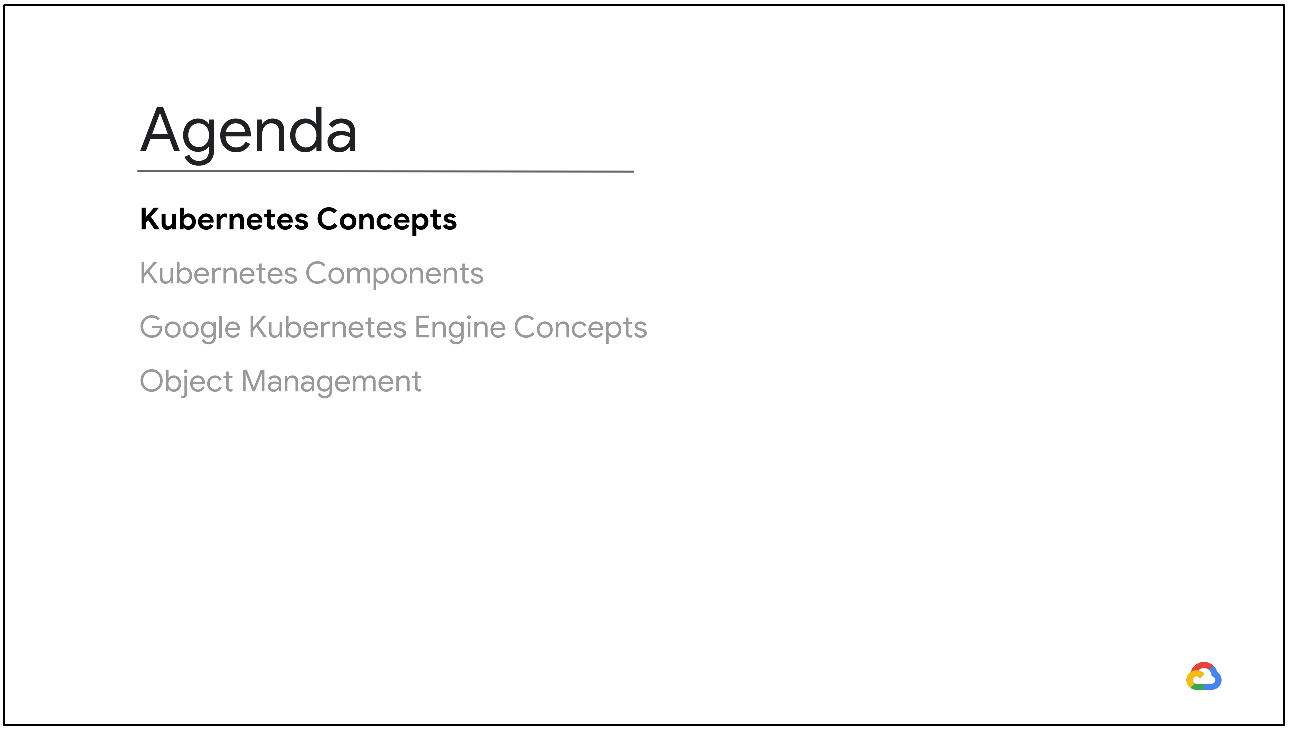


Agenda

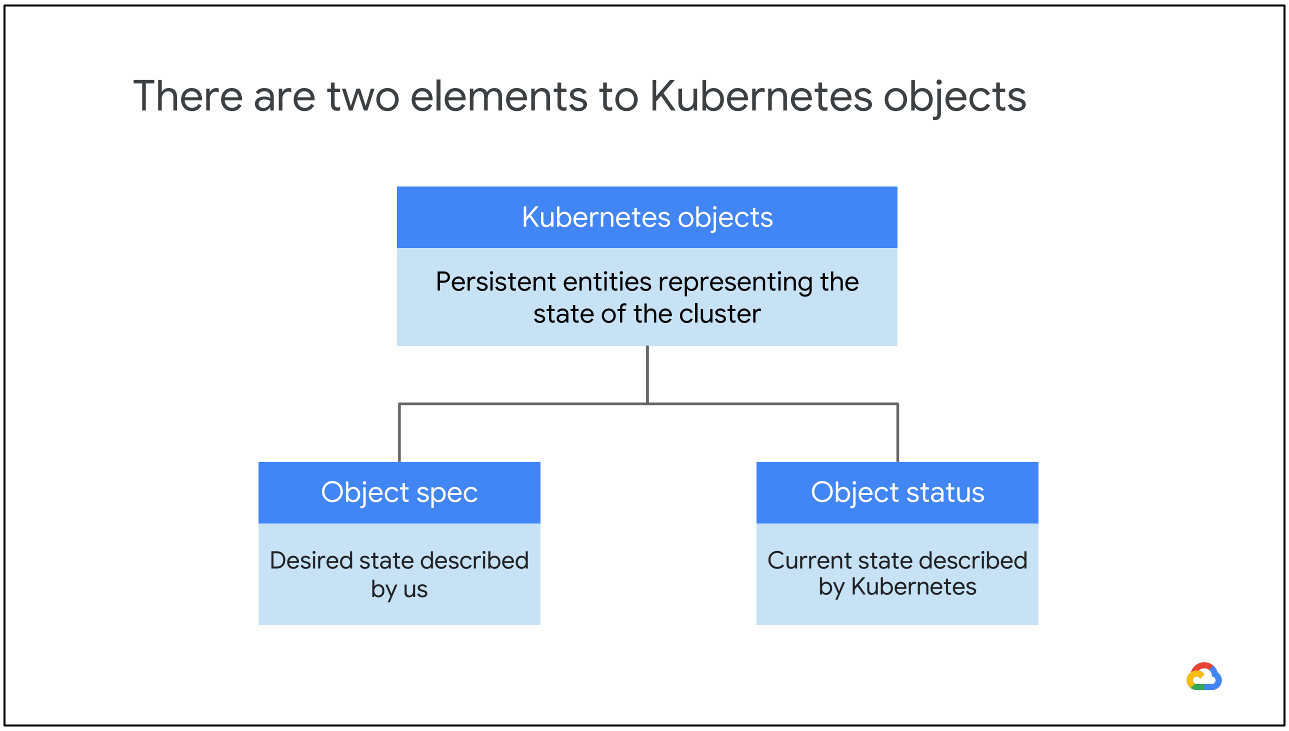
Kubernetes Concepts

Kubernetes Components

Google Kubernetes Engine Concepts

Object Management

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."



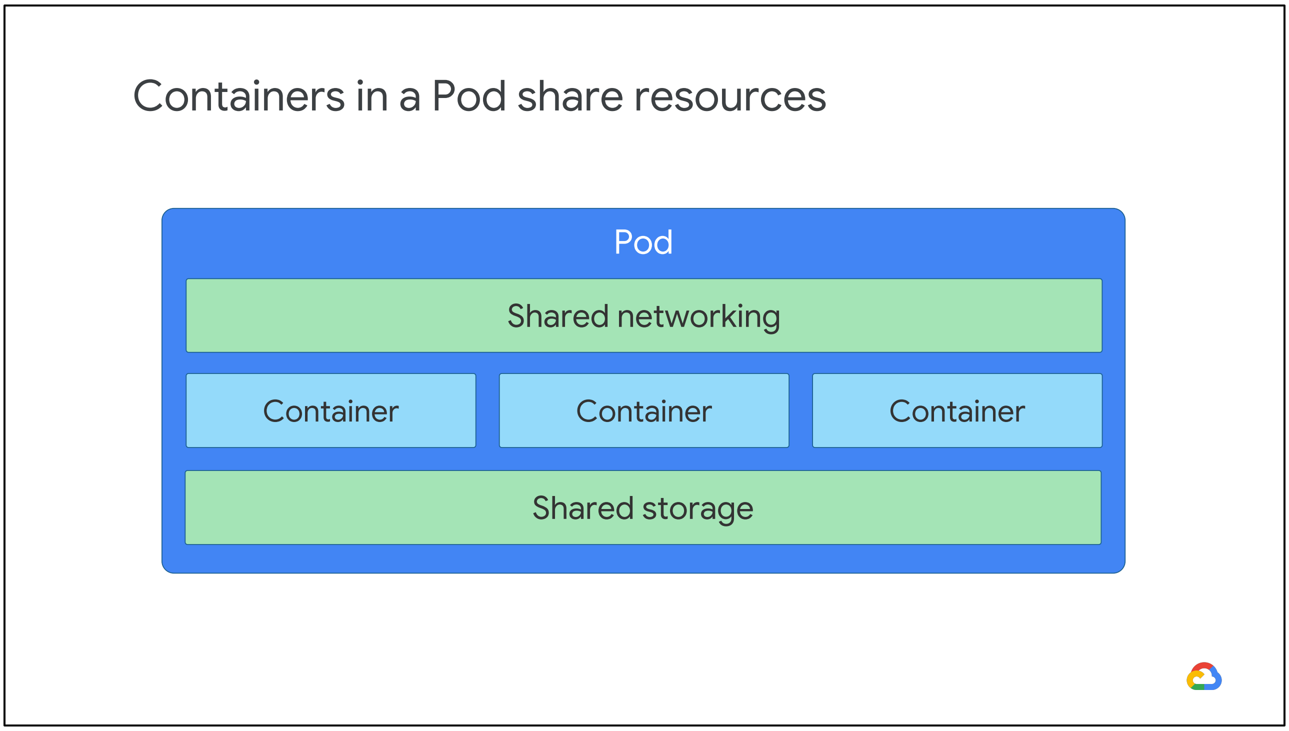
There are two elements to Kubernetes objects

Kubernetes objects
Persistent entities representing the state of the cluster
Object spec
Desired state described by us
Object status
Current state described by Kubernetes

Formally, a Kubernetes object is defined as a persistent entity that represent 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.

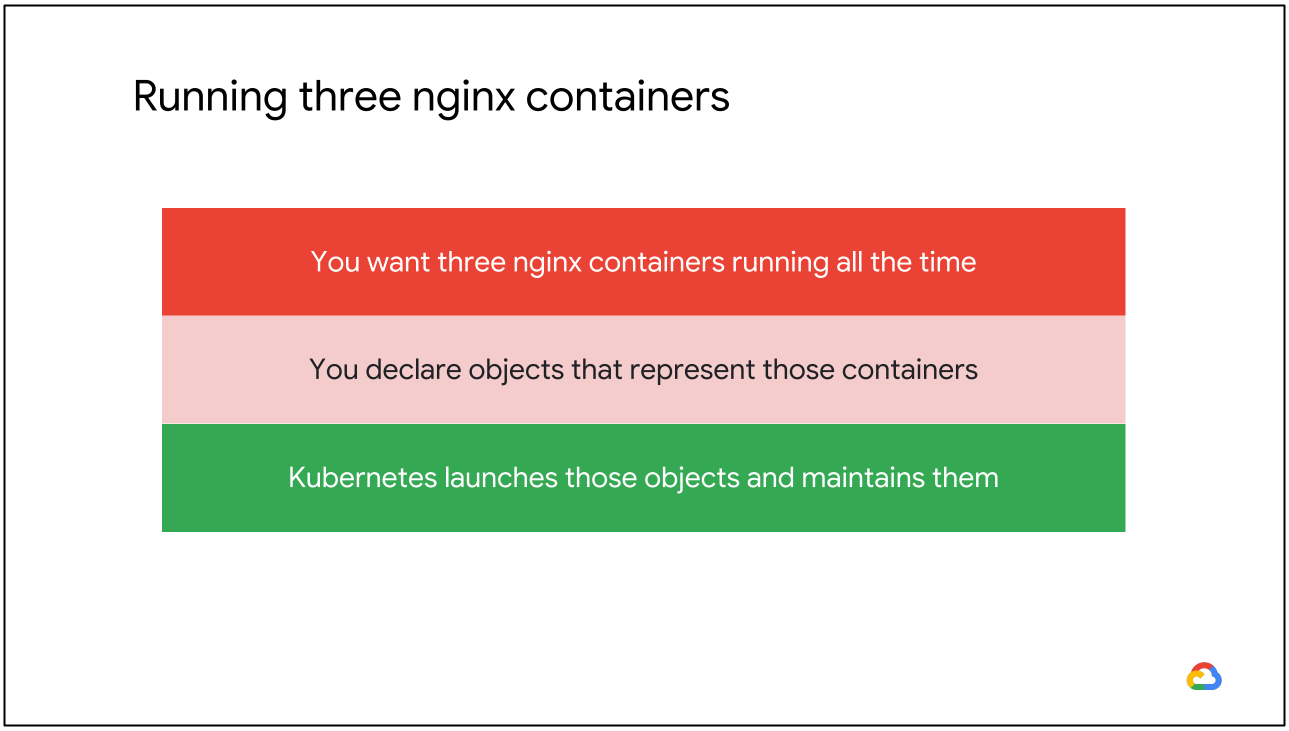


Containers in a Pod share resources

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. (By the way, later in this specialization, you’ll learn how Pods can share storage with one another, not just within a single Pod.)



