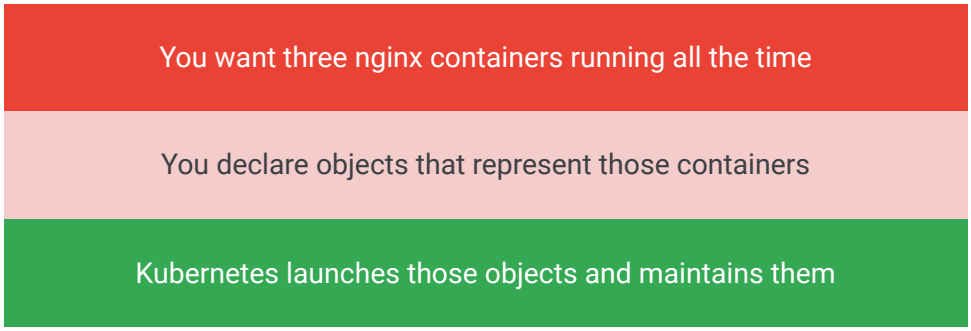
Google Cloud

Each object is of a certain type, or “Kind,” as Kubernetes calls them. Pods are the basic building block of the standard Kubernetes model, and they’re the smallest deployable Kubernetes object. Maybe you were expecting me to say that the smallest Kubernetes object is the container. Not so. Every running container in a Kubernetes system is in a Pod.

A Pod embodies the environment where the containers live, and that environment can accommodate one *or more* containers.

If there is more than one container in a pod, they are tightly coupled and share resources including networking and storage. Kubernetes assigns each Pod a unique IP address. Every container within a Pod shares the network namespace, including IP address and network ports. Containers within the same Pod can communicate through localhost, 127.0.0.1. A Pod can also specify a set of storage Volumes, to be shared among its containers.

Running three nginx containers



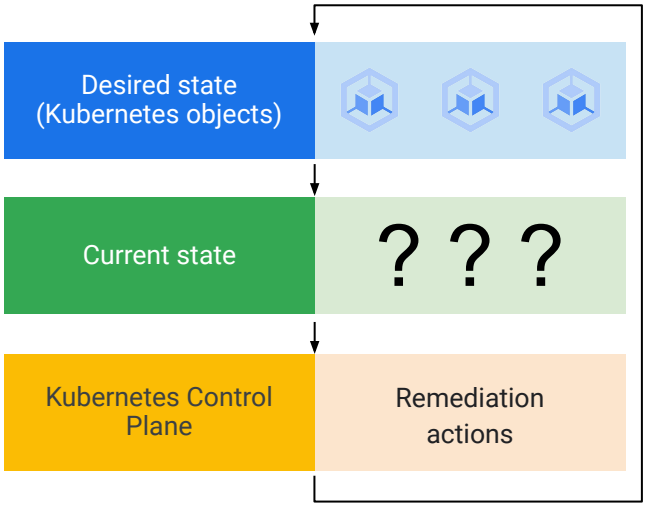
Google Cloud

Let's consider a simple example where you want three instances of the nginx Web server, each in its own container, running all the time.

How is this achieved in Kubernetes? Remember that Kubernetes embodies the principle of declarative management. You declare some objects to represent those nginx containers. What Kind of object? Perhaps Pods.

Now it is Kubernetes's job to launch those Pods and keep them in existence. Be careful: Pods are not self-healing. If we want to keep all our nginx Web servers not just in existence but also working together as a team, we might want to ask for them using a more sophisticated Kind of object. I'll tell you how later in this module.

Desired state compared to current state



Google Cloud

Let's suppose we have given Kubernetes a desired state that consists of three nginx Pods, always kept running. We did this by telling Kubernetes to create and maintain one or more objects that represent them.

Now Kubernetes compares the desired state to the current state. Let's imagine that our declaration of three nginx containers is completely new. The current state does not match the desired state.

So Kubernetes, specifically its control plane, will remedy the situation. Because the number of desired Pods running for the object we declared is 3, and 0 are presently running, 3 will be launched.

And the Kubernetes control plane will continuously monitor the state of the cluster, endlessly comparing reality to what has been declared, and remedying the state as needed.

Agenda

Kubernetes Concepts

Kubernetes Components

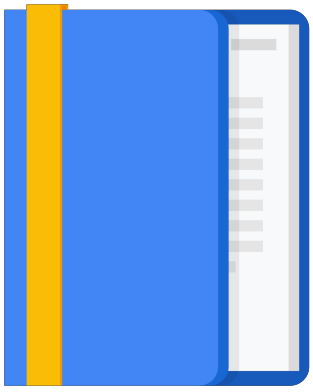
Google Kubernetes Engine Concepts

Object Management

Lab: Deploying Google Kubernetes Engine

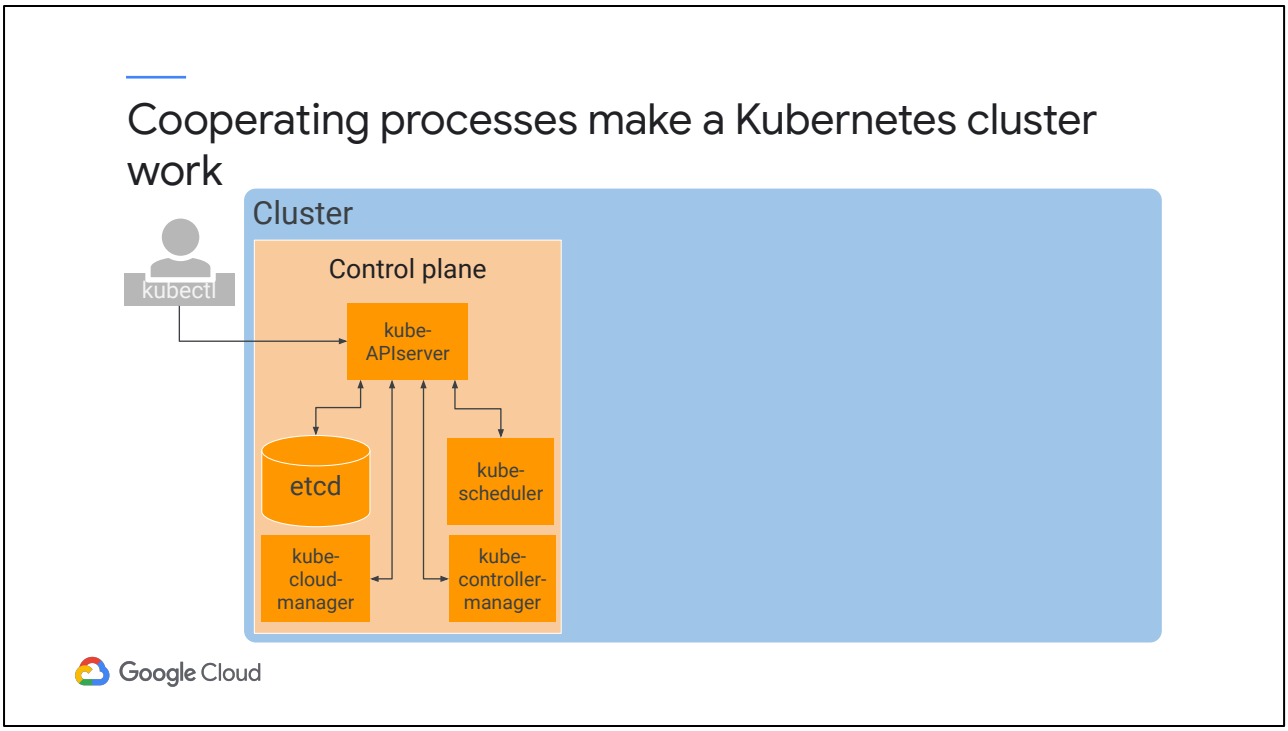
Migrate for Anthos

Summary



Google Cloud

In the previous lesson, I mentioned the Kubernetes control plane, which is the fleet of cooperating processes that make a Kubernetes cluster work. Even though you'll only work directly with a few of these components, it helps to know about them and the role each plays. I'll build up a Kubernetes cluster part by part, explaining each piece as I go. After I'm done, I'll show you how a Kubernetes cluster running in GKE is a lot less work to manage than one you provisioned yourself.



First and foremost, your cluster needs computers. Nowadays the computers that compose your clusters are usually virtual machines. They *always* are in GKE, but they could be physical computers too. One computer is called the “control plane,” and the others are called simply “nodes.” The job of the nodes is to run Pods. The job of the control plane is to coordinate the entire cluster. We will discuss its control-plane components first.

Several critical Kubernetes components run on the control plane. The single component that you interact with directly is the kube-apiserver. This component’s job is to accept commands that view or change the state of the cluster, including launching Pods.

You will use the kubectl command frequently; this command’s job is to connect to kube-apiserver and communicate with it using the Kubernetes API. kube-apiserver also authenticates incoming requests, determines whether they are authorized and valid, and manages admission control. But it’s not just kubectl that talks with kube-apiserver. In fact, any query or change to the cluster’s state must be addressed to the kube-apiserver.

etcd is the cluster’s database. Its job is to reliably store the state of the cluster. This includes all the cluster configuration data; and more dynamic information such as

what nodes are part of the cluster, what Pods should be running, and where they should be running. You never interact directly with etcd; instead, kube-apiserver interacts with the database on behalf of the rest of the system.

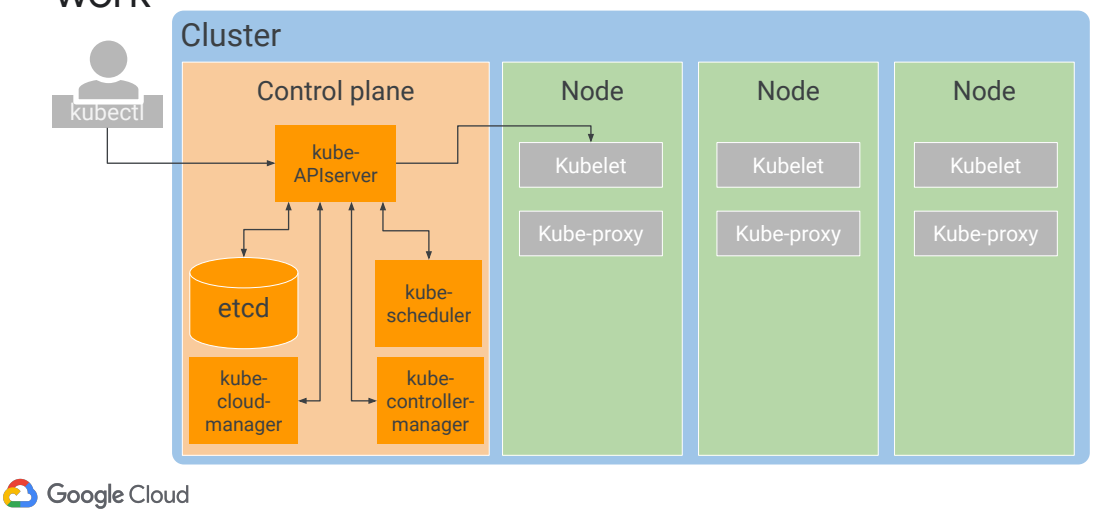
kube-scheduler is responsible for scheduling Pods onto the nodes. To do that, it evaluates the requirements of each individual Pod and selects which node is most suitable. But it doesn't do the work of actually launching Pods on Nodes. Instead, whenever it discovers a Pod object that doesn't yet have an assignment to a node, it chooses a node and simply writes the name of that node into the Pod object. Another component of the system is responsible for then launching the Pods, and you will see it very soon.

But how does kube-scheduler decide where to run a Pod? It knows the state of all the nodes, and it will also obey constraints that you define on where a Pod may run, based on hardware, software, and policy. For example, you might specify that a certain Pod is only allowed to run on nodes with a certain amount of memory. You can also define affinity specifications, which cause groups of pods to prefer running on the same node; or anti-affinity specifications, which ensure that pods do not run on the same node. You will learn more about some of these tools in later modules.

kube-controller-manager has a broader job. It continuously monitors the state of a cluster through Kube-APIserver. Whenever the current state of the cluster doesn't match the desired state, kube-controller-manager will attempt to make changes to achieve the desired state. It's called the "controller manager" because many Kubernetes objects are maintained by loops of code called controllers. These loops of code handle the process of remediation. Controllers will be very useful to you. To be specific, you'll use certain kinds of Kubernetes controllers to manage workloads. For example, remember our problem of keeping 3 nginx Pods always running. We can gather them into a controller object called a Deployment that both keeps them running and lets us scale them and bring them together underneath a front end. We'll meet Deployments later in this module.

Other kinds of controllers have system-level responsibilities. For example, Node Controller's job is to monitor and respond when a node is offline.
kube-cloud-manager manages controllers that interact with underlying cloud providers. For example, if you manually launched a Kubernetes cluster on Google Compute Engine, kube-cloud-manager would be responsible for bringing in Google Cloud features like load balancers and storage volumes when you needed them.

Cooperating processes make a Kubernetes cluster work



Google Cloud

Each node runs a small family of control-plane components too.

For example, each node runs a kubelet. You can think of kubelet as Kubernetes's agent on each node. When the kube-apiserver wants to start a Pod on a node, it connects to that node's kubelet. Kubelet uses the container runtime to start the Pod and monitors its lifecycle, including readiness and liveness probes, and reports back to Kube-APIserver. Do you remember our use of the term "container runtime" in the previous module? This is the software that knows how to launch a container from a container image. The world of Kubernetes offers several choices of container runtimes, but the Linux distribution that GKE uses for its nodes launches containers using containerd, the runtime component of Docker.

kube-proxy's job is to maintain network connectivity among the Pods in a cluster. In open-source Kubernetes, it does so using the firewalling capabilities of iptables, which are built into the Linux kernel. Later, we will learn how GKE handles pod networking.

Agenda

Kubernetes Concepts

Kubernetes Components

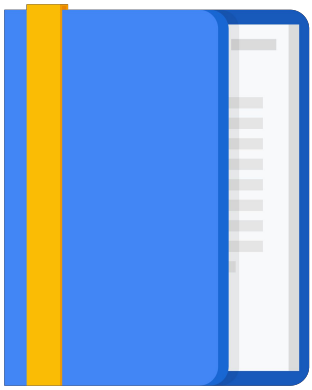
Google Kubernetes Engine Concepts

Object Management

Lab: Deploying Google Kubernetes Engine

Migrate for Anthos

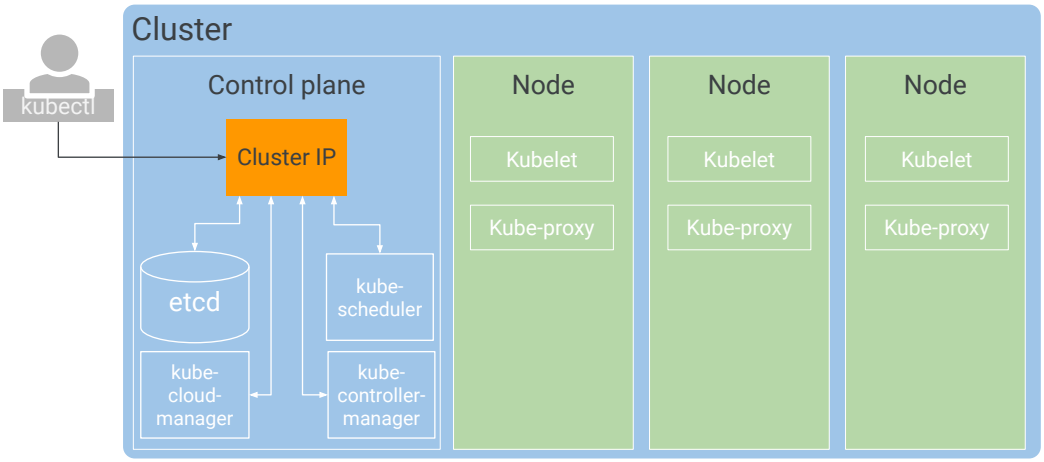
Summary



Google Cloud

Next, we'll introduce concepts specific to Google Kubernetes Engine. That diagram of the Kubernetes control plane had a lot of components, didn't it? Setting up a Kubernetes cluster by hand is tons of work.

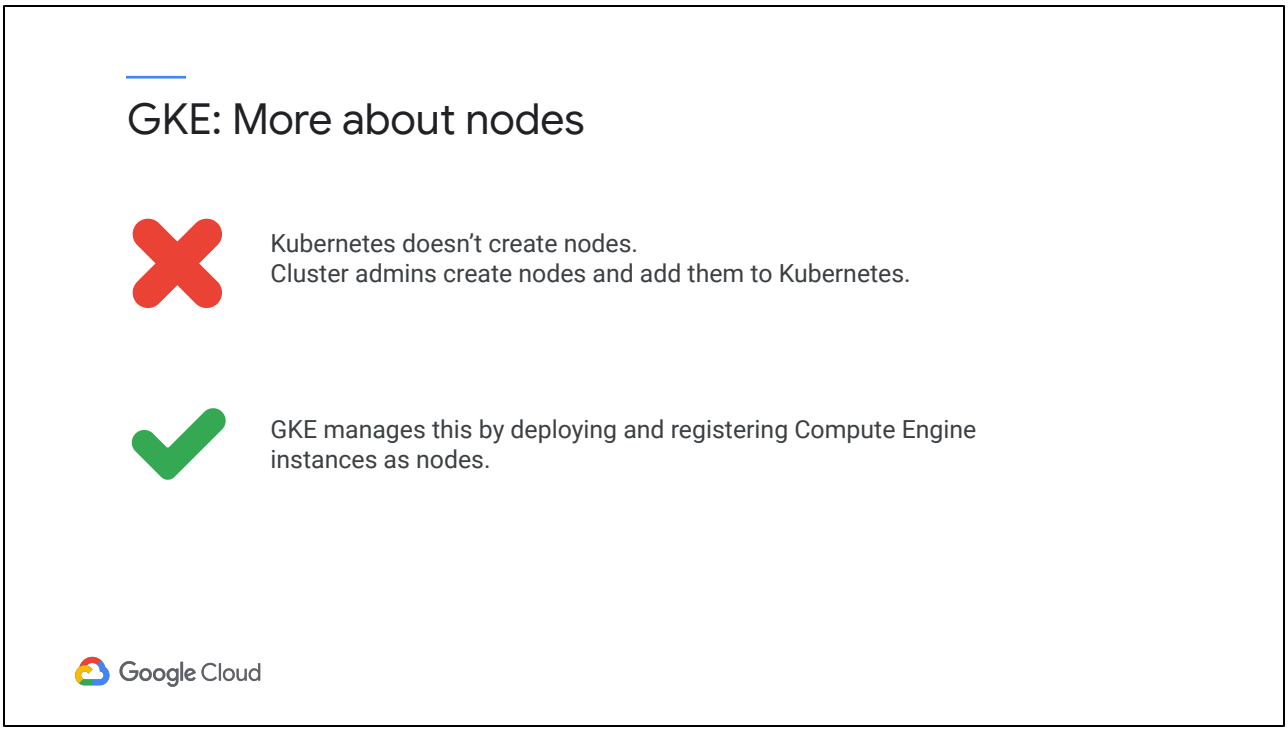
GKE manages all the control plane components