Running three nginx containers

You want three nginx containers running all the time

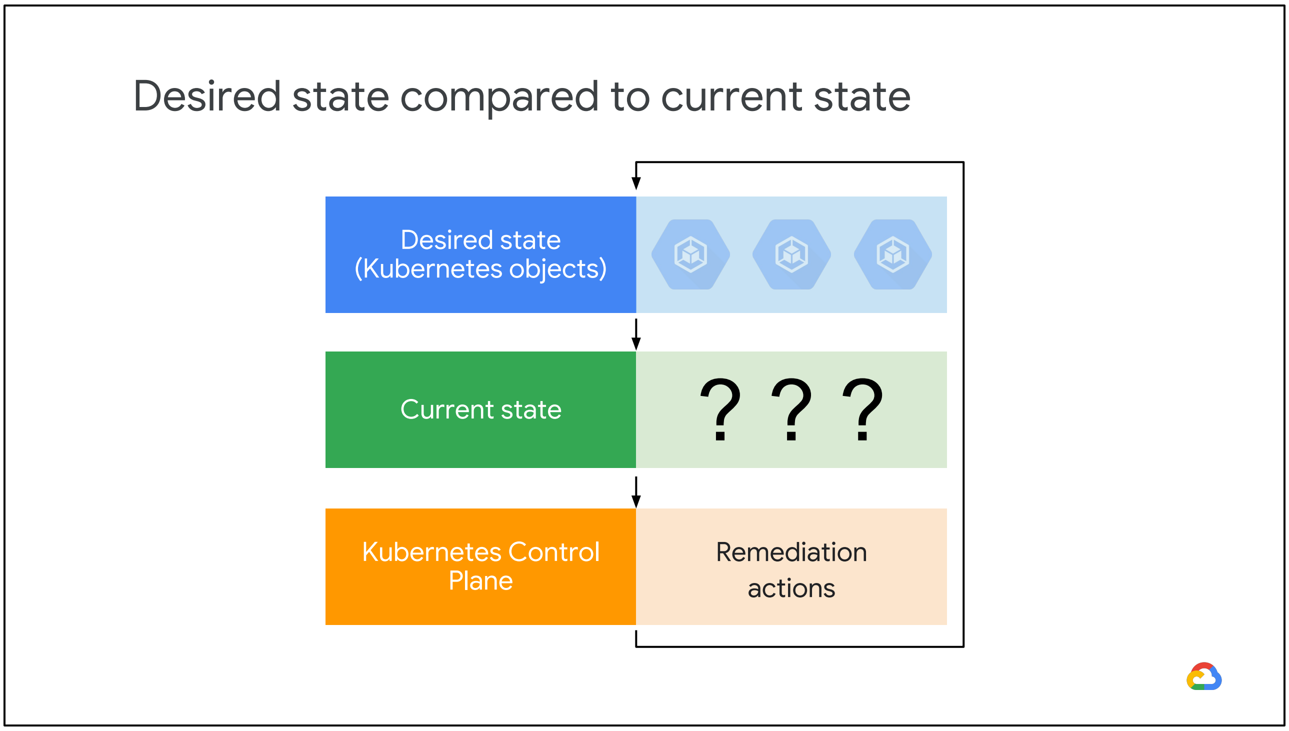
You declare objects that represent those containers

Kubernetes launches those objects and maintains them

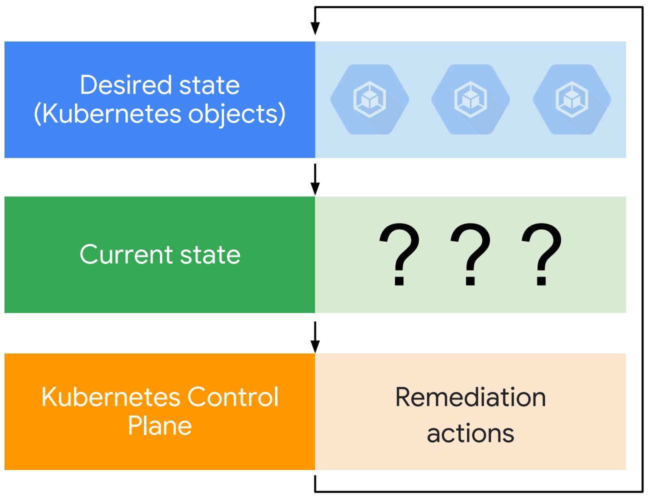
Let’s consider a simple example where you want three instances of the nginx Web server, each in its own container, running all the time.

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.

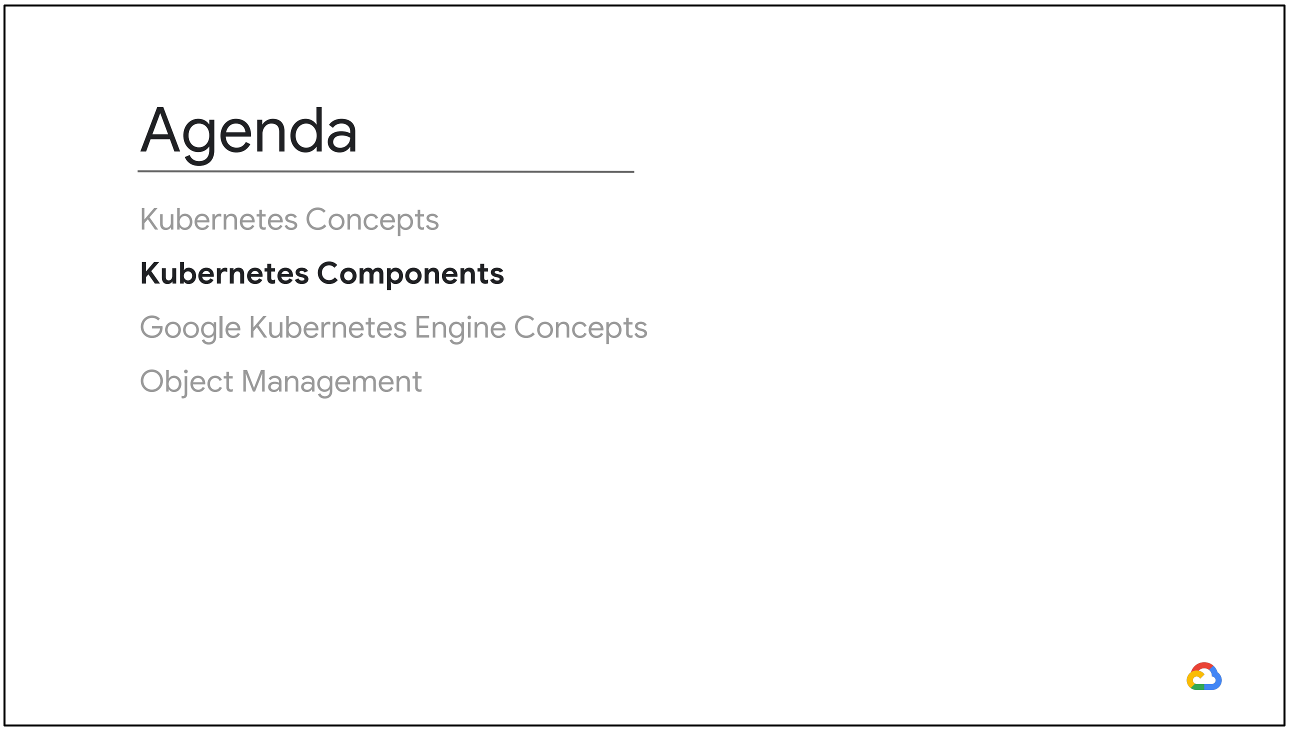
Now it is Kubernetes’s job to launch those Pods and keep them in existence. Be careful: Pods are not self-healing. If we want to keep all our nginx Web servers not just in existence but also working together as a team, we might want to ask for them using a more sophisticated Kind of object. I’ll tell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



Desired state compared to current state



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

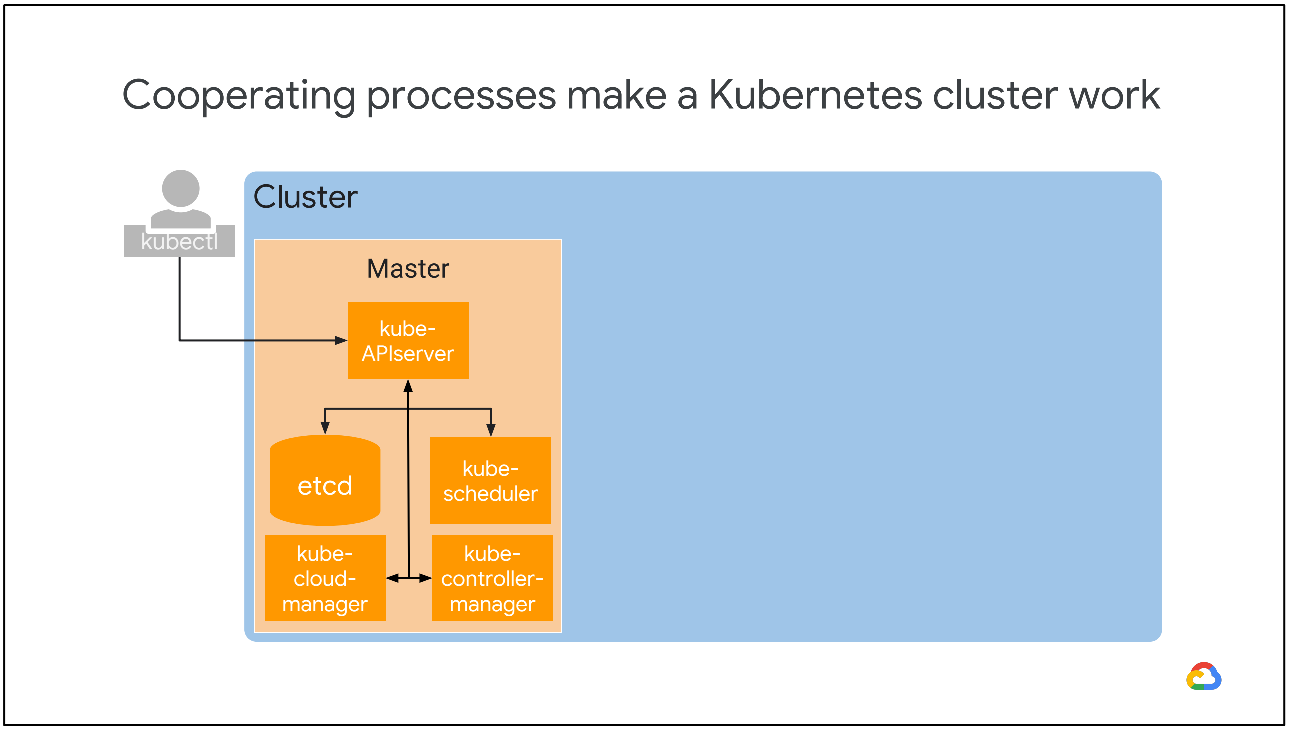
Kubernetes Concepts

Kubernetes Components

Google Kubernetes Engine Concepts

Object Management

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.



Cooperating processes make a Kubernetes cluster work

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 “master,” and the others are called simply “nodes.” The job of the nodes is to run Pods. The job of the master is to coordinate the entire cluster. We will meet its control-plane components first.

Several critical Kubernetes components run on the master. 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.

In this specialization, 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.

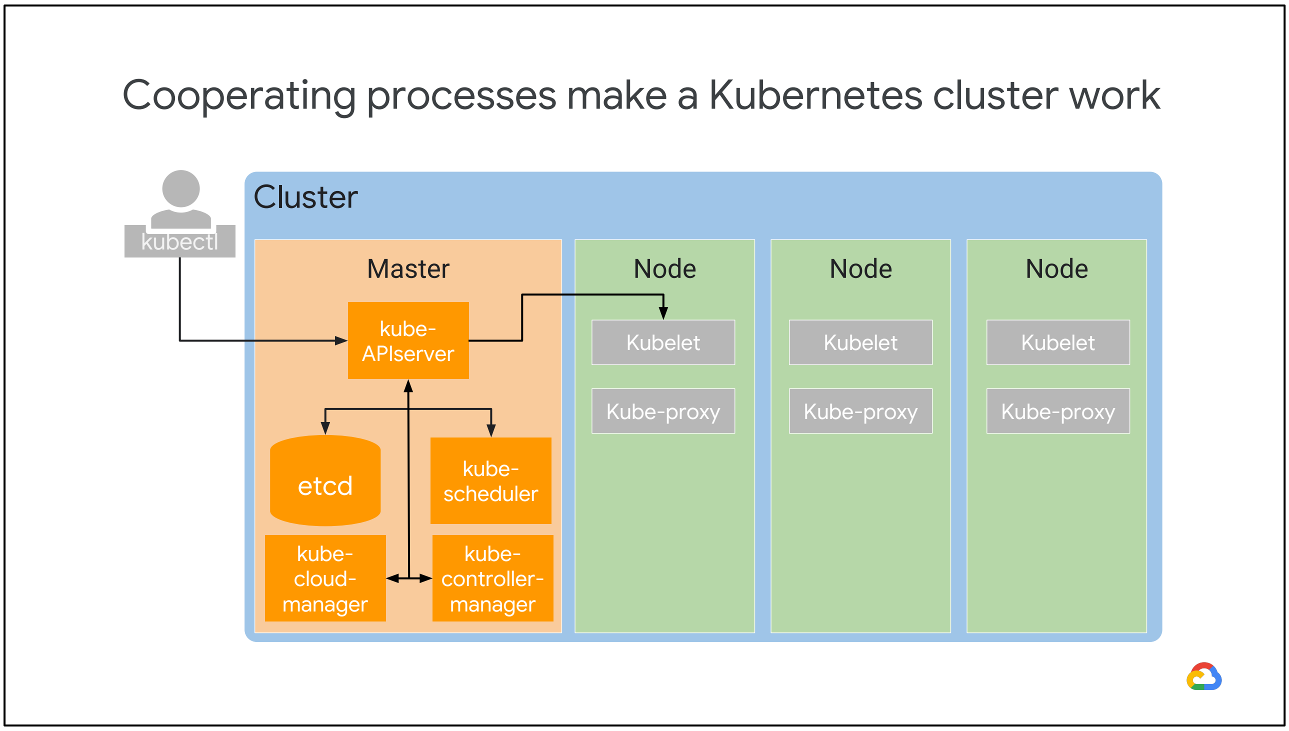
etcd is the cluster's database. Its job is to reliably store the state of the cluster. This includes all the cluster configuration data; and more dynamic information such as what nodes are part of the cluster, what Pods should be running, and where they should be running. You never interact directly 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 selecting 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 that 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 together into a controller object called a Deployment that not only keeps them running but also 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 GCP features like load balancers and storage volumes when you needed them.