Google Cloud

Fortunately, there is an open-source command called kubeadm that can automate much of the initial setup of a cluster. But if a node fails or needs maintenance, a human administrator has to respond manually. I suspect you can see why many people like the idea of a managed service for Kubernetes. You may be wondering how that picture we just saw differs for GKE. Well, here it is:

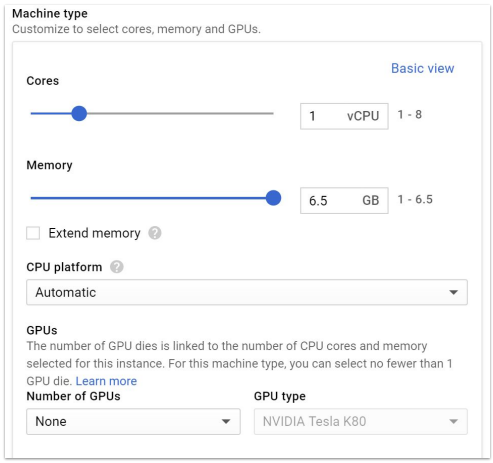
From the user's perspective, it's a lot simpler. GKE manages all the control plane components for us. It still exposes an IP address to which we send all of our Kubernetes API requests, but GKE takes responsibility for provisioning and managing all the control plane infrastructure behind it. It also abstracts away having a separate control plane. The responsibilities of the control plane are absorbed by Google Cloud, and you are not separately billed for your control plane.

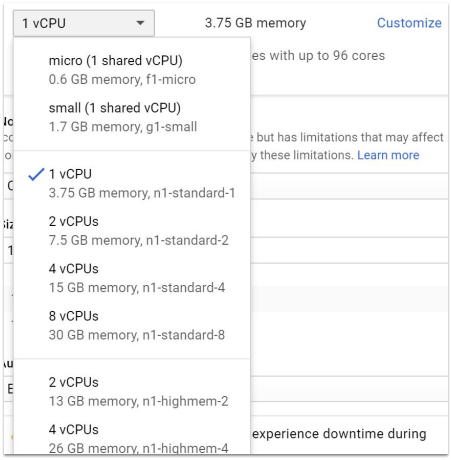


Now let's talk about nodes. In any Kubernetes environment, nodes are created externally by cluster administrators, not by Kubernetes itself.

GKE automates this process for you. It launches Compute Engine virtual machine instances and registers them as nodes. You can manage node settings directly from the Cloud Console. You are charged per second of allocated time for your nodes (not counting the control plane).

GKE: More about nodes





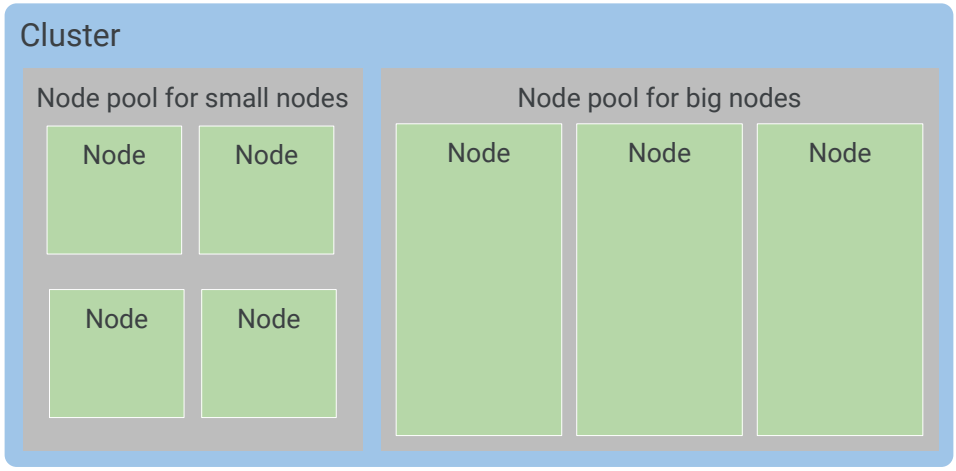
Google Cloud

Because nodes run on Compute Engine, you choose your node machine type when you create your cluster. By default, the node machine type is n1-standard-1, which providing 1 vCPU and 3.75 gigabytes of memory. Google Cloud offers a wide variety of Compute Engine options. At the time this course was developed, the generally available maximum was 96 vCPU cores. That's a moderately big virtual machine.

You can customize your nodes' number of cores and their memory capacity. You can select a CPU platform.

You can choose a baseline minimum CPU platform for the nodes or node pool. This allows you to improve node performance. GKE will never use a platform that is older than the CPU platform you specify, and if it picks a newer platform, the cost will be same as the specified platform.

Use node pools to manage different kinds of nodes



Google Cloud

You can also select multiple node machine types by creating multiple node pools. A node pool is a subset of nodes within a cluster that share a configuration, such as their amount of memory, or their CPU generation. Node pools also provide an easy way to ensure that workloads run on the right hardware within your cluster: you just label them with a desired node pool.

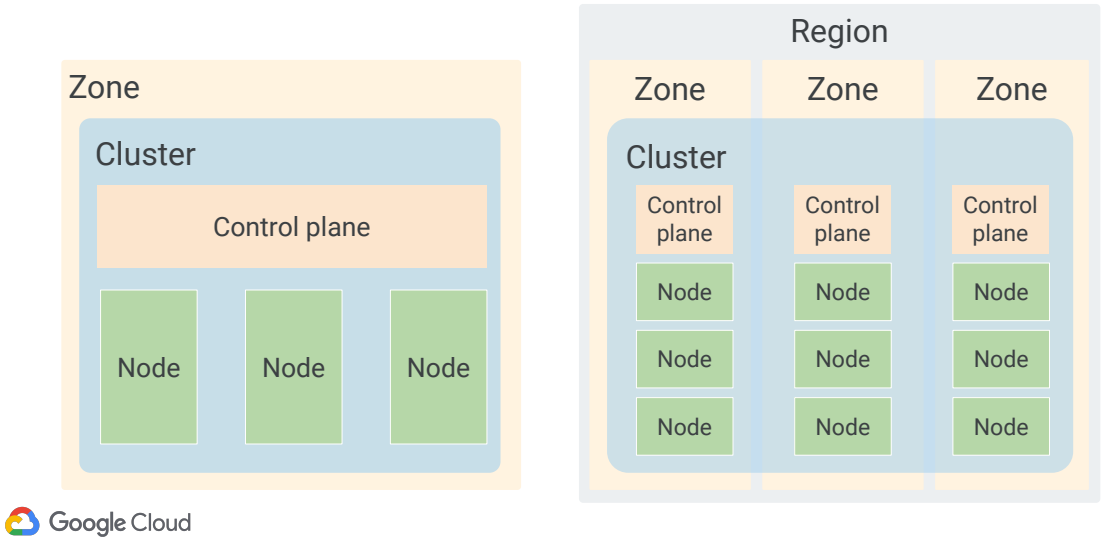
By the way, node pools are a GKE feature rather than a Kubernetes feature. You can build an analogous mechanism within open-source Kubernetes, but you would have to maintain it yourself.

You can enable automatic node upgrades, automatic node repairs, and cluster autoscaling at this node pool level.

Here's a word of caution. Some of each node's CPU and memory are needed to run the GKE and Kubernetes components that let it work as part of your cluster. So, for example, if you allocate nodes with 15 gigabytes of memory, not quite all of that 15 gigabytes will be available for use by Pods. This module has a documentation link that explains how much CPU and memory are reserved.

https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-architecture

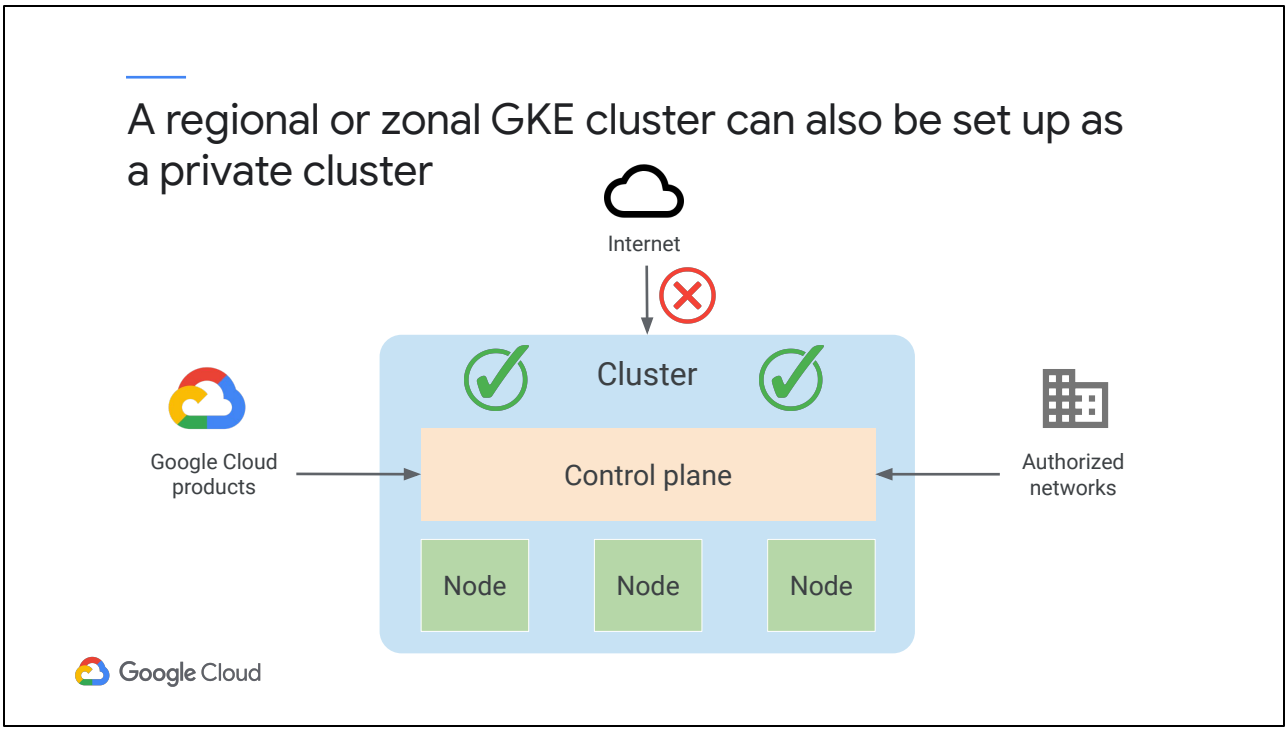
Zonal versus regional clusters



By default, a cluster launches in a single Google Cloud compute zone with three identical nodes, all in one node pool. The number of nodes can be changed during or after the creation of the cluster. Adding more nodes and deploying multiple replicas of an application will improve an application's availability. But only up to a point. What happens if the entire compute zone goes down?

You can address this concern by using a GKE regional cluster. Regional clusters have a single API endpoint for the cluster. However, its control planes and nodes are spread across multiple Compute Engine zones within a region.

Regional clusters ensure that the availability of the application is maintained across multiple zones in a single region. In addition, the availability of the control plane is also maintained so that both the application and management functionality can withstand the loss of one or more, but not all, zones. By default, a regional cluster is spread across 3 zones, each containing 1 control plane and 3 nodes. These numbers can be increased or decreased. For example, if you have five nodes in Zone 1, you will have exactly the same number of nodes in each of the other zones, for a total of 15 nodes. Once you build a zonal cluster, you can't convert it into a regional cluster, or vice versa.



The entire cluster (that is, the control plane and its nodes) are hidden from the public internet.

Cluster control planes can be accessed by Google Cloud products, such as Cloud Logging or Cloud Monitoring, through an internal IP address.

They can also be accessed by authorized networks through an external IP address. Authorized networks are basically IP address ranges that are trusted to access the control plane. In addition, nodes can have limited outbound access through Private Google Access, which allows them to communicate with other Google Cloud services. For example, nodes can pull container images from Container Registry without needing external IP addresses.

Agenda

Kubernetes Concepts

Kubernetes Components

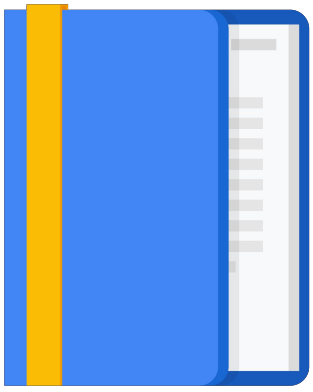
Google Kubernetes Engine Concepts

Object Management

Lab: Deploying Google Kubernetes Engine

Migrate for Anthos

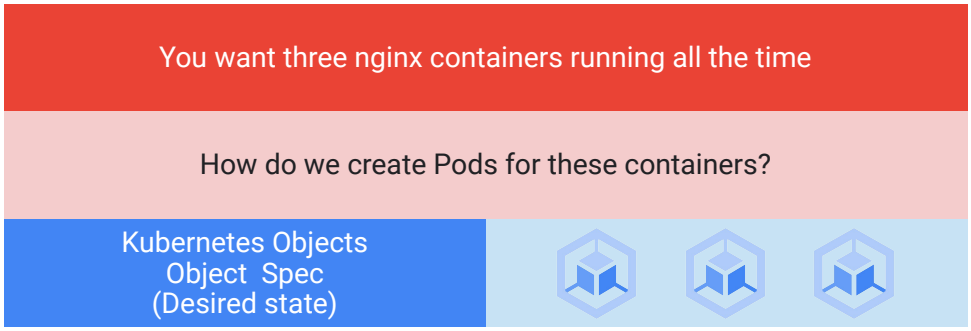
Summary



Google Cloud

Finally, we'll discuss Kubernetes object management. All Kubernetes objects are identified by a unique name and a unique identifier.

Running three nginx containers



Google Cloud

Let's return once again to our example, in which we want three nginx Web servers running all the time.

Well, the simplest way would before us to declare three Pod objects and specify their state: that, for each, a Pod must be created and an nginx container image must be used. Let's see how we declare this.

Objects are defined in a YAML file

```yaml
apiVersion: apps/v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name:  nginx
    image: nginx:latest
```

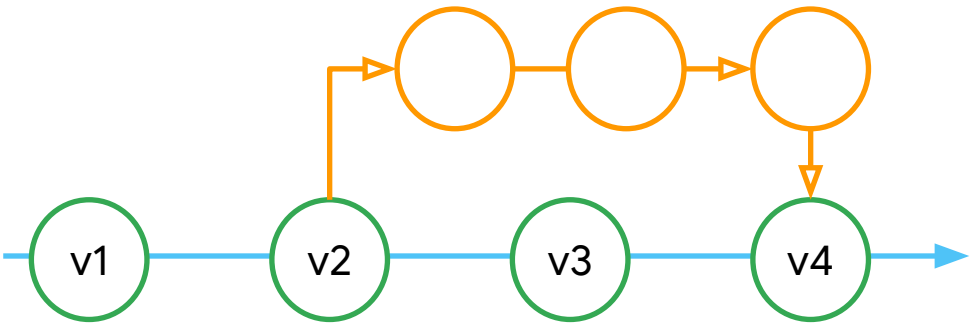
Google Cloud

You define the objects you want Kubernetes to create and maintain with manifest files. These are ordinary text files. You may write them in YAML or JSON format. YAML is more human-readable and less tedious to edit. This YAML file defines a desired state for a pod: its name and a specific container image for it to run.

Your manifest files have certain required fields. ApiVersion describes which Kubernetes API version is used to create the object. The Kubernetes protocol is versioned so as to help maintain backwards compatibility.

Kind identifies the object you want (in this case a Pod) and Metadata helps identify the object using Name, Unique ID, and an optional Namespace. You can define several related objects in the same YAML file, and it is a best practice to do so. One file is often easier to manage than several.

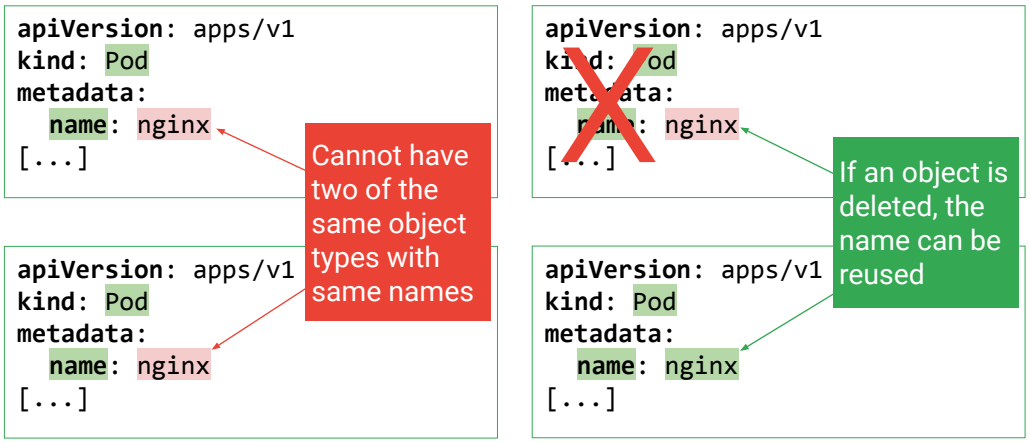
Best practice tip: Use version control on YAML files



Google Cloud

Another, even more important tip: You should save your YAML files in version-controlled repositories. This practice makes it easier to track and manage changes and to back-out those changes when necessary. It's also a big help when you need to recreate or restore a cluster. Many Google Cloud customers use Cloud Source Repositories for this purpose, because that service lets them control the permissions of those files in the same way as their other Google Cloud resources.

All objects are identified by a name



Google Cloud

When you create a Kubernetes object, you name it with a string. Names must be unique. Only one object of a particular kind can have a particular name at the same time in the same Kubernetes namespace. However, if an object is deleted, its name can be reused. Alphanumeric characters, hyphens, and periods are allowed in the names, with a maximum character length of 253.

All objects are assigned a unique identifier (UID) by Kubernetes

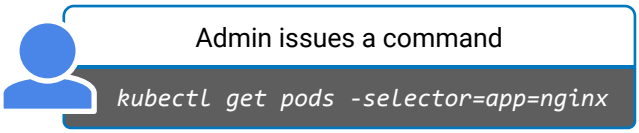
```
apiVersion: apps/v1
kind: Pod
metadata:
  name: nginx
    uid: 4dd474fn-f389-11f8-b38c-42010a8009z7
[...]
```

Google Cloud

Every object created throughout the life of a cluster has a unique UID generated by Kubernetes. This means that no two objects will have same UID throughout the life of a cluster.