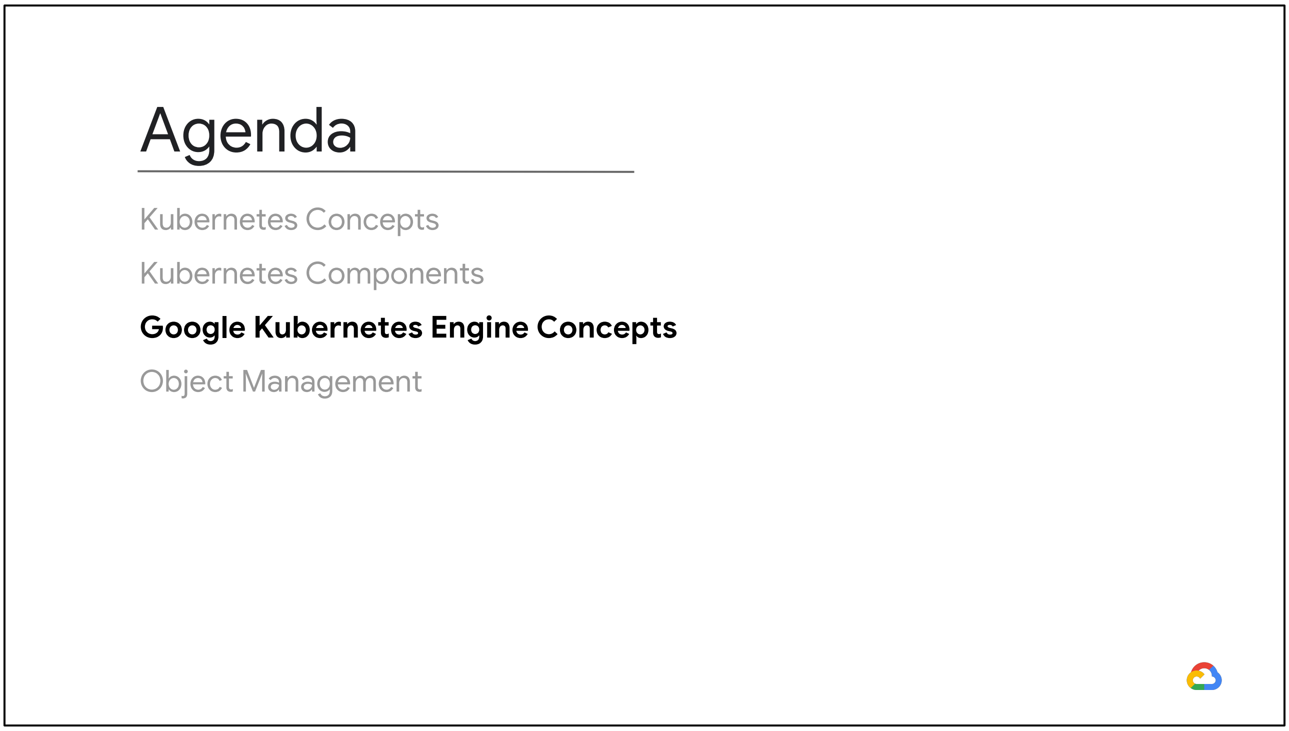


Cooperating processes make a Kubernetes cluster work

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 in this specialization, we will learn how GKE handles pod networking.



Agenda

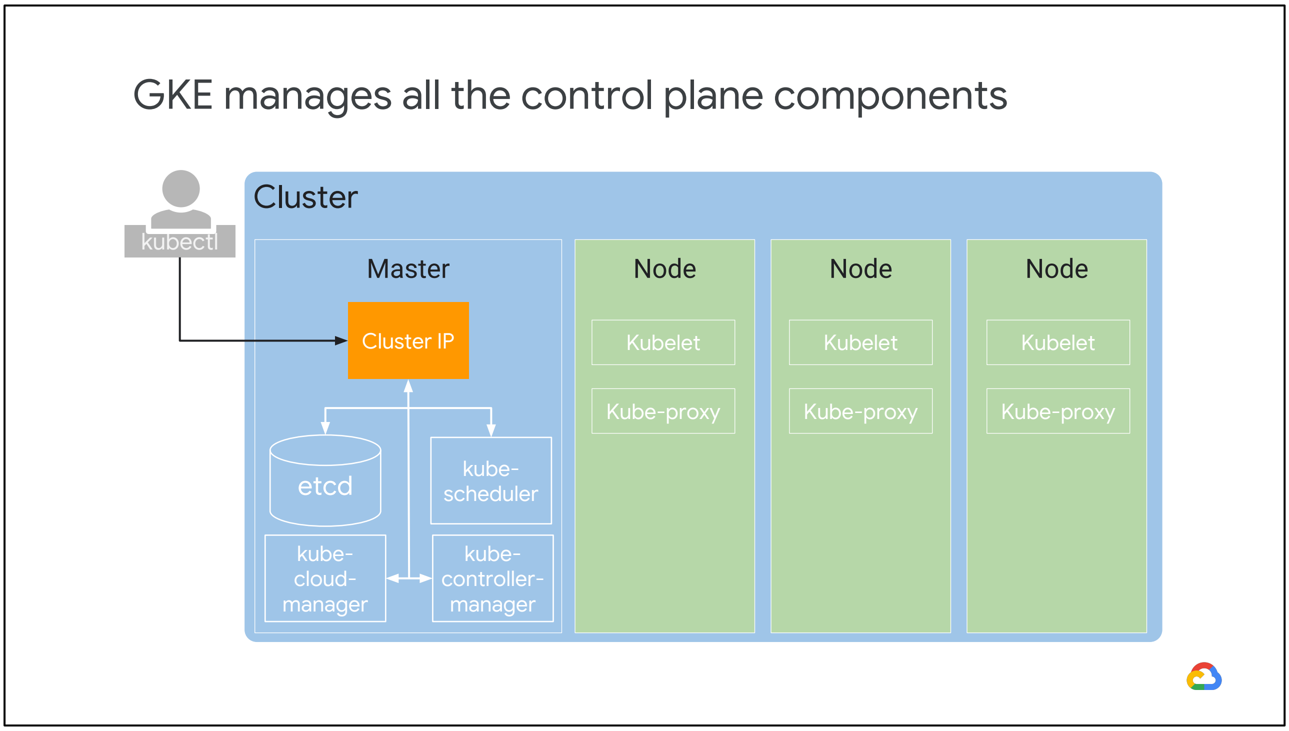
Kubernetes Concepts

Kubernetes Components

Google Kubernetes Engine Concepts

Object Management

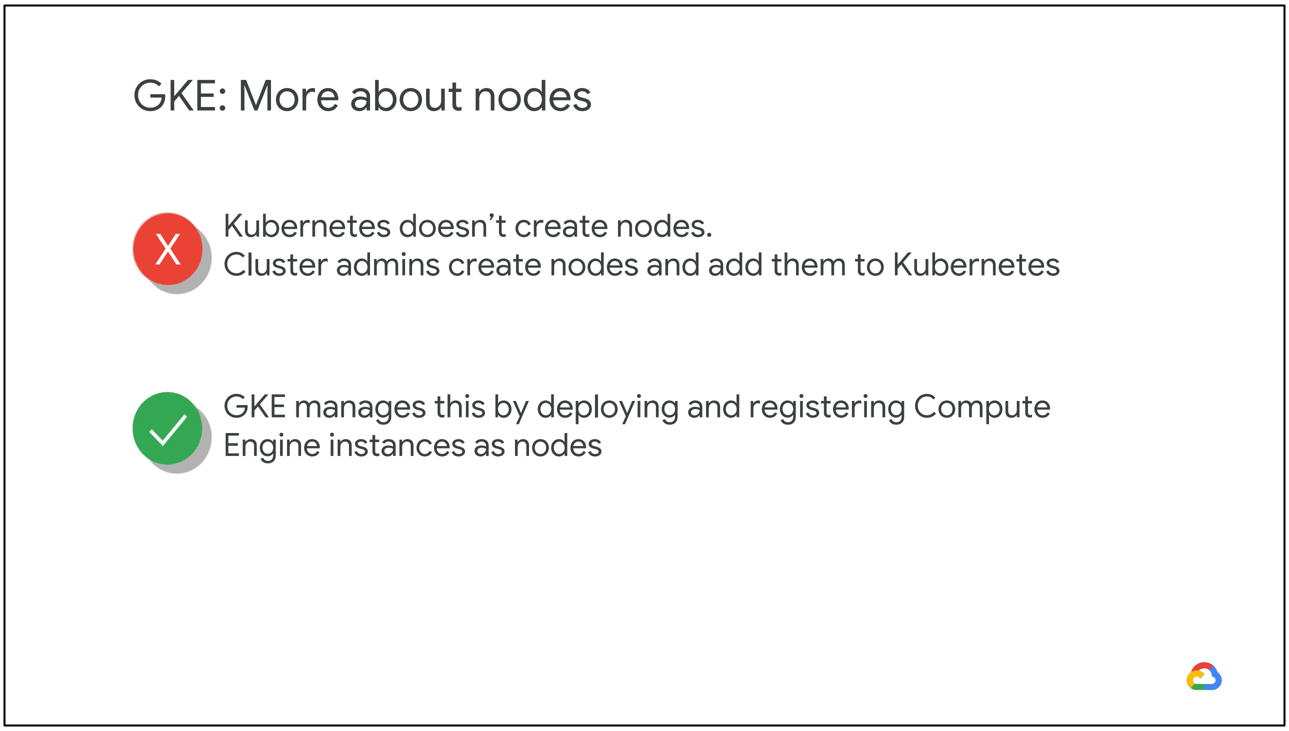
Next, we'll introduce concepts specific to Google Kubernetes Engine. That diagram of the Kubernetes control plane had a lot of components, didn't it? Setting up a Kubernetes cluster by hand is tons of work.



GKE manages all the control plane components

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 master infrastructure behind it. It also abstracts away having a separate master. The responsibilities of the master are absorbed by GCP, and you are not separately billed for your master.



GKE: More about nodes

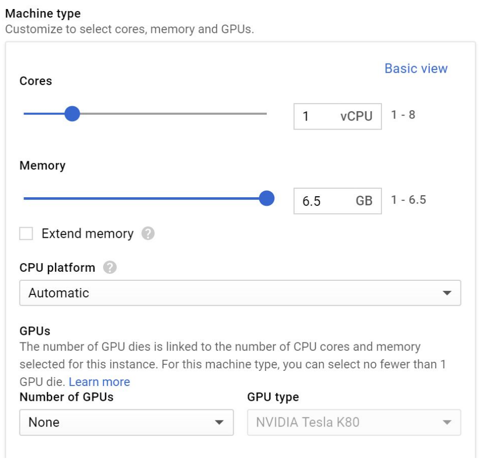
✗ Kubernetes doesn't create nodes.
Cluster admins create nodes and add them to Kubernetes

✓ GKE manages this by deploying and registering Compute Engine instances as nodes

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 GCP Console. You are charged per second of allocated time for your nodes (not counting the master).

GKE: More about nodes



Machine type
Customize to select cores, memory and GPUs.

Basic view

Cores: 1 vCPU (1 - 8)

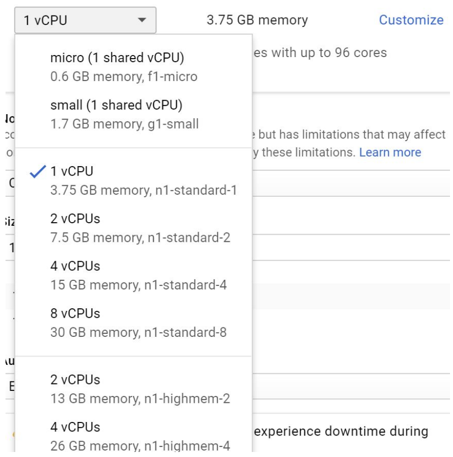
Memory: 6.5 GB (1 - 6.5)

☐ Extend memory

CPU platform: Automatic

GPUs
The number of GPU dies is linked to the number of CPU cores and memory selected for this instance. For this machine type, you can select no fewer than 1 GPU die. Learn more

Number of GPUs	GPU type
None	NVIDIA Tesla K80



1 vCPU — 3.75 GB memory — Customize

es with up to 96 cores

e but has limitations that may affect
y these limitations. Learn more

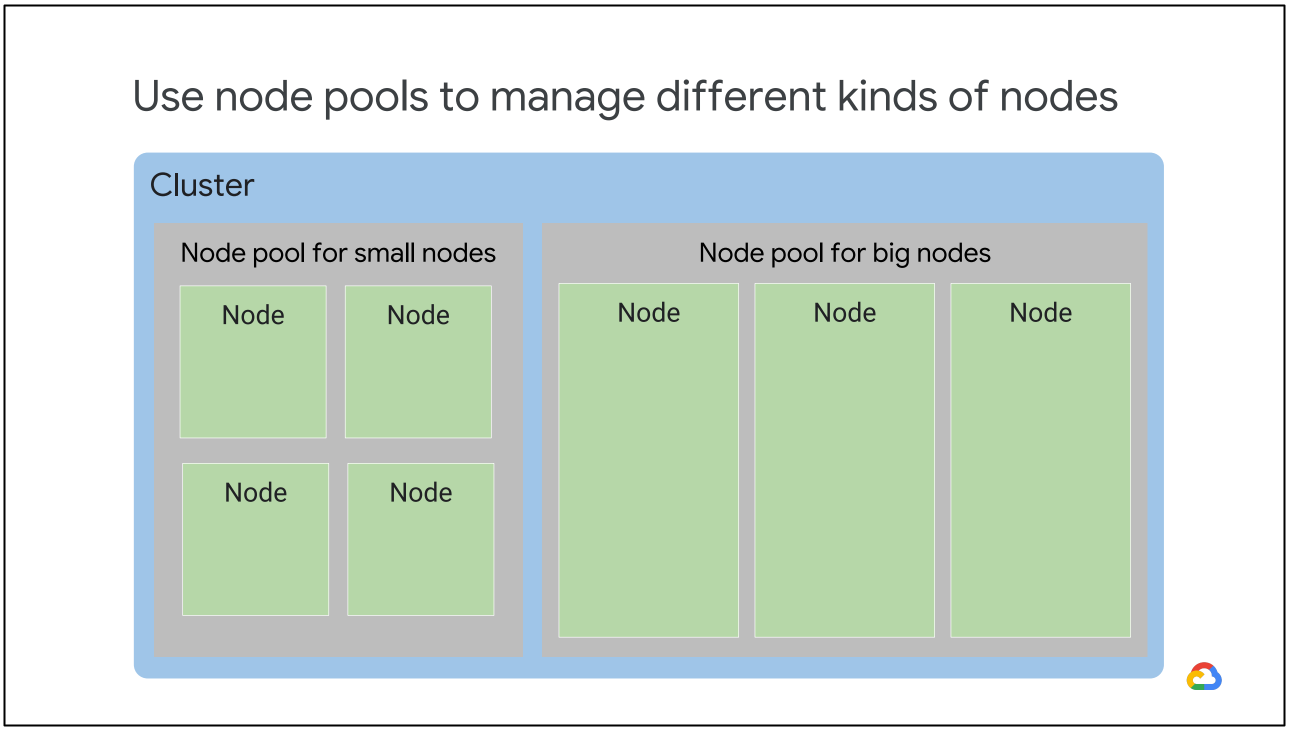
experience downtime during

micro (1 shared vCPU)	0.6 GB memory, f1-micro
small (1 shared vCPU)	1.7 GB memory, g1-small
✓ 1 vCPU	3.75 GB memory, n1-standard-1
2 vCPUs	7.5 GB memory, n1-standard-2
4 vCPUs	15 GB memory, n1-standard-4
8 vCPUs	30 GB memory, n1-standard-8
2 vCPUs	13 GB memory, n1-highmem-2
4 vCPUs	26 GB memory, n1-highmem-4