Labels can be matched by label selectors

```
apiVersion: apps/v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
    env: dev
    stack: frontend
spec:
  selector:
    matchLabels
    app: nginx
```



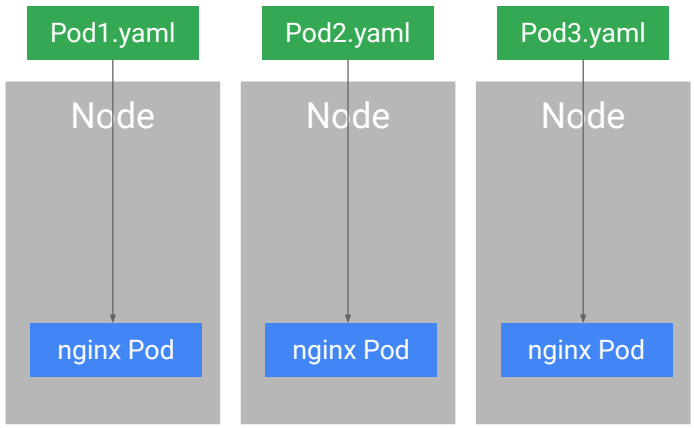
Google Cloud

Labels are key-value pairs with which you tag your objects during or after their creation. Labels help you identify and organize objects and subsets of objects. For example, you could create a label called “app” and give as its value the application of which this object is a part.

In this simple example, a Pod object is labeled with three different key-values: its application, its environment, and which stack it forms a part of.

Various contexts offer ways to select Kubernetes resources by their labels. Here’s an example of using it to show all the pods that contain a label called “app” with a value of “nginx.” Label selectors are very expressive. You can ask for all the resources that have a certain value for a label, all those that don’t have a certain value, or even all those that have a value in a set you supply.

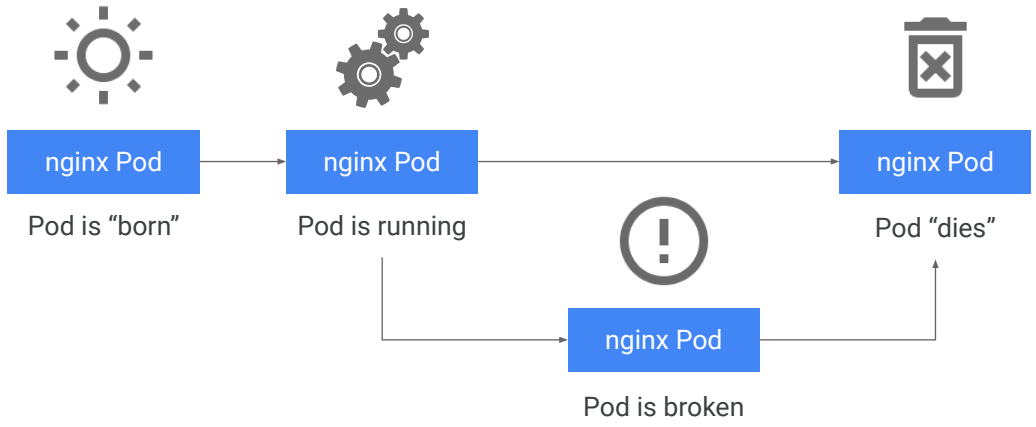
A workload is spread evenly across available nodes by default



Google Cloud

So one way to bring three nginx Web servers into being would be to declare three Pod objects, each with its own section of YAML. Kubernetes's default scheduling algorithm prefers to spread the workload evenly across the nodes available to it, so we'd get a situation like this one. Looks good, doesn't it? Maybe not. Suppose I want 200 more nginx instances. Managing 200 more sections of YAML sounds very inconvenient.

Pods have a life cycle



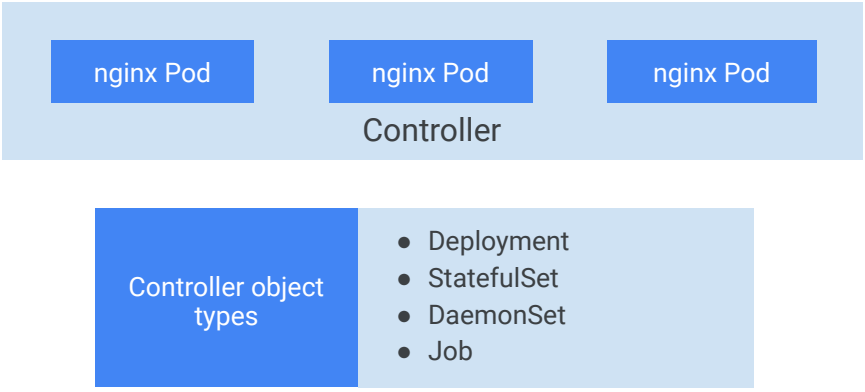
Google Cloud

Here’s another problem: Pods don’t heal or repair themselves, and are not meant to run forever. They are designed to be ephemeral and disposable.

For these reasons, there are better ways to manage what you run in Kubernetes than specifying individual Pods. You need a setup like this to maintain an application’s high availability along with horizontal scaling.

So how do you tell Kubernetes to maintain the desired state of three nginx containers?

Pods and Controller Objects



Google Cloud

We can instead declare a controller object whose job is to manage the state of the Pods. Some examples of these objects: Deployments, StatefulSets, DaemonSets, and Jobs.

Deployments are a great choice for long-lived software components

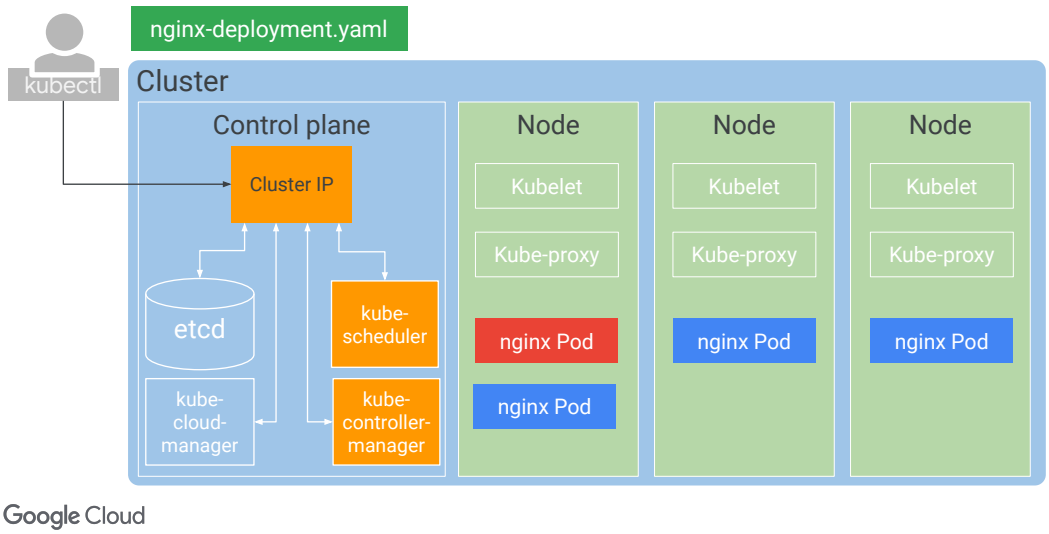
You want three nginx containers running all the time

How does Kubernetes maintain 3 nginx containers at any given time?

Google Cloud

Deployments are a great choice for long-lived software components like Web servers, especially when we want to manage them as a group.

A Deployment maintains the desired state



Google Cloud

In our example, when Kube-scheduler schedules Pods for a Deployment, it notifies the Kube-APIserver.
These changes are constantly monitored by controllers—especially by the Deployment controller. The practical effect of the Deployment controller is to monitor and maintain 3 nginx Pods.

The Deployment controller creates a child object, a ReplicaSet, to launch the desired Pods. If one of these Pods fails, the ReplicaSet controller will recognize the difference between the current state and the desired state and will try to fix it by launching a new Pod. Instead of using multiple yaml manifests or files for each Pod, you used a single Deployment yaml to launch 3 replicas of the same container.