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

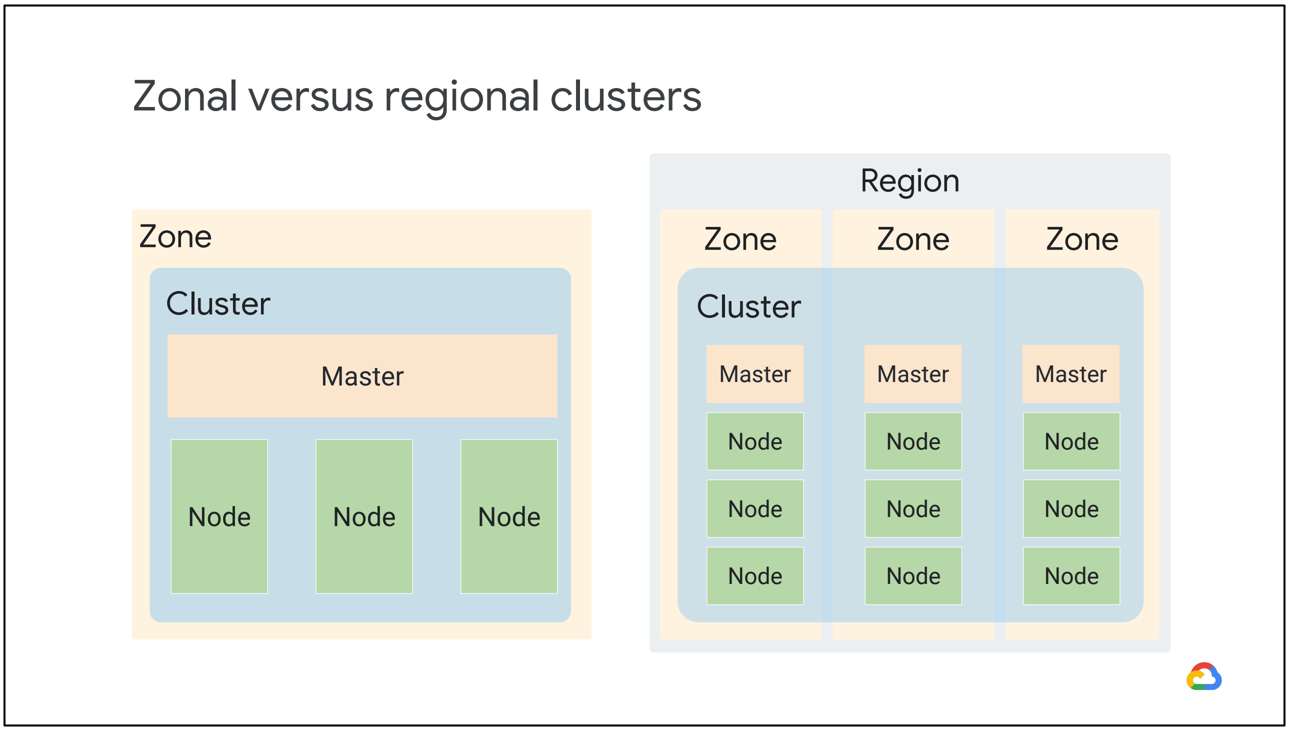
You can also select multiple node machine types by creating multiple node pools. A node pool is a subset of nodes within a cluster that share a configuration, such as their amount of memory, or their CPU generation. Node pools also provide an easy way to ensure that workloads run on the right hardware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

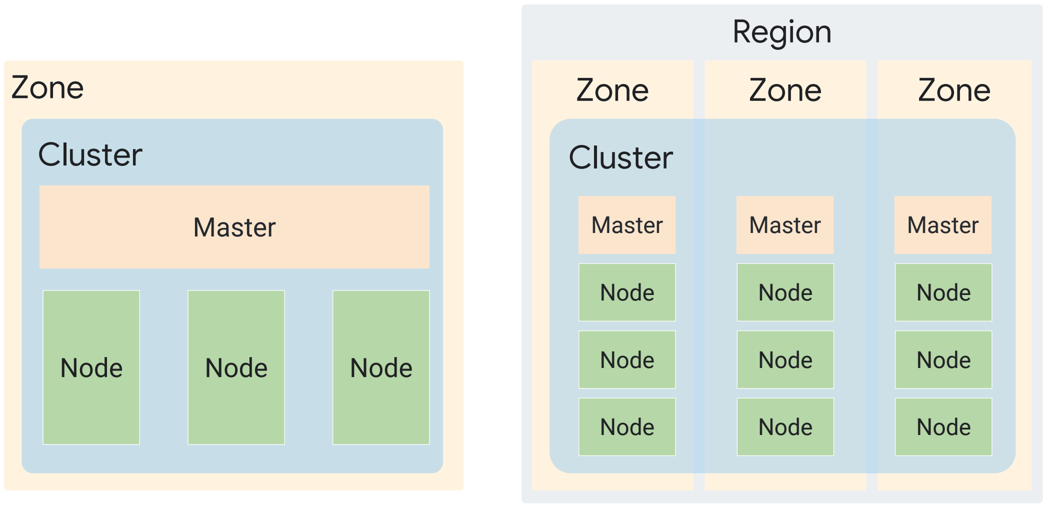
You can enable automatic node upgrades, automatic node repairs, and cluster autoscaling at this node pool level.

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



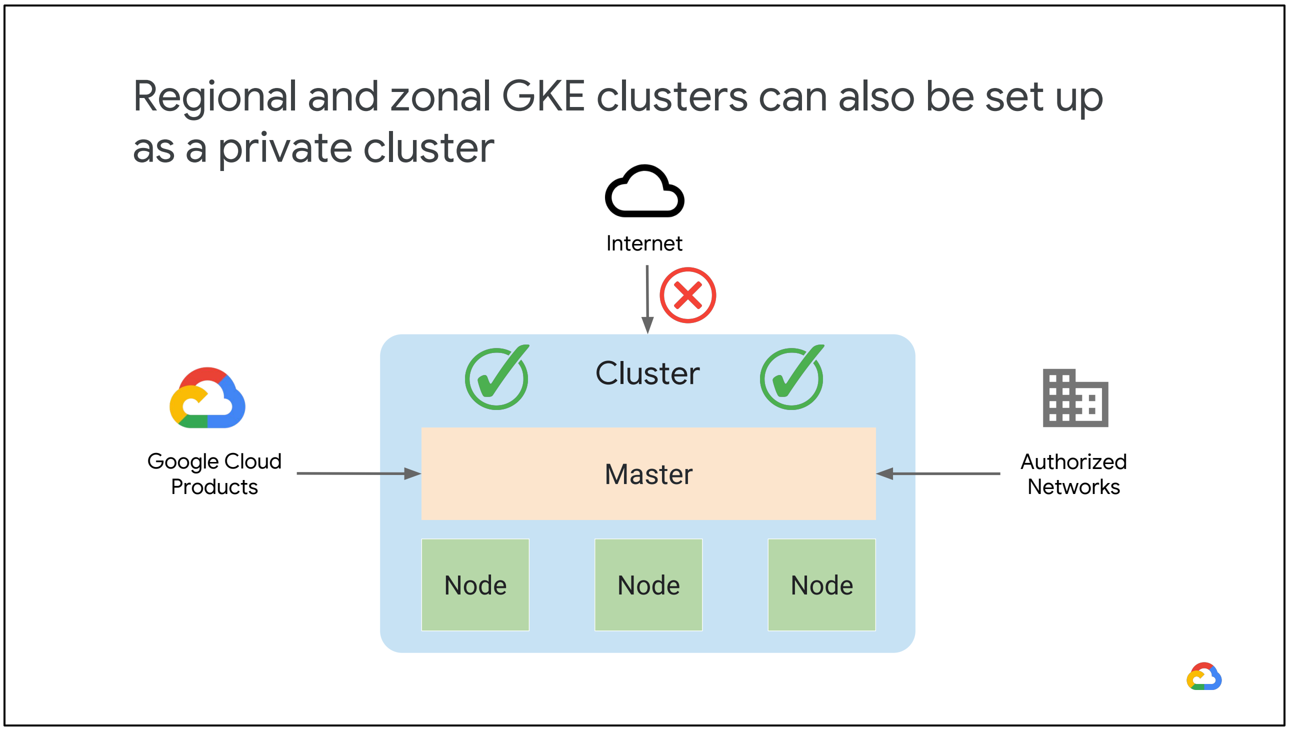
Zonal versus regional clusters



By default, a cluster launches in a single GCP 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 masters 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 master 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 master 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.

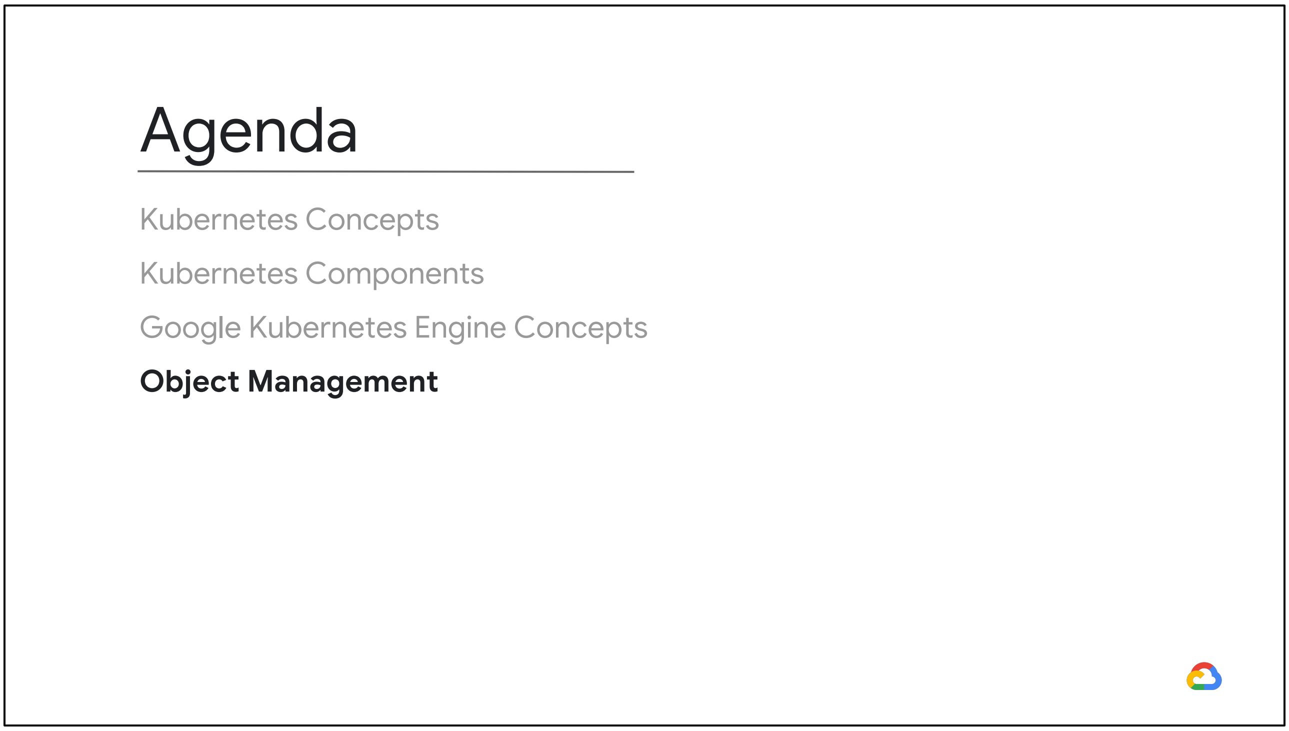


Regional and zonal GKE clusters can also be set up as a private cluster

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

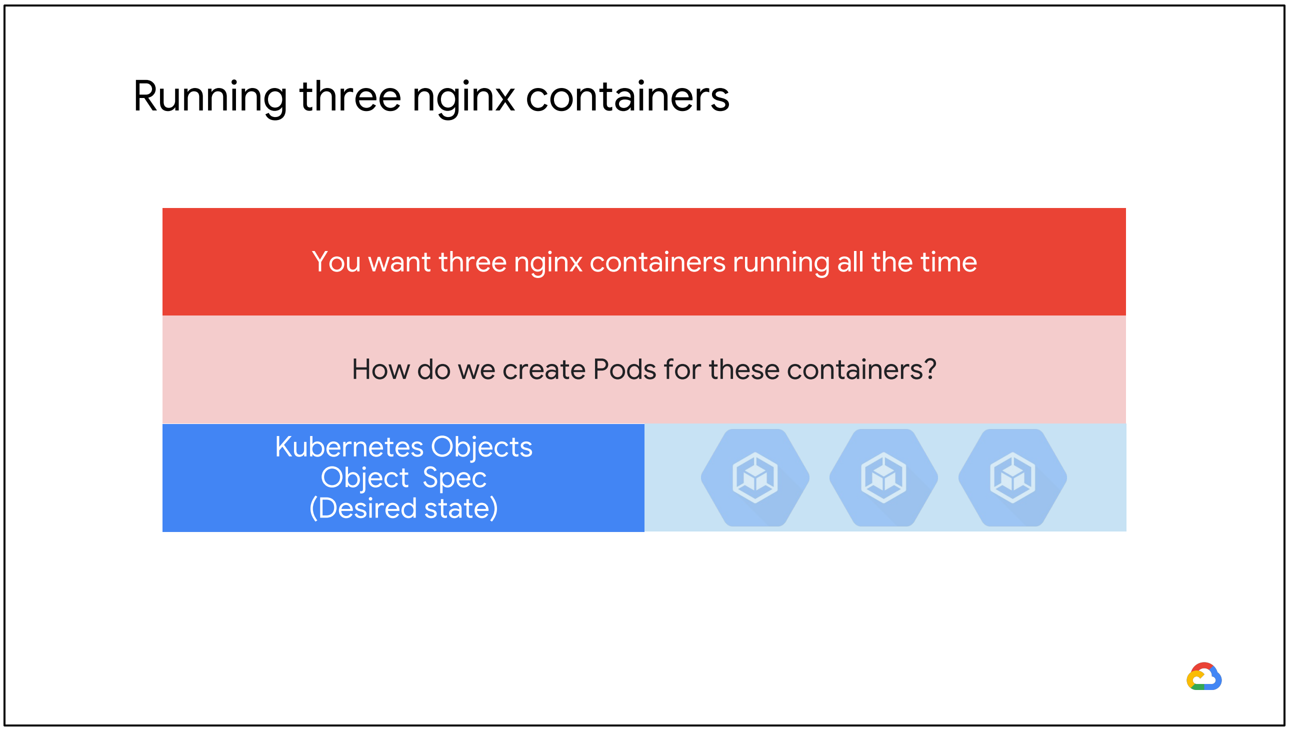
Cluster masters can be accessed by Google Cloud products, such as Stackdriver, through an internal IP address.

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 master. In addition, nodes can have limited outbound access through Private Google Access, which allows them them to communicate with other GCP services. For example, nodes can pull container images from Google Container Registry without needing external IP addresses. The topic of private clusters is discussed in more detail in another module in this specialization.



Agenda

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



Running three nginx containers

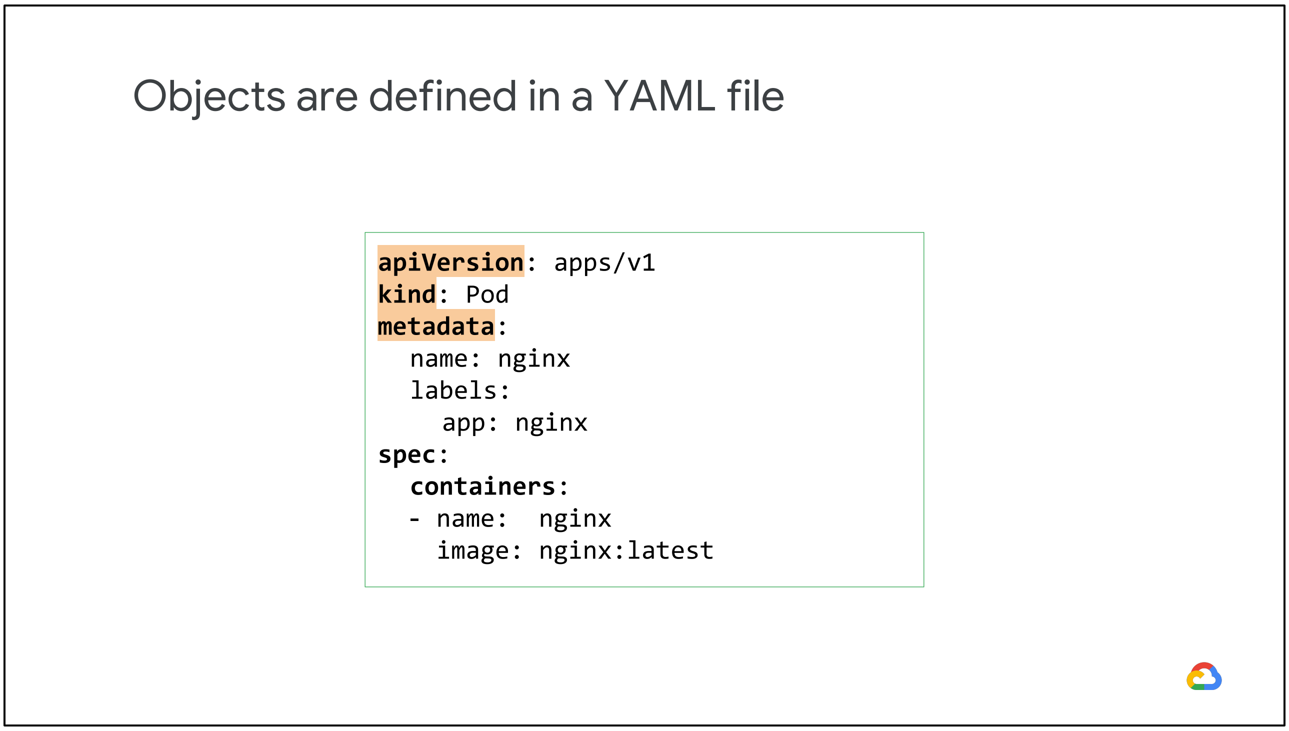
You want three nginx containers running all the time

How do we create Pods for these containers?

Kubernetes Objects
Object Spec
(Desired state)

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.



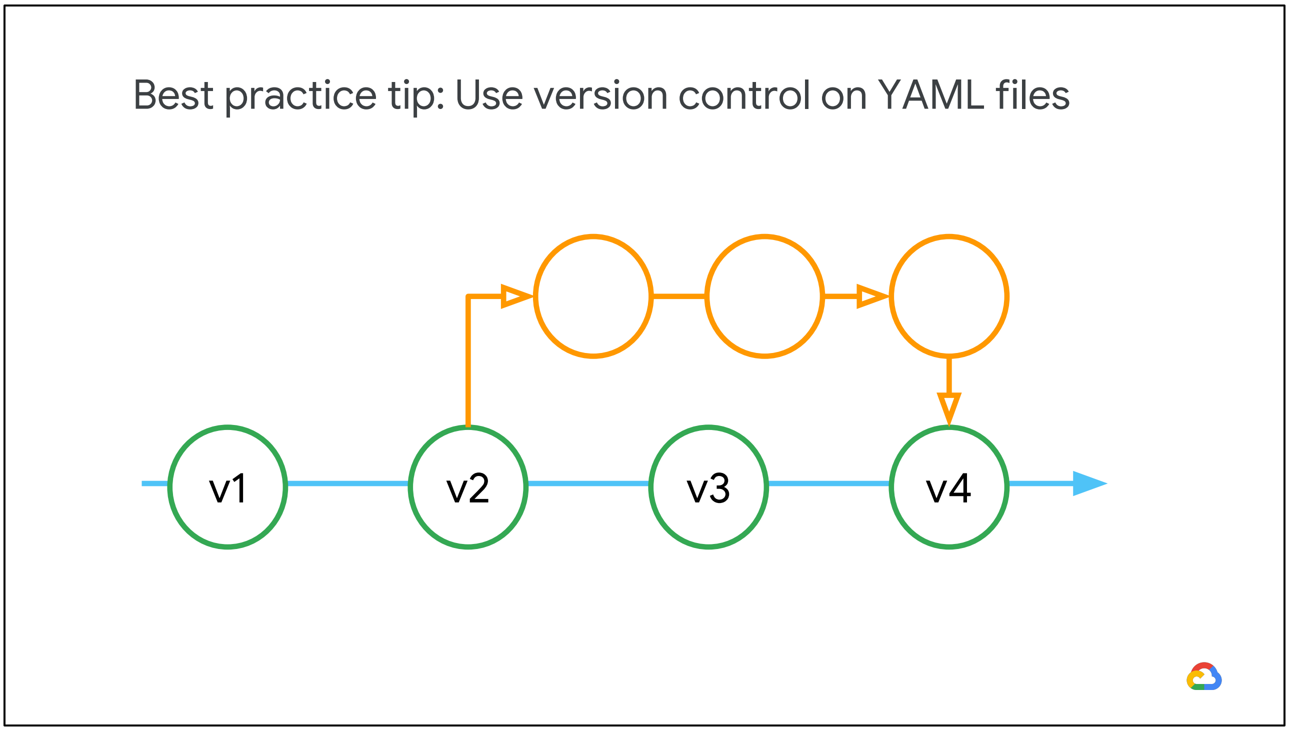
Objects are defined in a YAML file

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

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, and we will use it throughout this specialization. This YAML file defines a desired state for a pod: its name and a specific container image for it to run.

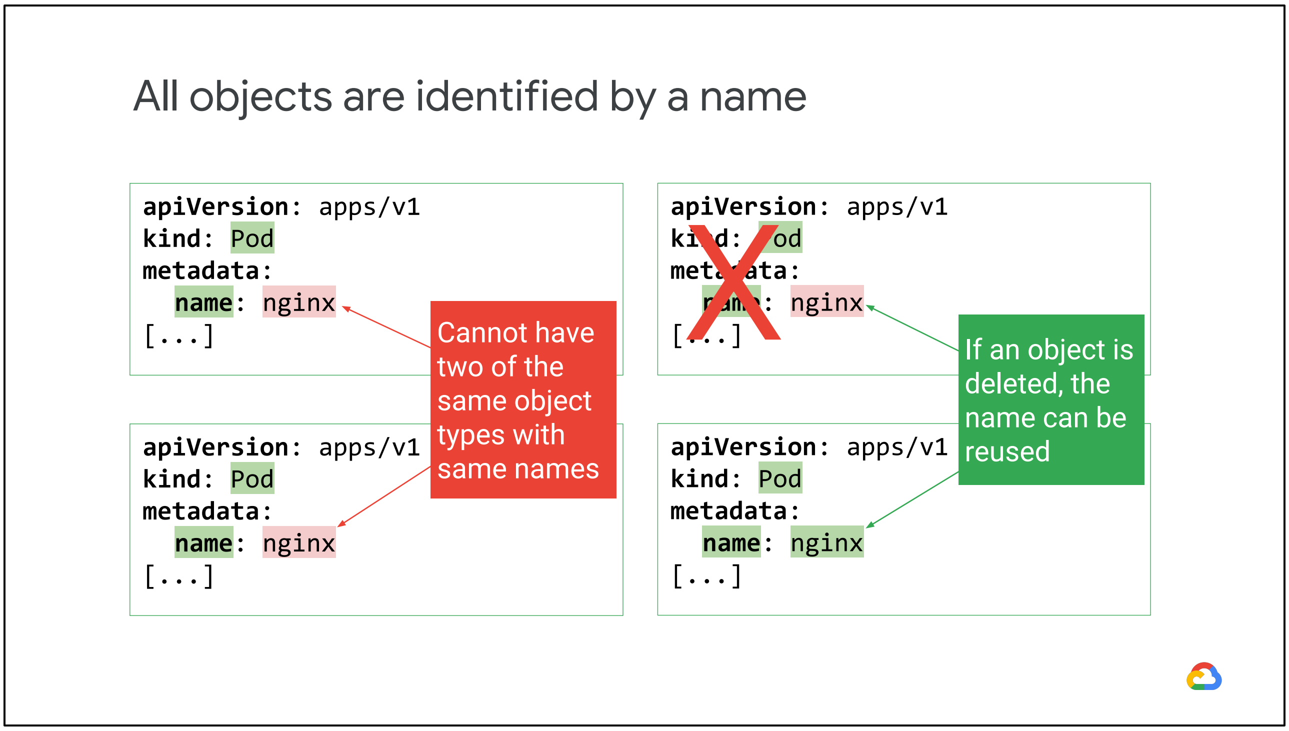
Your manifest files have certain required fields. ApiVersion describes which Kubernetes API version is used to create the object. The Kubernetes protocol is versioned so as to help maintain backwards compatibility.

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

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 GCP 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 GCP resources.



All objects are identified by a name

```
apiVersion: apps/v1
kind: Pod
metadata:
  name: nginx
[...]
```

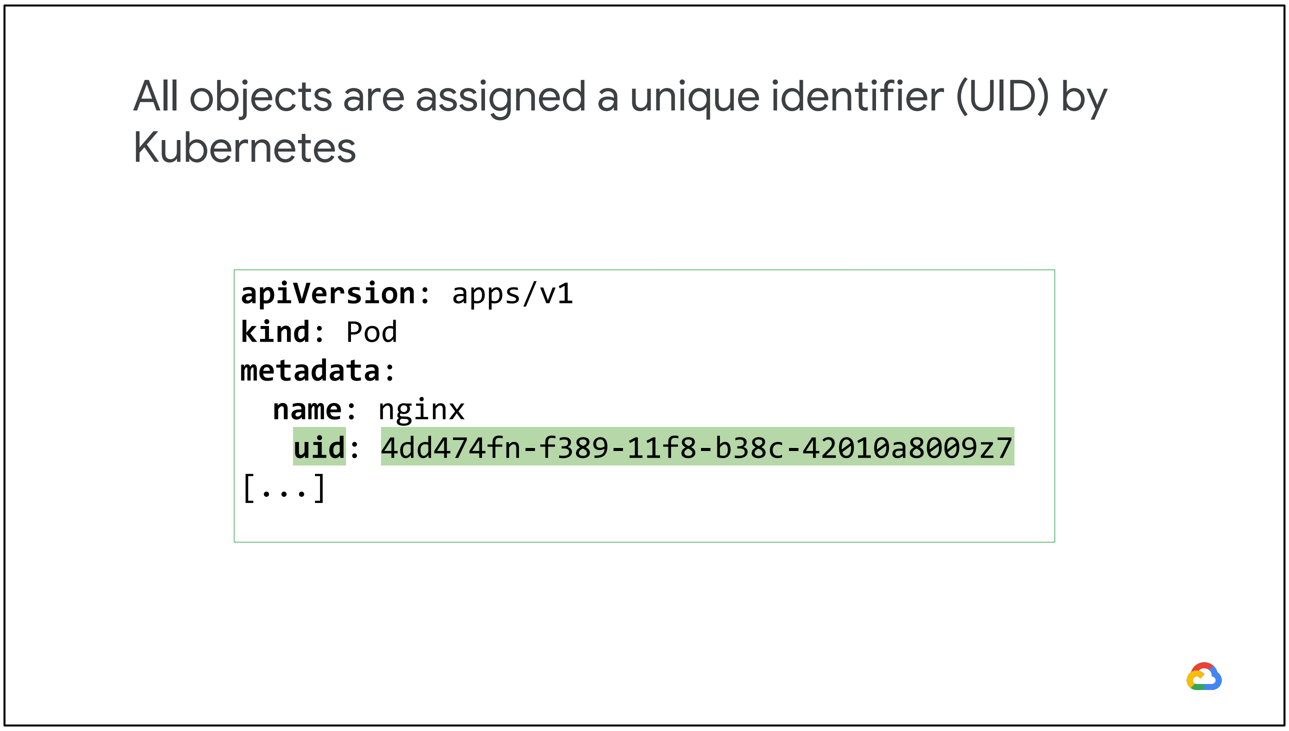
```
apiVersion: apps/v1
kind: Pod
metadata:
  name: nginx
[...]
```

```
apiVersion: apps/v1
kind: Pod
metadata:
  name: nginx
[...]
```

```
apiVersion: apps/v1
kind: Pod
metadata:
  name: nginx
[...]
```

Cannot have two of the same object types with same names
If an object is deleted, the name can be reused