Deployments ensure that sets of Pods are running

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
```

Google Cloud

A Deployment ensures that a defined set of Pods is running at any given time.

Within its object spec, you specify how many replica Pods you want, how Pods should run, which containers should run within these Pods, and which Volumes should be mounted. Based on these templates, controllers maintain the Pod's desired state within a cluster. Controllers are discussed later in the course.

Deployments can also do a lot more than this, which you will see later in the course.

Allocating resource quotas

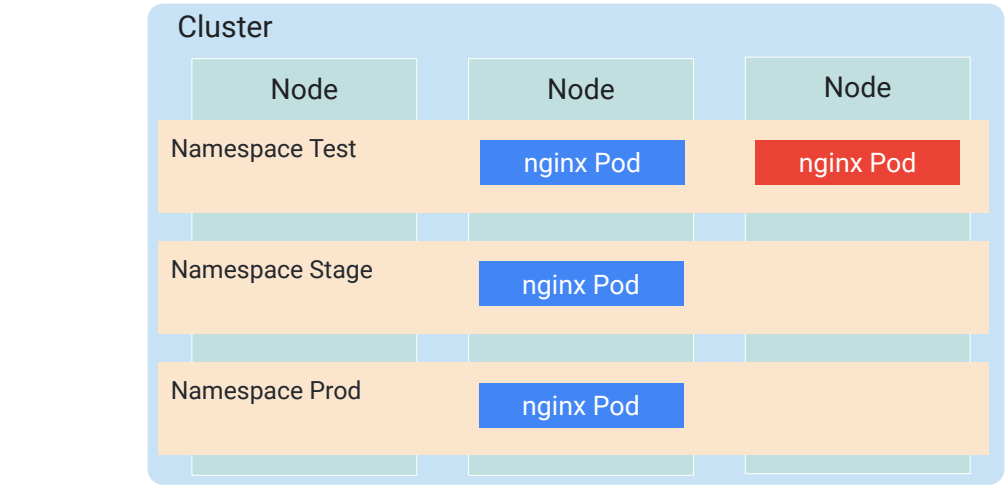
Multiple activities run on a single cluster

How can I allocate resource quotas?

Google Cloud

It's very probable that you'll be using a single cluster for multiple projects. At the same time, it's essential to maintain resource quotas based on projects or teams. By the way: when I say "projects" here, I mean projects in the informal sense of the word: things you and your colleagues are working on. Each Kubernetes cluster is associated with a Google Cloud project, in the formal sense of the word "project", and that's how IAM policies apply to it and how you're billed for it.

Namespaces provide scope for naming resources



Google Cloud

So how do you keep everybody's work on your cluster tidy and organized? Kubernetes allows you to abstract a single physical cluster into multiple virtual clusters known as *namespaces*. Namespaces provide scope for naming resources such as Pods, Deployments, and controllers.

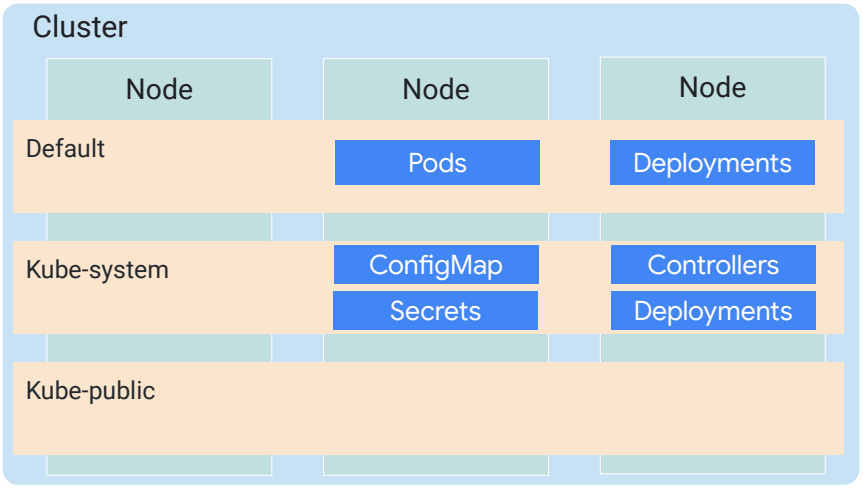
As you can see in this example, there are three namespaces in this cluster: test, stage, and prod.

Remember that you cannot have duplicate object names in the *same* namespace. You can create three Pods with the same name (nginx), but only if they don't share the same namespace. If you attempt to create another Pod with same the name 'nginx Pod' in namespace "test", you won't be allowed. Object names need only be unique within a namespace, not across all namespaces. Namespaces also let you implement resource quotas across the cluster. These quotas define limits for resource consumption within a namespace. They're not the same as your Google Cloud quotas, which we discussed in an earlier module. These quotas apply specifically to the Kubernetes cluster they're defined on.

You're not require to use namespaces for your day-to-day management; you can also use labels. Still, namespaces are a valuable tool. Suppose you want to spin up a copy of a deployment as a quick test. Doing so in a new namespace makes it easy and

free of name collisions.

There are three initial namespaces in a cluster



Google Cloud

The first is a default namespace, for objects with no other namespace defined. Your workload resources will use this namespace by default.

Then there is the kube-system namespace for objects created by the Kubernetes system itself. When you use the kubectl command, by default items in the kube-system namespace are excluded, but you can choose to view its contents explicitly.

The third namespace is the kube-public namespace for objects that are publicly readable to all users. kube-public is a tool for disseminating information to everything running in a cluster. You're not required to use it, but it can come in handy, especially when everything running in a cluster is related to the same goal and needs information in common.

Best practice tip: namespace-neutral YAML

Most flexible:

```
kubectl -n demo apply -f mypod.yaml
```

Legal but less flexible:

```
apiVersion: v1
kind: Pod
metadata:
    name: mypod
  namespaces: demo
```

Google Cloud

You can apply a resource to a namespace when creating it, using a command-line namespace flag. Or, you can specify a namespace in the YAML file for the resource. Whenever possible, apply namespaces at the command line level. This practice makes your YAML files more flexible. For example, someday you might want to create two identical but completely independent instances of one of your deployments, each in its own namespace. This could be the case if you want to deploy into a separate namespaces for testing before deploying into production. This is difficult if you have chosen to embed namespace names in your YAML files.

Agenda

Kubernetes Concepts

Kubernetes Components

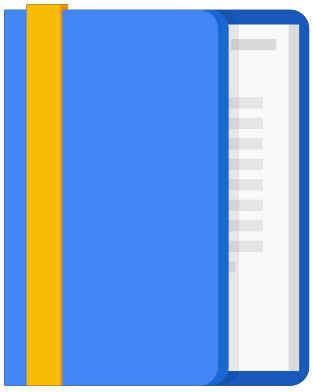
Google Kubernetes Engine Concepts

Object Management

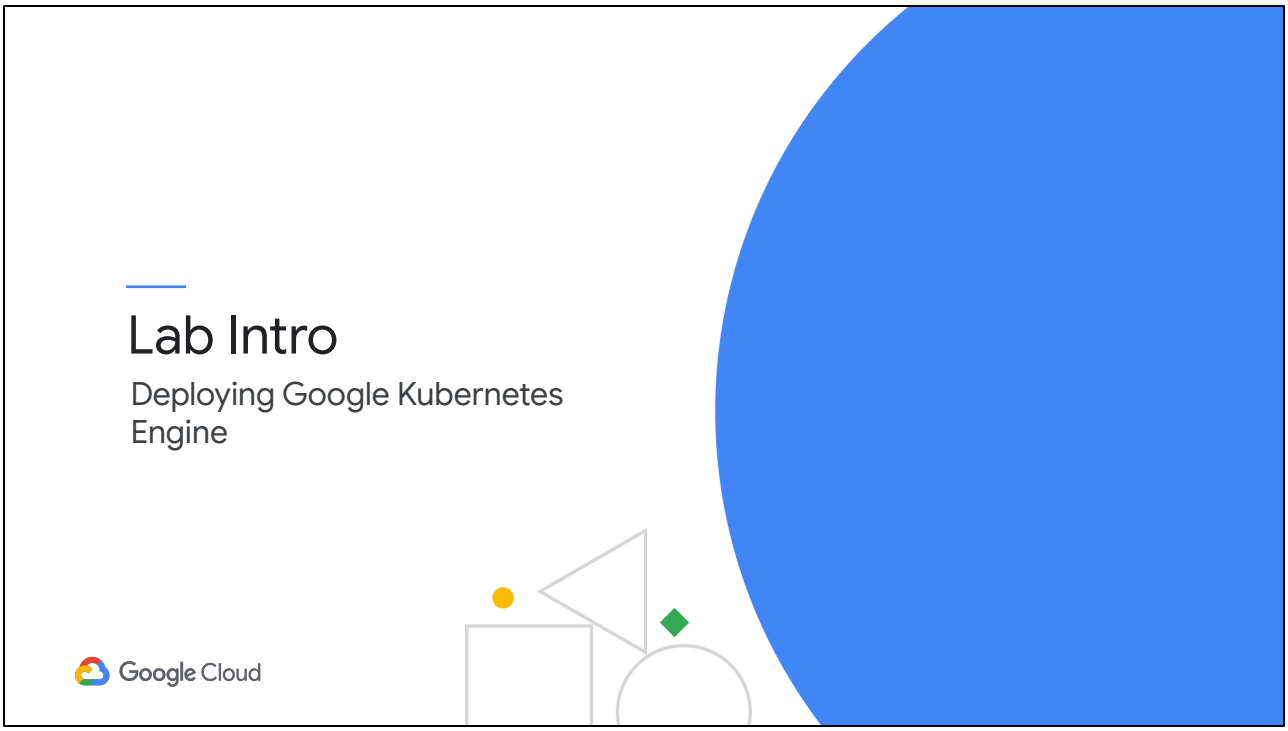
Lab: Deploying Google Kubernetes Engine

Migrate for Anthos

Summary



Google Cloud



In this lab, you'll build and use GKE clusters and deploy a sample Pod. The tasks that you'll learn to perform include using the Cloud Console to build and manipulate GKE clusters, deploy a Pod, and examine the cluster and Pods.

Agenda

Kubernetes Concepts

Kubernetes Components

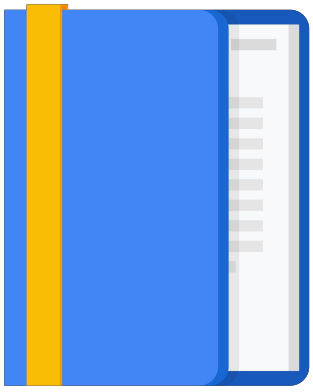
Google Kubernetes Engine Concepts

Object Management

Lab: Deploying Google Kubernetes Engine

Migrate for Anthos

Summary



Google Cloud

Migrate for Anthos moves VMs to containers

- Move and convert workloads into containers.
- Workloads can start as physical servers or VMs.
- Moves workload compute to container immediately (<10 min).
- Data can be migrated all at once or "streamed" to the cloud until the app is live in the cloud.

Google Cloud

Migrate for Anthos moves your existing applications into a Kubernetes environment and the best thing about this is that the process is automated!

Your workloads can be on-premises or in other cloud providers.

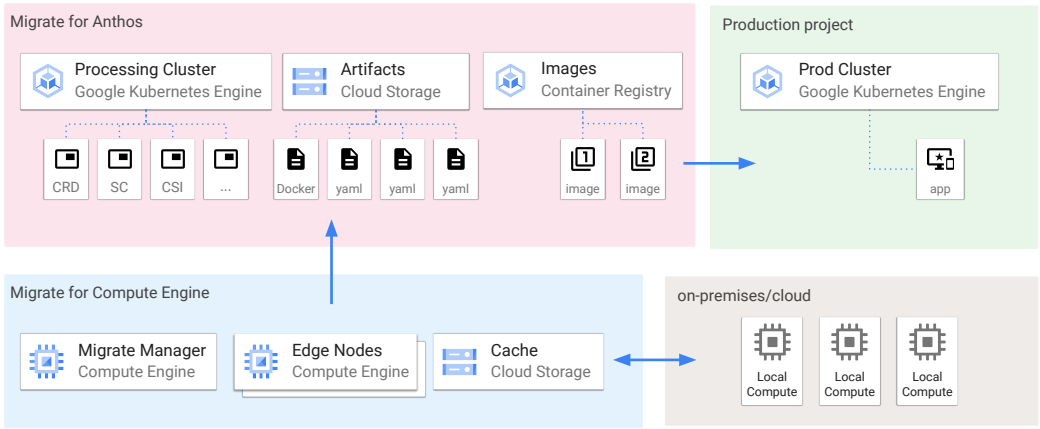
You know that Migrate for Anthos is automated, but it is also very fast.

Most migrations are completed in less than 10 minutes.

And you have the choice to migrate your applications data in one move, or to stream it to the cloud until the app is live.

Let's have a quick overview of what happens in a migration.

A migration requires an architecture to be built



Google Cloud

First let's inspect the architecture required for a migration

The first step is to allow Migrate for Compute Engine to create the pipeline for streaming/migrating data from on-premises or cloud systems into Google Cloud. Migrate for Compute Engine is a tool that allows you to bring your existing applications into VMs on Google Cloud.

Migrate for Anthos is then installed on a GKE processing cluster, and is composed of many Kubernetes resources.

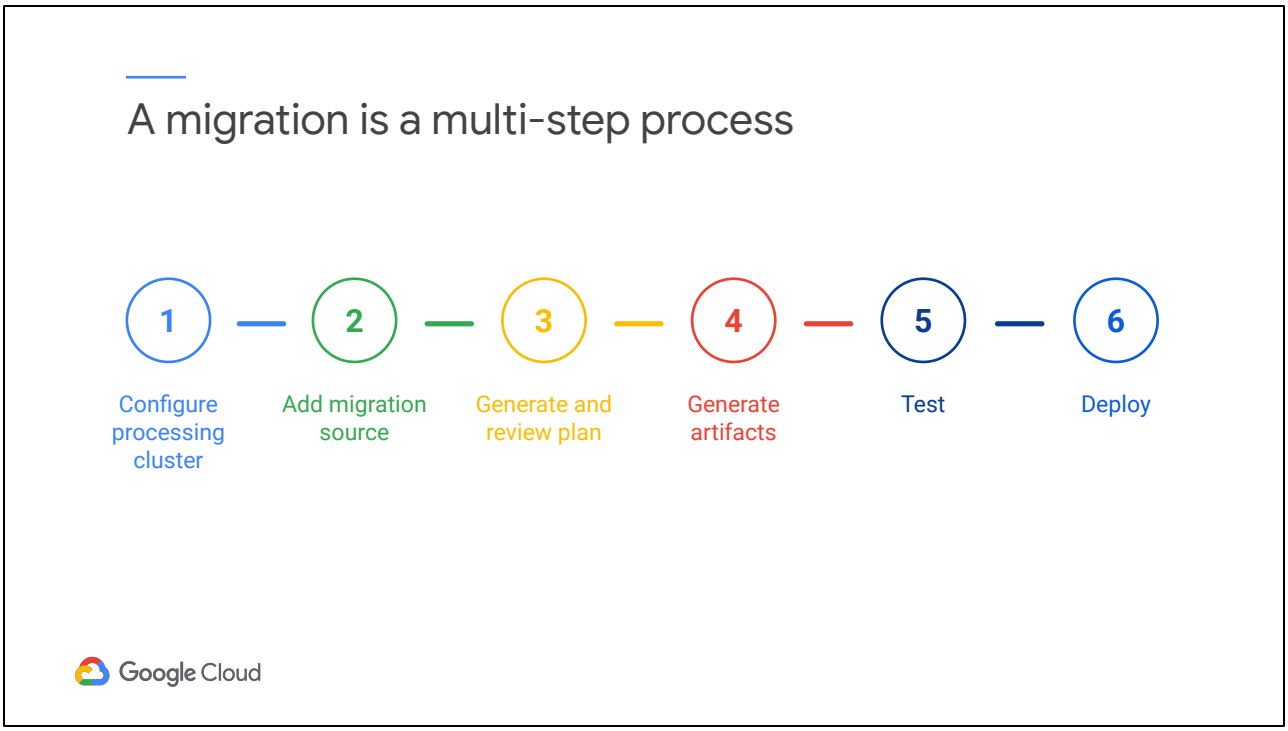
Migrate for Anthos is used to generate the deployment artifacts.

Some of these artifacts, like the Kubernetes configurations and the Docker file, are used to create the VM-wrapping container. This container goes into Cloud Storage.

The container images themselves are stored in Container Registry.

After the deployment assets are created, they can be used to deploy your application into a target cluster.

You simply apply the generated configuration, and it creates the necessary Kubernetes elements on the target cluster.



Now that you have seen the architecture required for a migration, let's look at what happens when you migrate an application using Migrate for Anthos.

1. First you need to create the processing cluster. After that, you install the Migrate for Anthos components onto that cluster.

2. Next you need to add a migration source.
   You can migrate from VMware, AWS, Azure, or Google Cloud.

3. You will need to create a migration object with the details of the migration that you are performing.
   This will generate a plan template for you in a YAML file.
   You may need to alter this configuration file to create the level of customization you desire.
   When the plan is ready, you will need to generate the artifacts for the migration.
   This means generating the container images of your applications and the YAML files required for the deployment.

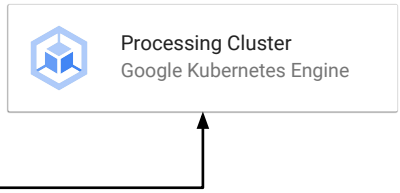
4. After your migration artifacts have been generated, they need to be tested.
   Both the container images and the deployment will be tested at this stage.

5. Finally, if the tests are successful, you can use the generated artifacts to deploy your application to your production clusters.

https://cloud.google.com/migrate/anthos/docs/migration-journey

Migrate for Anthos requires a processing cluster

```
gcloud container --project $PROJECT_ID \
clusters create $CLUSTER_NAME \
--zone $CLUSTER_ZONE \
--username "admin" \
--cluster-version 1.14 \
--machine-type "n1-standard-4" \
--image-type "UBUNTU" \
--num-nodes 1 \
--enable-stackdriver-kubernetes \
--scopes "cloud-platform" \
--enable-ip-alias \
--tags="http-server"
```



Google Cloud

Let's go through an example to see what happens at each stage.

The first thing you will do is set up the processing cluster.

Before you run the command on screen, you need to make sure that you are a GKE admin to set up the cluster.

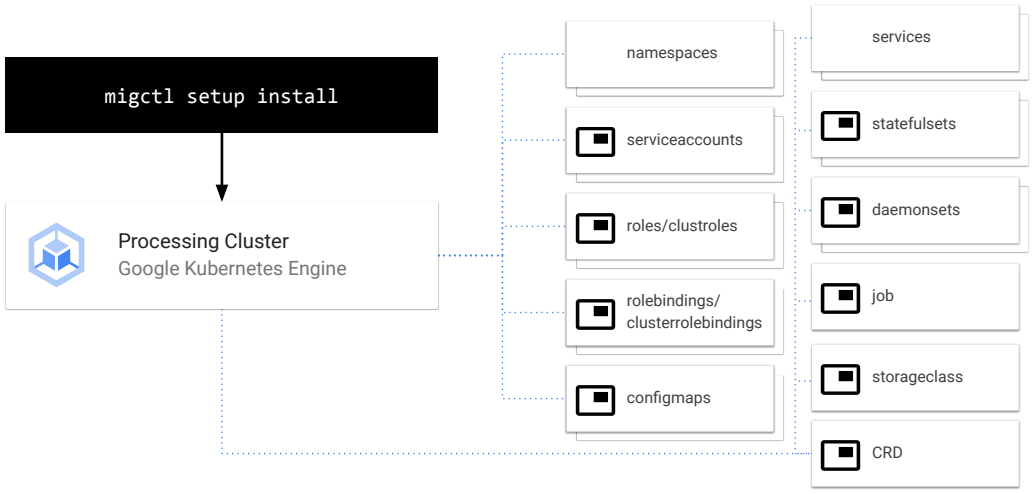
You also must have firewall rules in place that allow communications between Migrate for Anthos and Migrate for Compute Engine.

After that is done, you can create the processing cluster.

The example command enables a VPC-native cluster.

https://cloud.google.com/migrate/anthos/docs/configuring-a-cluster

Installing Migrate for Anthos uses `migctl`



Google Cloud

When the processing cluster is up and running, you need to install Migrate for Anthos using the **migctl** command. This command installs all of the required Kubernetes resources onto the processing cluster for the migration.

Adding a source enables migrations from a specific environment

```
migctl source create ce my-ce-src --project my-project --zone zone
```

Google Cloud

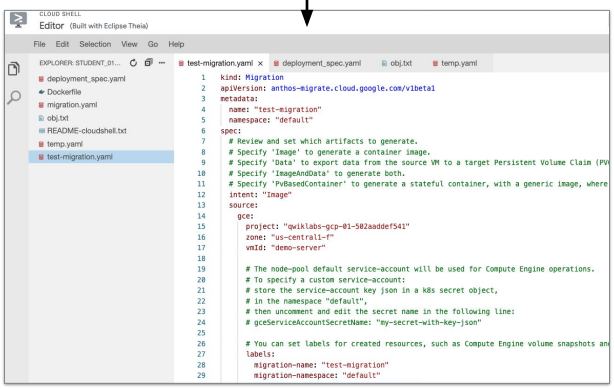
The **migctl source create** command specifies the location of the application to migrate.

The example on screen is for migrating from Google Compute Engine.

If you are migrating from a VMware backend or another cloud provider, you need to install some additional packages.

Creating a migration generates a migration plan

```
migctl migration create test-migration --source my-ce-src --vm-id my-id --intent Image
```



Google Cloud

Now that the infrastructure elements are set up, the next step is to create a migration plan.

The **migctl migration create** command will create a migration plan.

This command will define the migration resources that will be created on cluster.

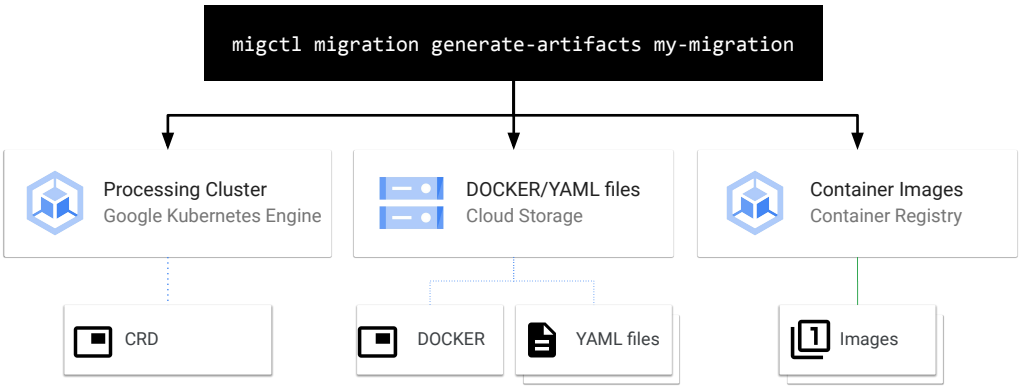
You identify the source VM and what data to exclude in the migration.

You can also specify what migration intents you want.

You can specify the following intents:

- Image
- ImageAndData
- Data
- PvBasedContainer

Executing a migration generates resources and artifacts



Google Cloud

After creating a migration plan, you need to generate the artifacts for the migration. The **migctl migration generate artifacts** command on screen will start this process.

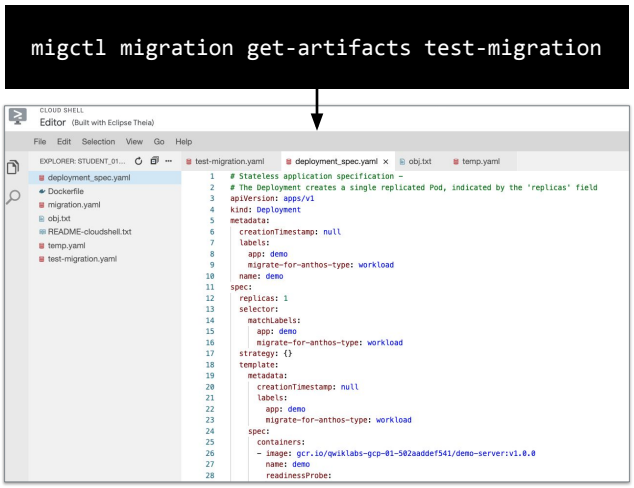
This process will first copy files and directories representing the VM to a container image registry as images.

Migrate for Anthos creates two images: a runnable image for deployment to another cluster and a non-runnable image layer that can be used to update the container image in the future.

Next Migrate for Anthos with generate configuration YAML files that you can use to deploy the VM to another GKE cluster.

These are copied into a Cloud Storage bucket as an intermediate location.

Deployment files typically need modification



Google Cloud

You run the **migctl migration get-artifacts** to download the YAML configuration files generated from the last step.

The YAML configuration defines resources to deploy, such as:

- Are you creating a Deployment or a StatefulSet?
- Is the deployment a Headless service?
- Are you using PersistentVolumes or PersistentVolumeClaims?

You can edit the YAML file to customize the deployment. Examples of customizations include:

- To enable load-balancing
- To allow ingress
- To define disk size

https://cloud.google.com/migrate/anthos/docs/review-deployment-files

Apply the configuration to deploy the workload

```
kubectl apply -f deployment_spec.yaml
```

Google Cloud

Finally, you run the **kubectl apply** command to deploy the defined spec.

https://cloud.google.com/migrate/anthos/docs/review-deployment-files

Agenda

Kubernetes Concepts

Kubernetes Components

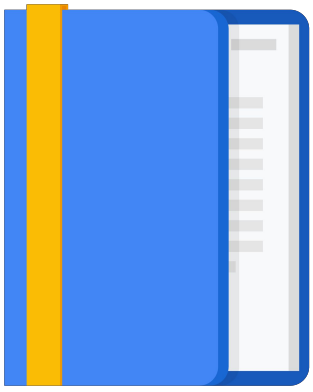
Google Kubernetes Engine Concepts

Object Management

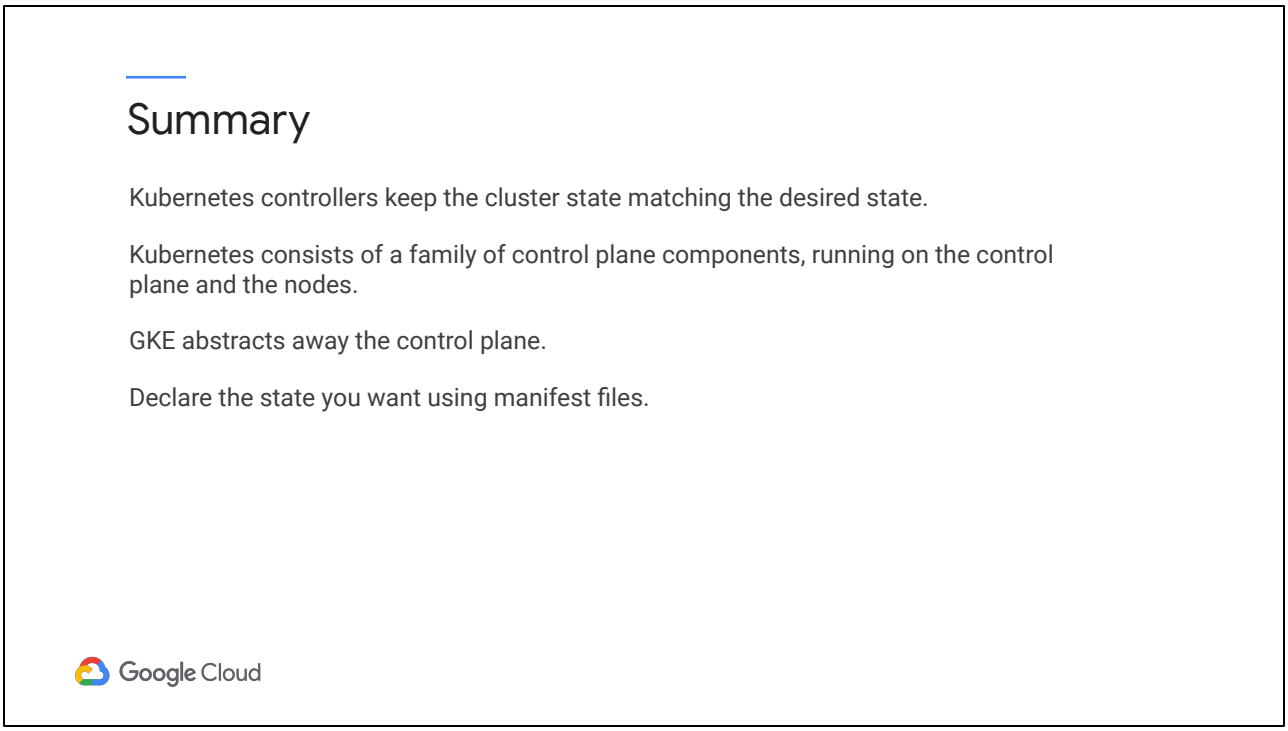
Lab: Deploying Google Kubernetes Engine

Migrate for Anthos

Summary



Google Cloud



This concludes the Kubernetes Architecture module.

In this module, you learned about the Kubernetes operating philosophy. Every item under Kubernetes's control is represented by an object, and Kubernetes tries to keep the state of its cluster matching the state you have declared that you want.

You learned about the control plane components that make up Kubernetes. You learned about kube-apiserver, which is the point of control for your Kubernetes cluster. And you learned about kubelet, which is your cluster's agent on each node.

And you also learned that a GKE-managed cluster implements the control plane for you, behind the scenes, and that you are not charged separately for it.

To tell Kubernetes what you want the state of your cluster to be, create manifest files. Typically, you'll build these files in YAML format. These files name and describe the objects you want Kubernetes to keep alive and healthy. They also document the desired state of your cluster, so you should keep these files in a source control system.


Google Cloud