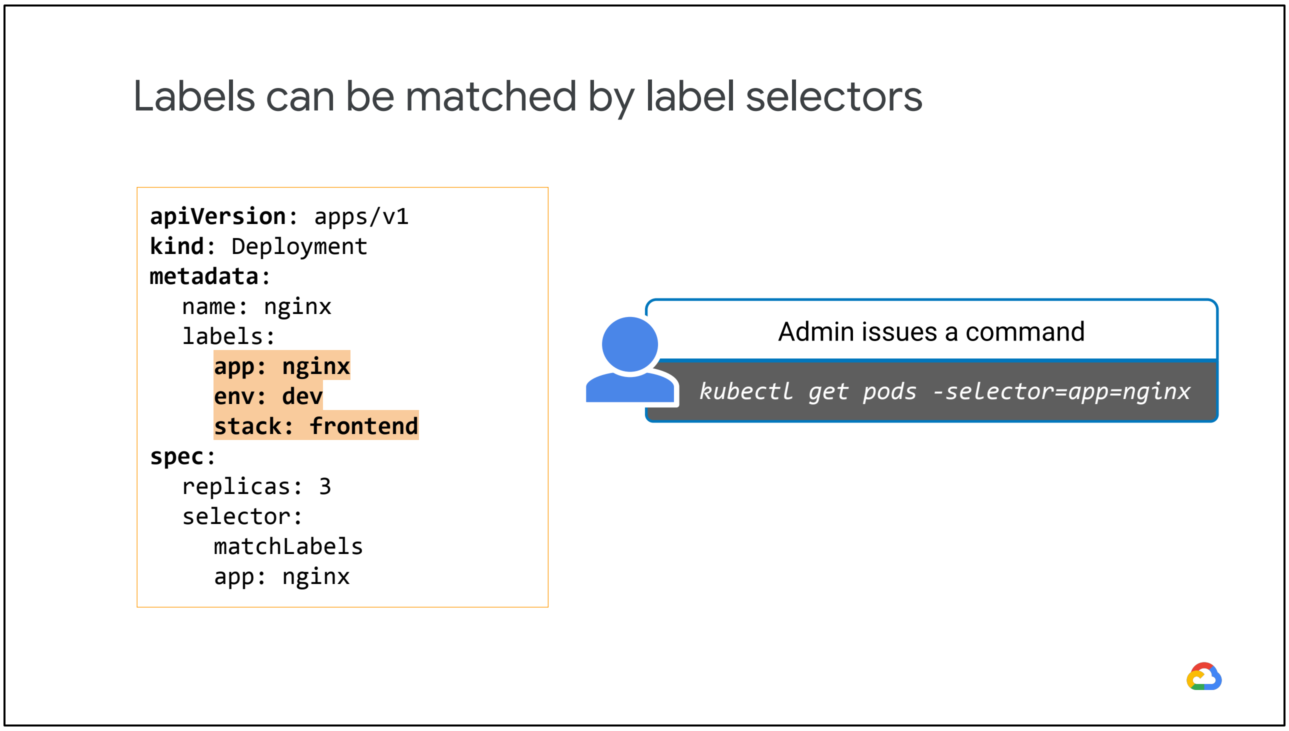
When you create a Kubernetes object, you name it with a string. Names must be unique. Only one object of a particular kind can have a particular name at the same time in the same Kubernetes namespace. However, if an object is deleted, its name can be reused. Alphanumeric characters, hyphens, and periods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

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: Deployment
metadata:
   name: nginx
   labels:
      app: nginx
      env: dev
      stack: frontend
spec:
   replicas: 3
   selector:
      matchLabels
      app: nginx
```

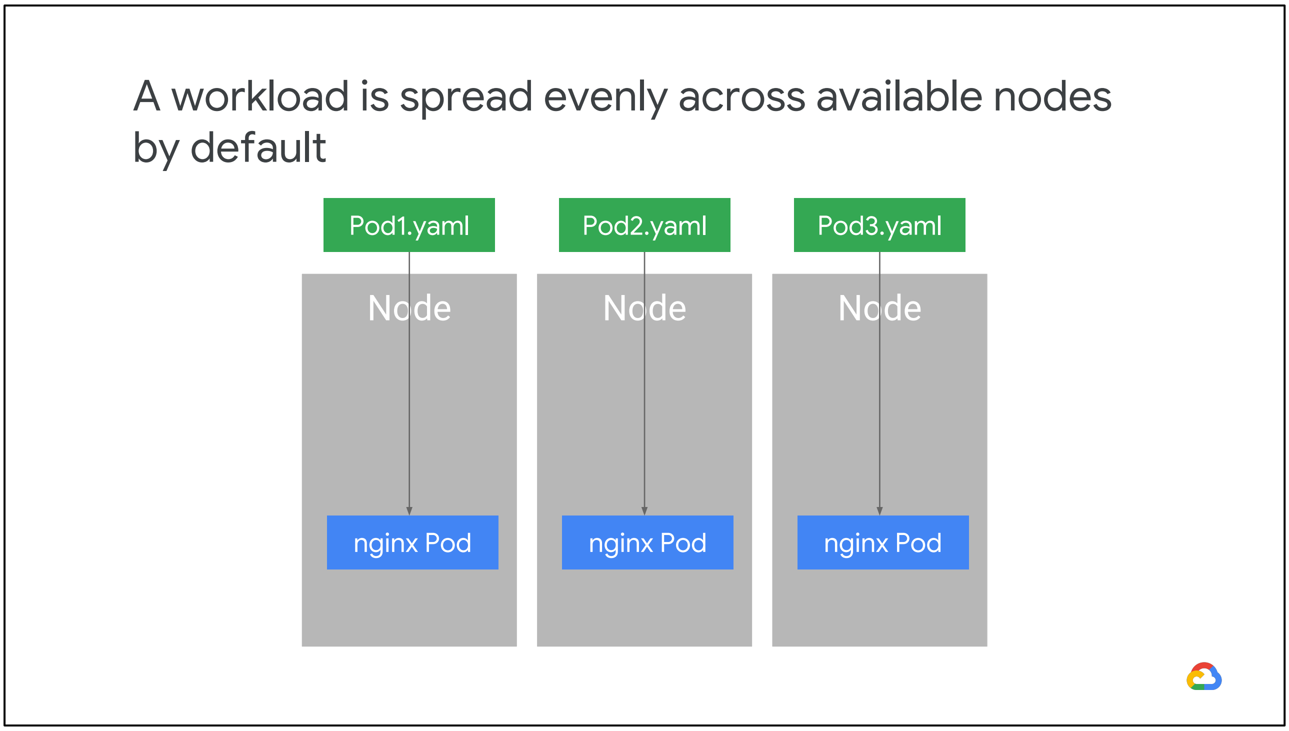
Admin issues a command

```
kubectl get pods -selector=app=nginx
```

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.

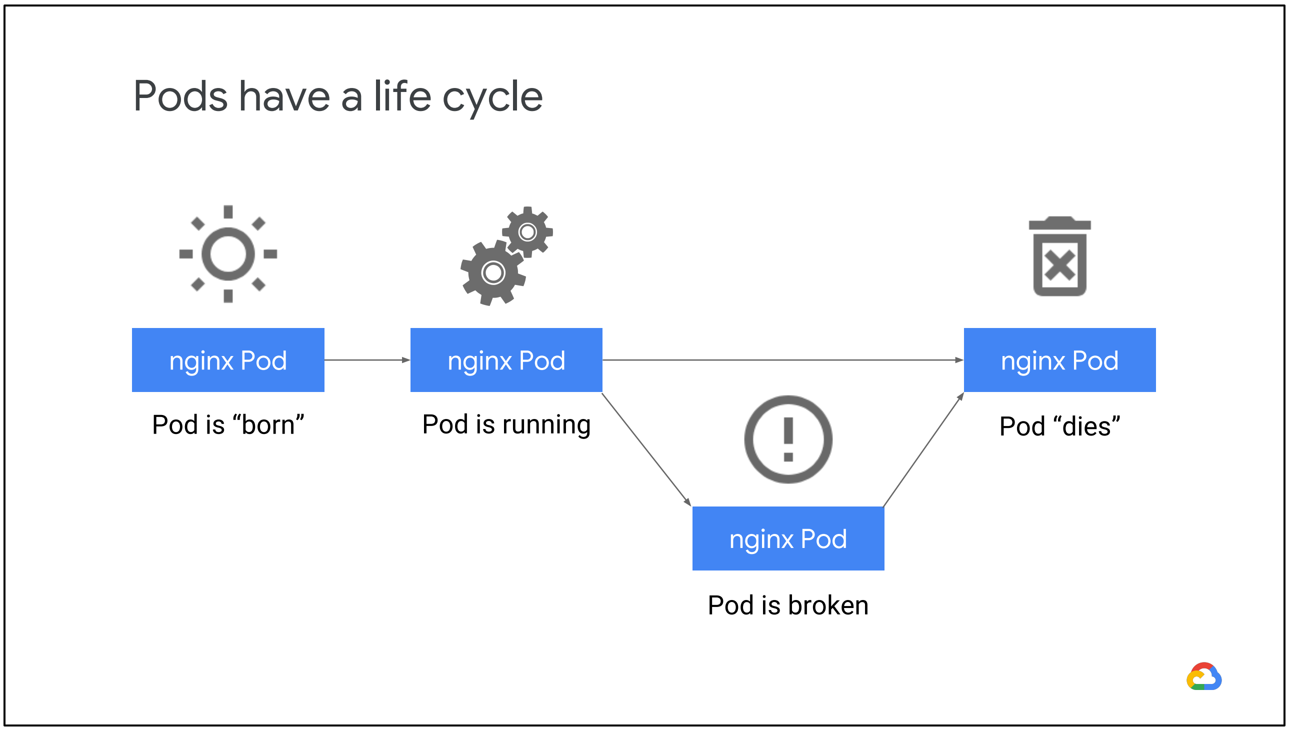
In this simple example, a Deployment object is labeled with three different key-values: its application, its environment, and which stack it forms a part of.

Various contexts offer ways to select Kubernetes resources by their labels. In this specialization, you will spend plenty of time with the kubectl command; 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

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.

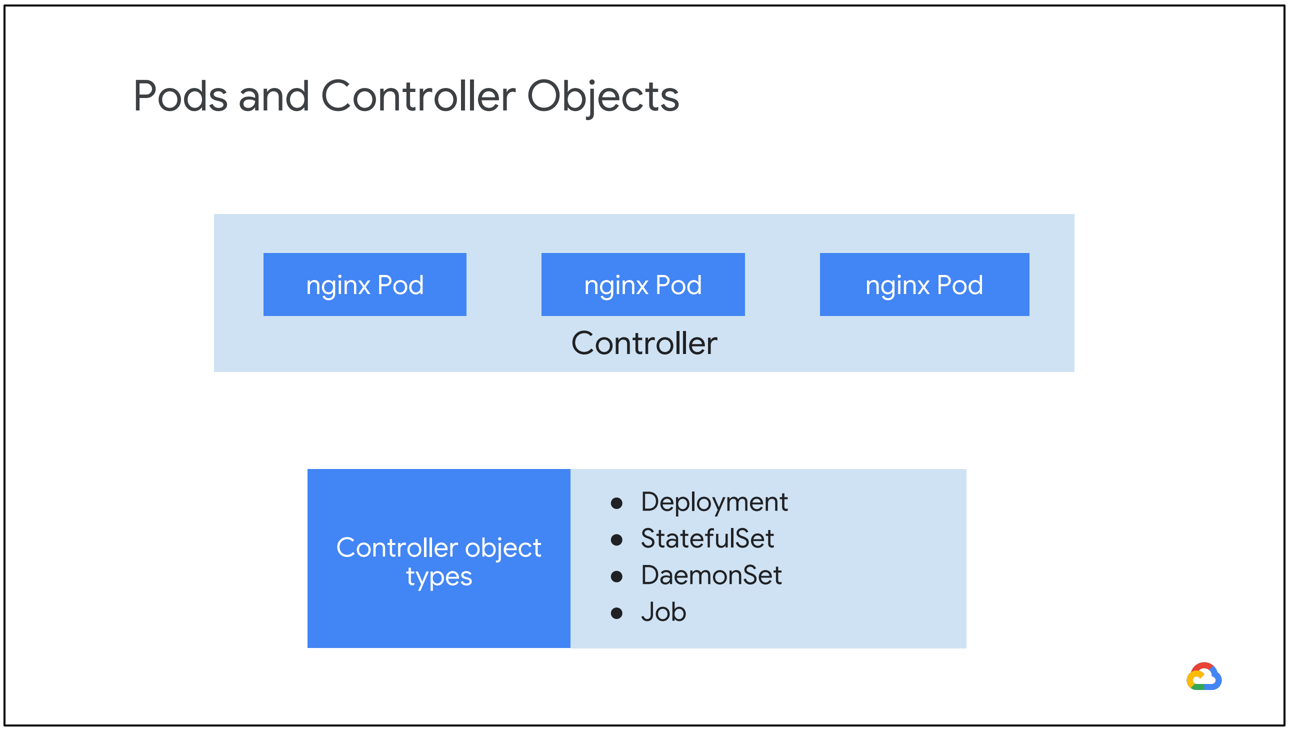


Pods have a life cycle

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?
Image source: Gears
https://svgsilh.com/search/clock-1.html

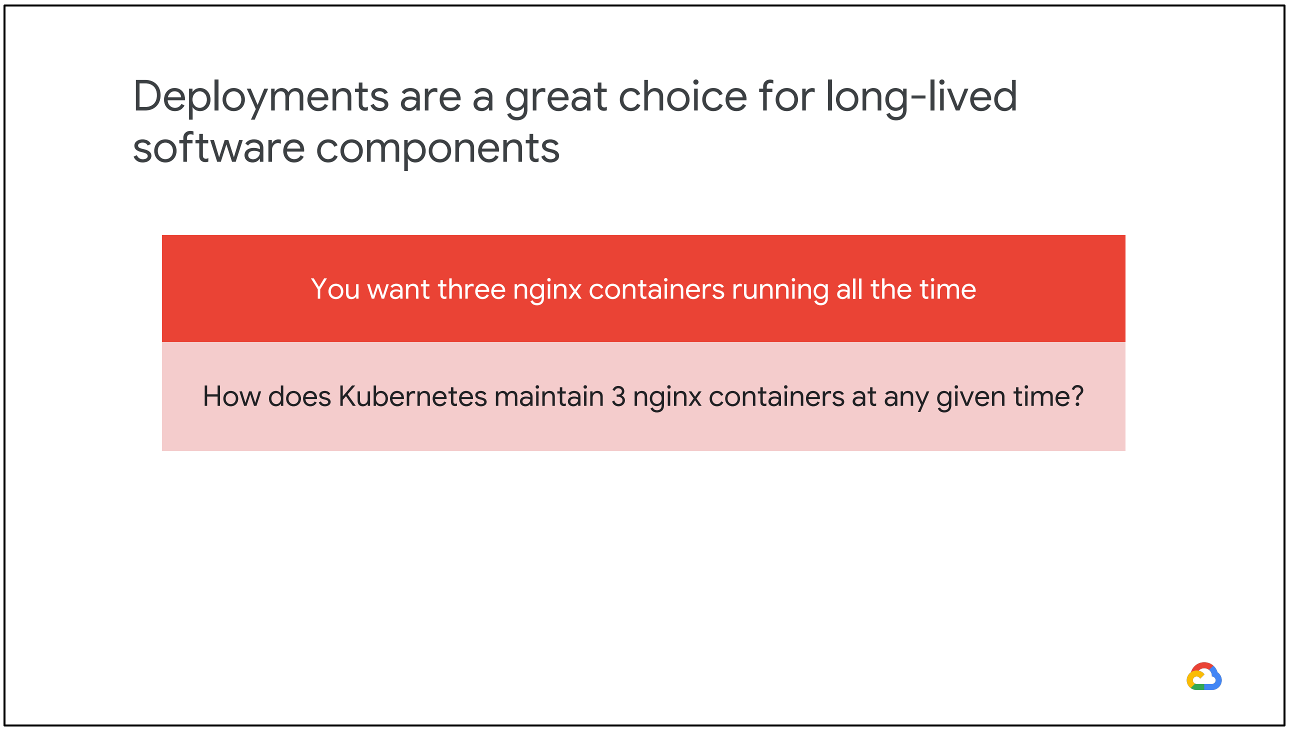


Pods and Controller Objects

nginx Pod
nginx Pod
nginx Pod
Controller

Controller object types	
	● Deployment ● StatefulSet ● DaemonSet ● Job

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. We'll meet all of these in our specialization.

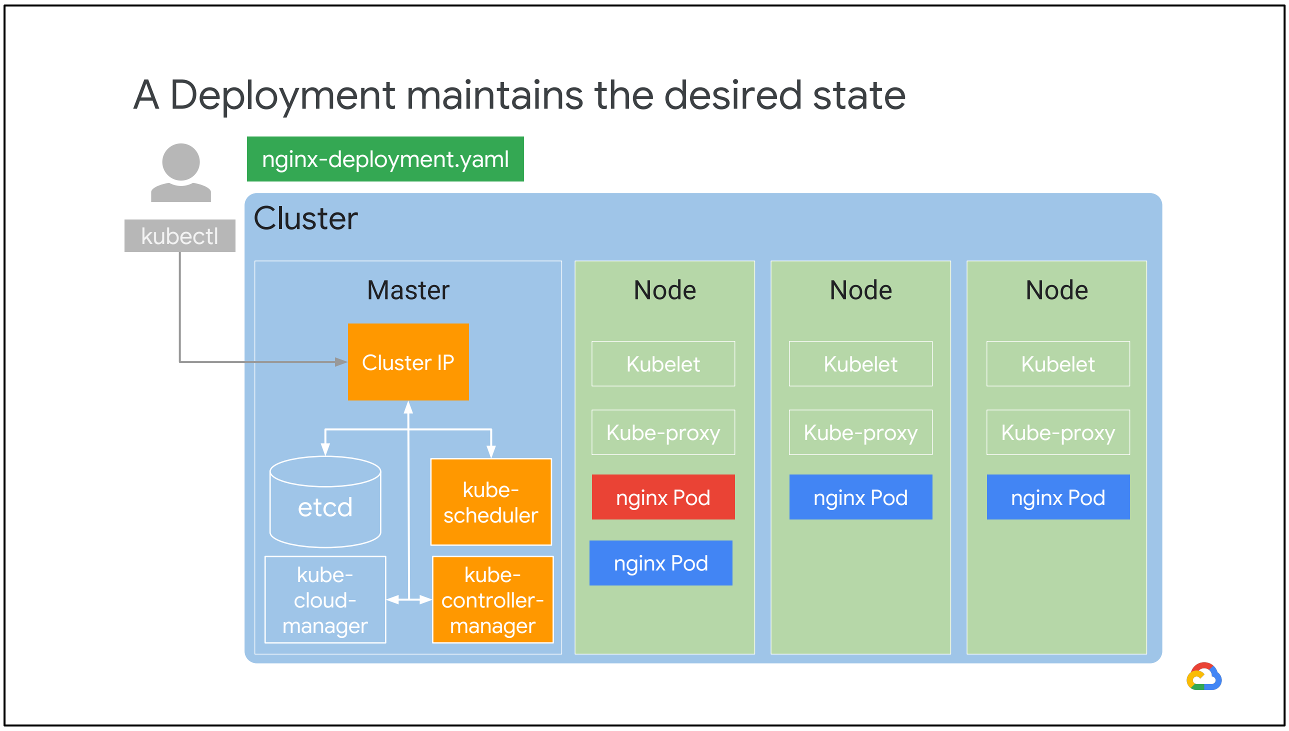


Deployments are a great choice for long-lived software components

You want three nginx containers running all the time

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

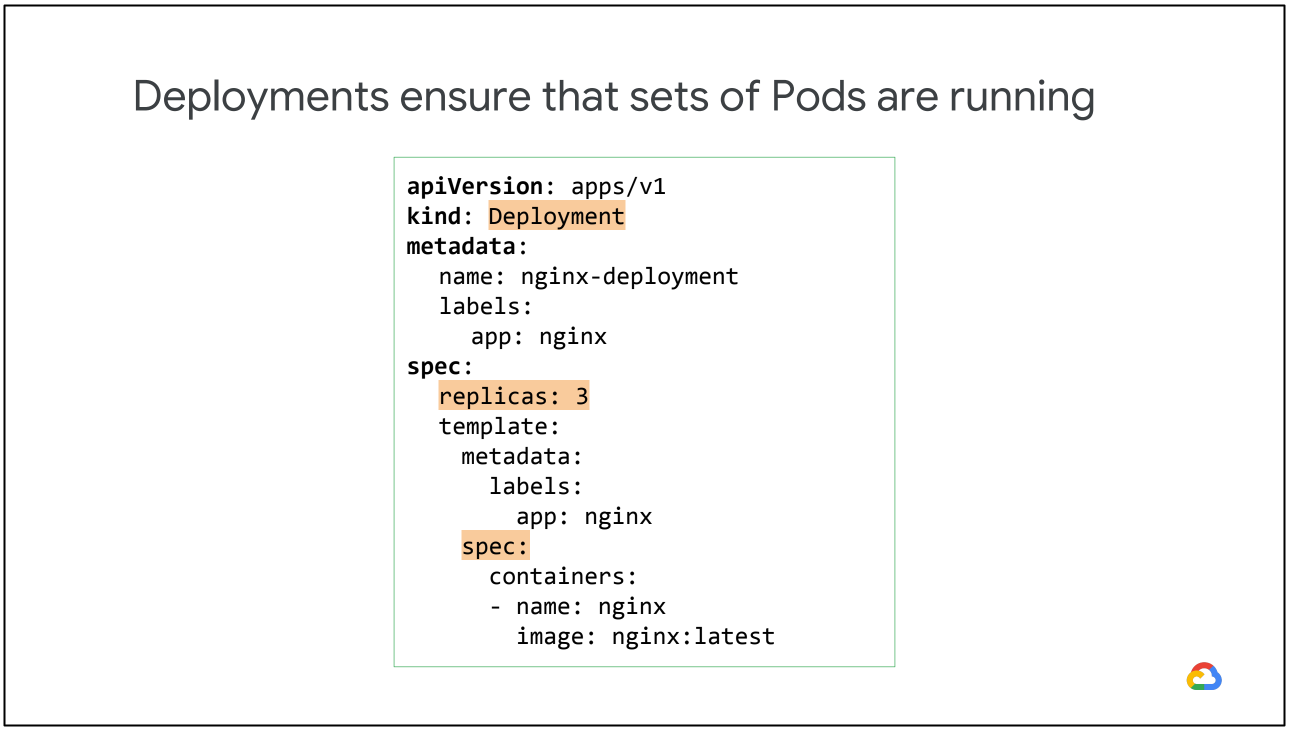
Deployments are a great choice for long-lived software components like Web servers, especially when we want to manage them as a group.



A Deployment maintains the desired state

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.



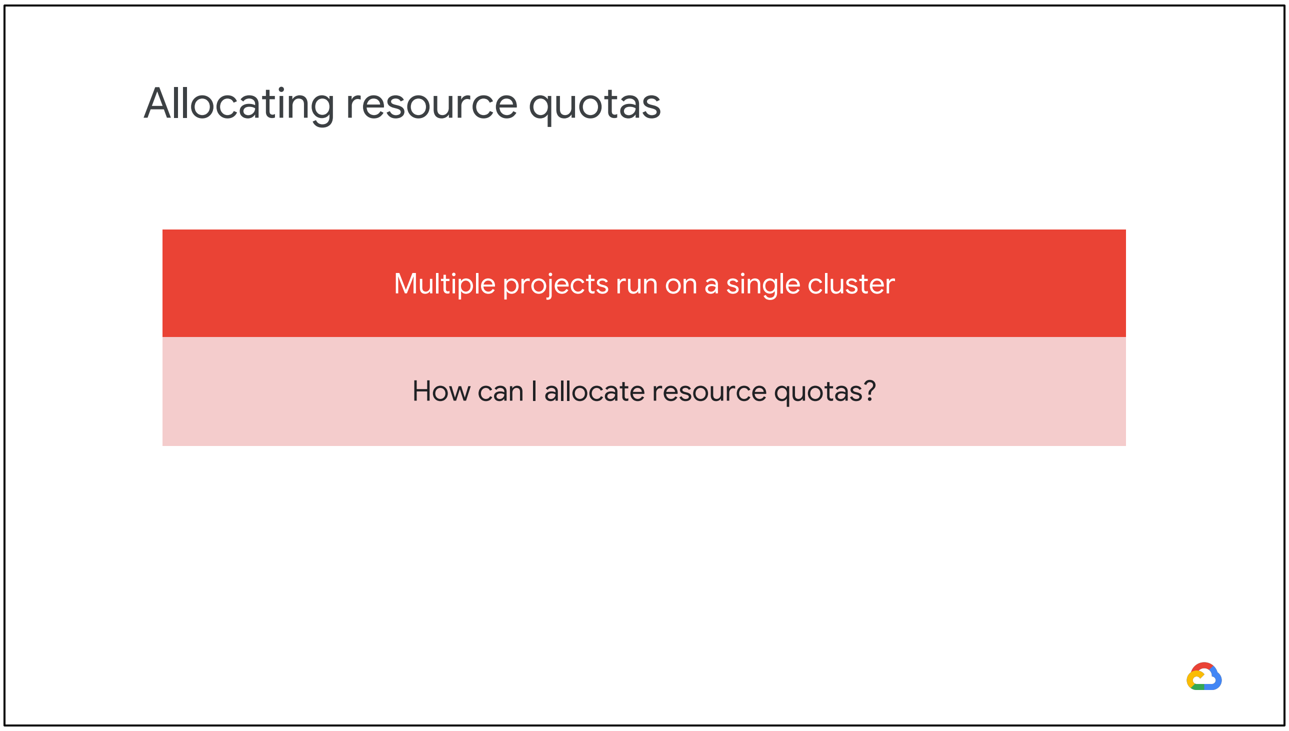
Deployments ensure that sets of Pods are running

```
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
```

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.

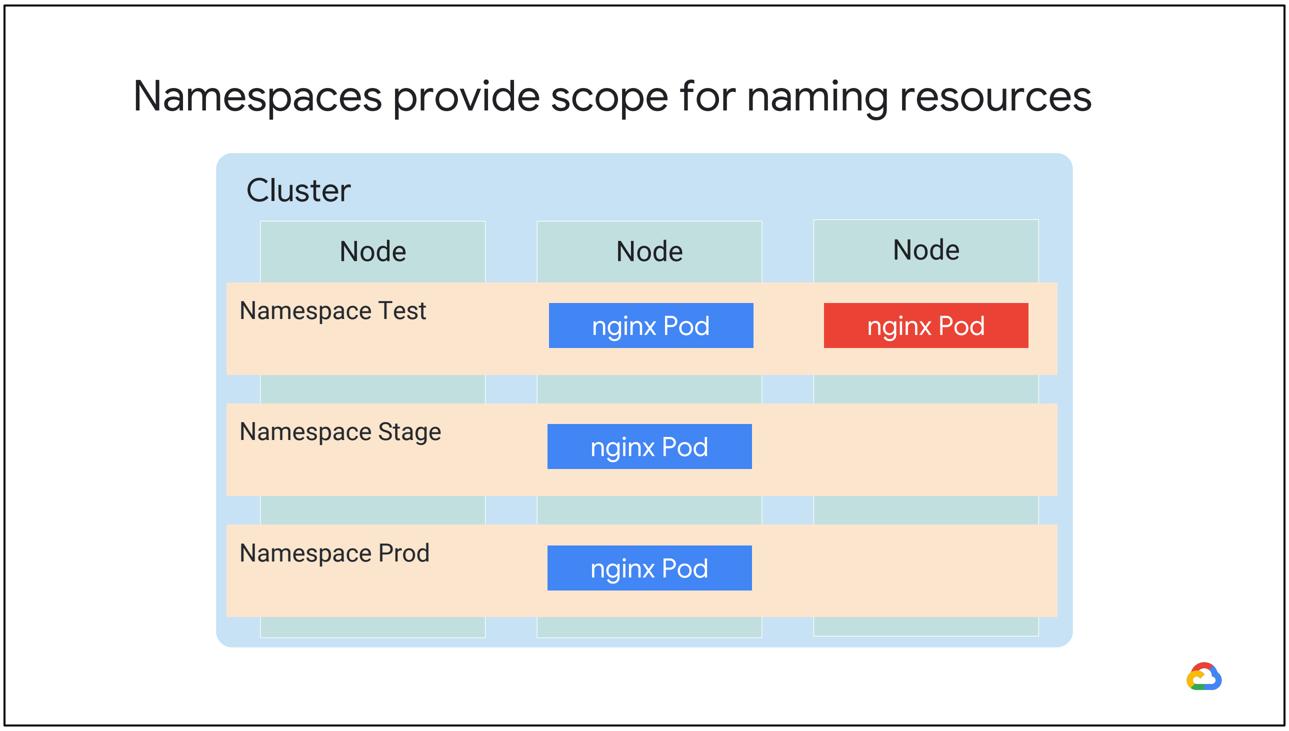


Allocating resource quotas

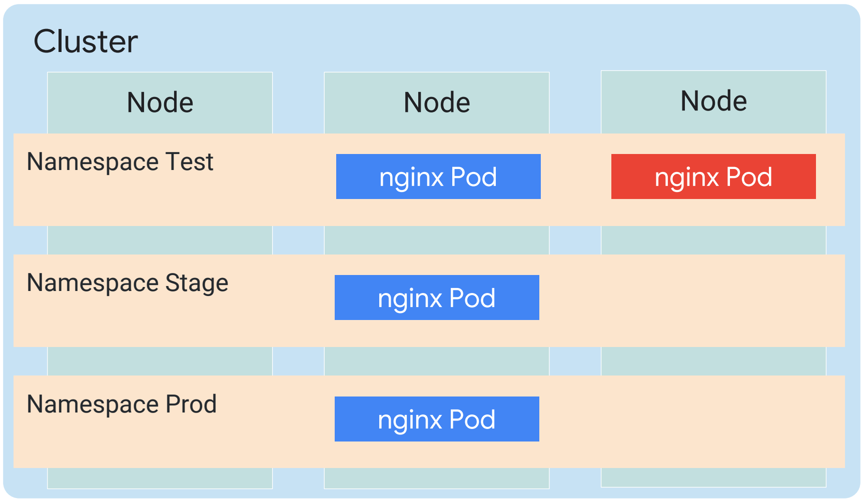
Multiple projects run on a single cluster

How can I allocate resource quotas?

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 GCP 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



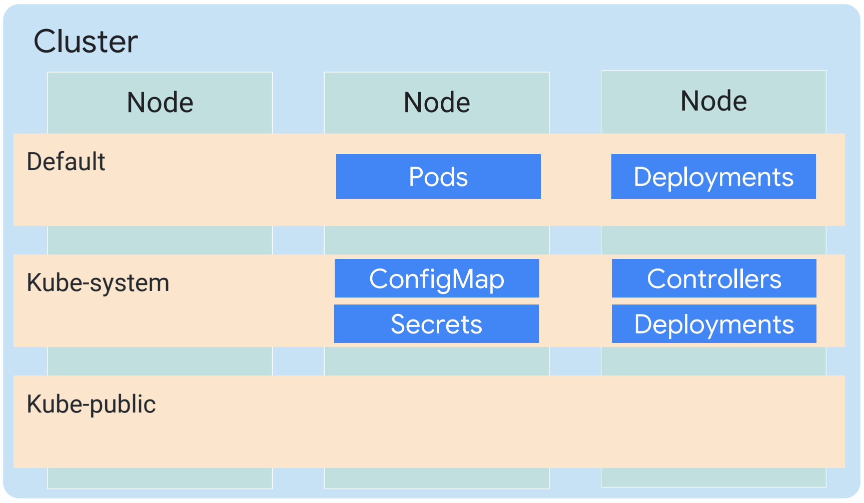
So how do you keep everybody's work on your cluster tidy and organized? Kubernetes allows you to abstract a single physical cluster into multiple virtual clusters known as *namespaces*. Namespaces provide scope for naming resources such as Pods, Deployments, and controllers.

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 GCP 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.

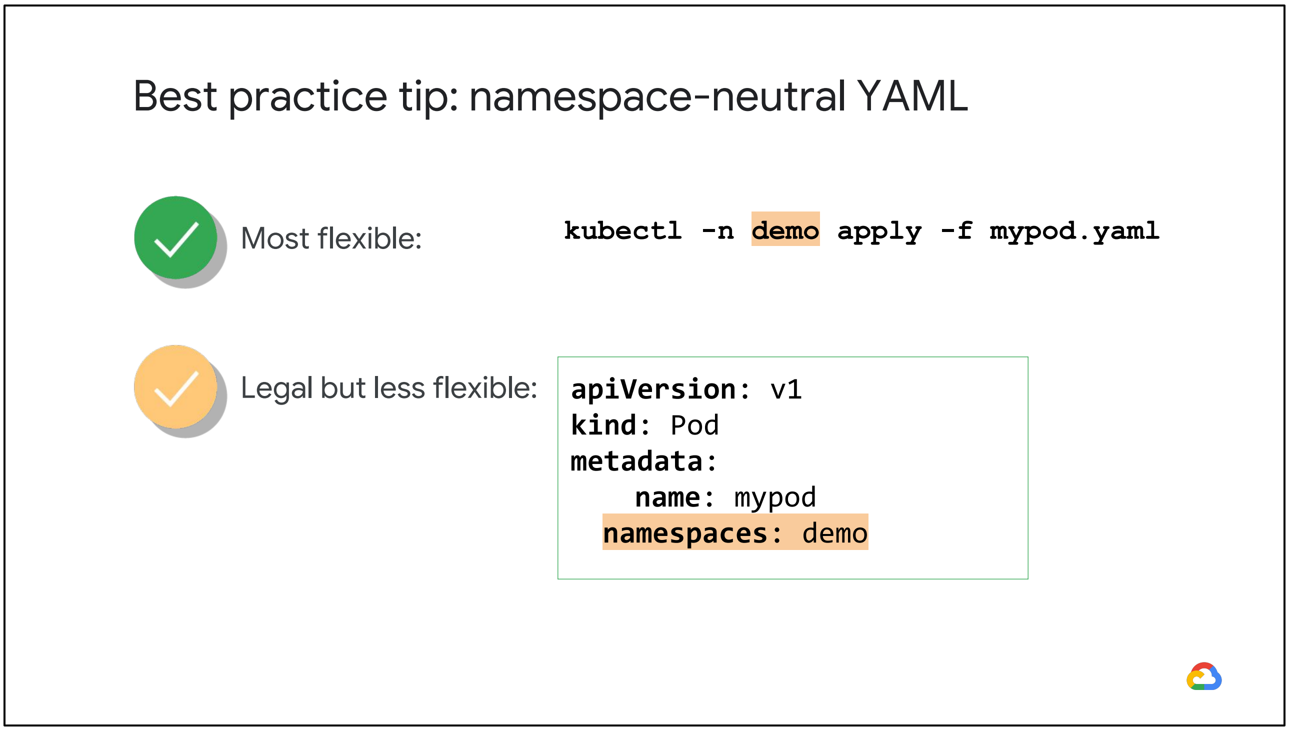
There are three initial namespaces in a cluster



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. We'll see more of the object kinds in this diagram elsewhere in this specialization. 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

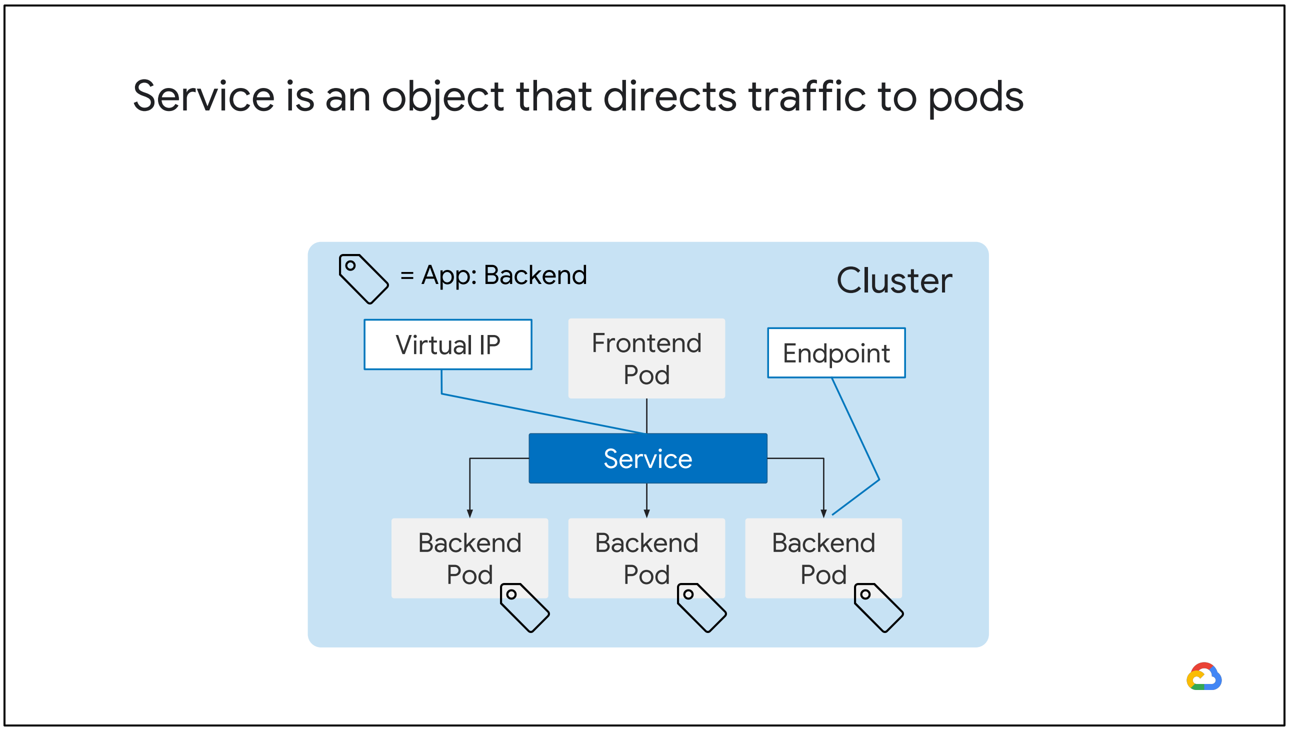
✓ Most flexible:

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

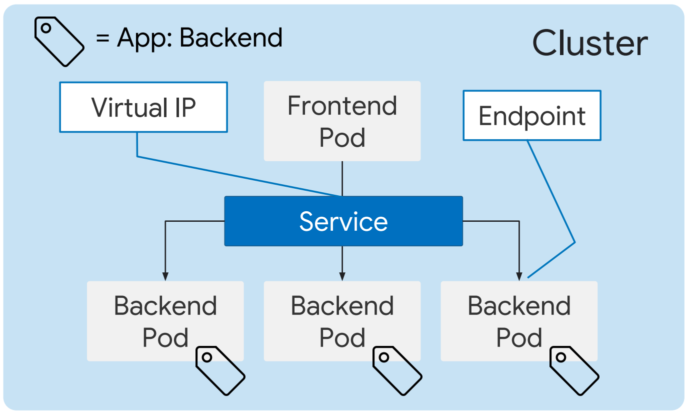
✓ Legal but less flexible:

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

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.



Service is an object that directs traffic to pods



Remember that Pods are created and destroyed dynamically. Although Pods can communicate using their assigned Pod IP addresses, these IP addresses are ephemeral; they are not guaranteed to remain constant when Pods are restarted or when scaling changes which nodes are used to run Pods.

Imagine you have two sets of Pods: frontend Pods and backend Pods. How will the frontend Pods discover and keep track of dynamically scaling backend Pods? This is where the concept of Kubernetes Services comes in.

A Kubernetes Service is a static IP address that represents a Service, or a function, in your infrastructure. It's a network abstraction for a set of Pods that deliver that Service, and it hides the ephemeral nature of the IP addresses of the individual Pods. In the example, a set of backend Pods are exposed to the frontend Pod using a Kubernetes Service. Basically, the Service defines a set of Pods and creates a load balancer, of one of a few types, by which those Pods can be accessed.

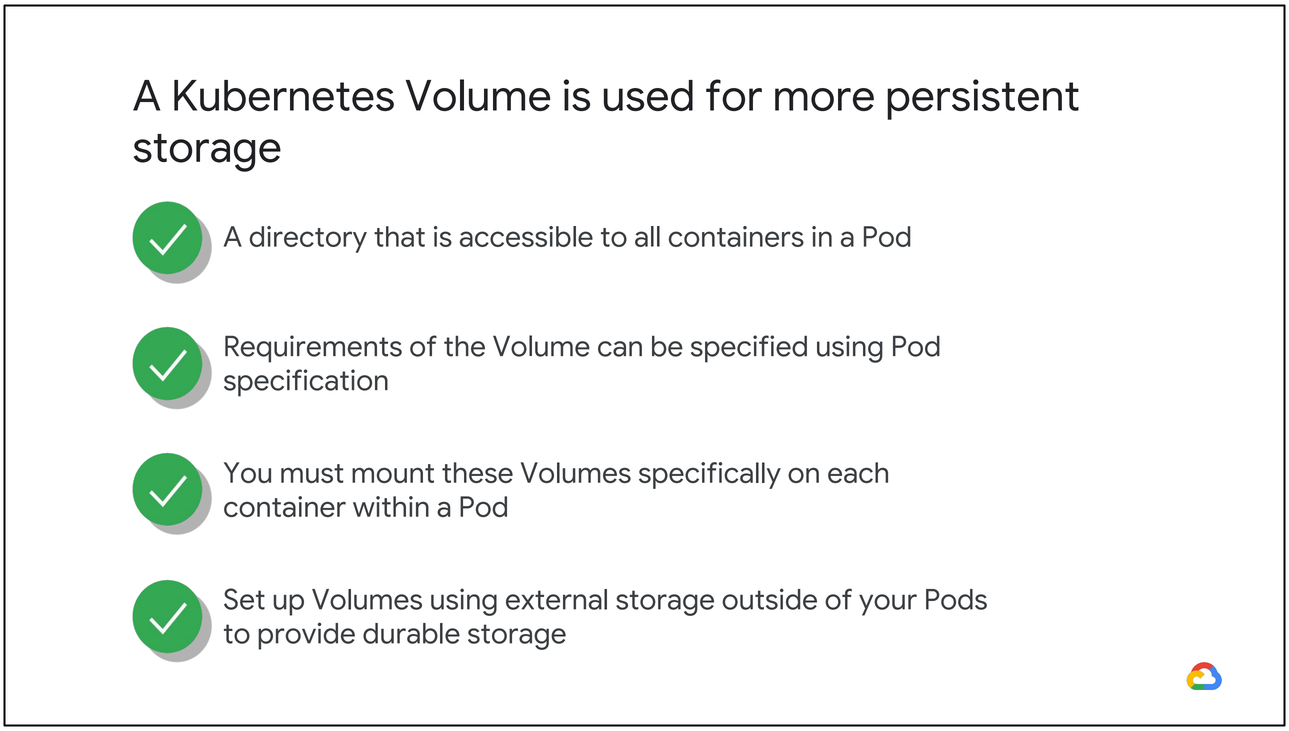
The Pods are selected using a label selector. By the way, you can also get a service quickly by asking Kubernetes to expose a Deployment. When you do that, Kubernetes handles selecting the right pods for you.

Whenever a Service is created, Kubernetes automatically creates endpoints for the selected Pods by creating endpoint resources.

By default, the master assigns a virtual IP address (also known as a ClusterIP) to the Service from internal IP tables. With GKE, this is assigned from the cluster's VPC network.

You will learn more about Services in a later module in this specialization. GKE offers other ways your Service can be exposed, not just through ClusterIPs.

Overall, a Service provides durable endpoints for Pods. These endpoints can be accessed by exposing the Service internally within a cluster, or externally to the outside world. The option to expose a Service internally or externally depends on the Service type itself. The frontend Pod can reliably access the backend Pods internally within the cluster using a Service.



A Kubernetes Volume is used for more persistent storage

- ✓ A directory that is accessible to all containers in a Pod
- ✓ Requirements of the Volume can be specified using Pod specification
- ✓ You must mount these Volumes specifically on each container within a Pod
- ✓ Set up Volumes using external storage outside of your Pods to provide durable storage

A container application can easily write data to the read/write layer inside the container. But it's ephemeral, so when the container terminates, whatever was written will be lost. What if you want to store data permanently? Or what if you need storage to be shared between tightly coupled containers within a Pod?

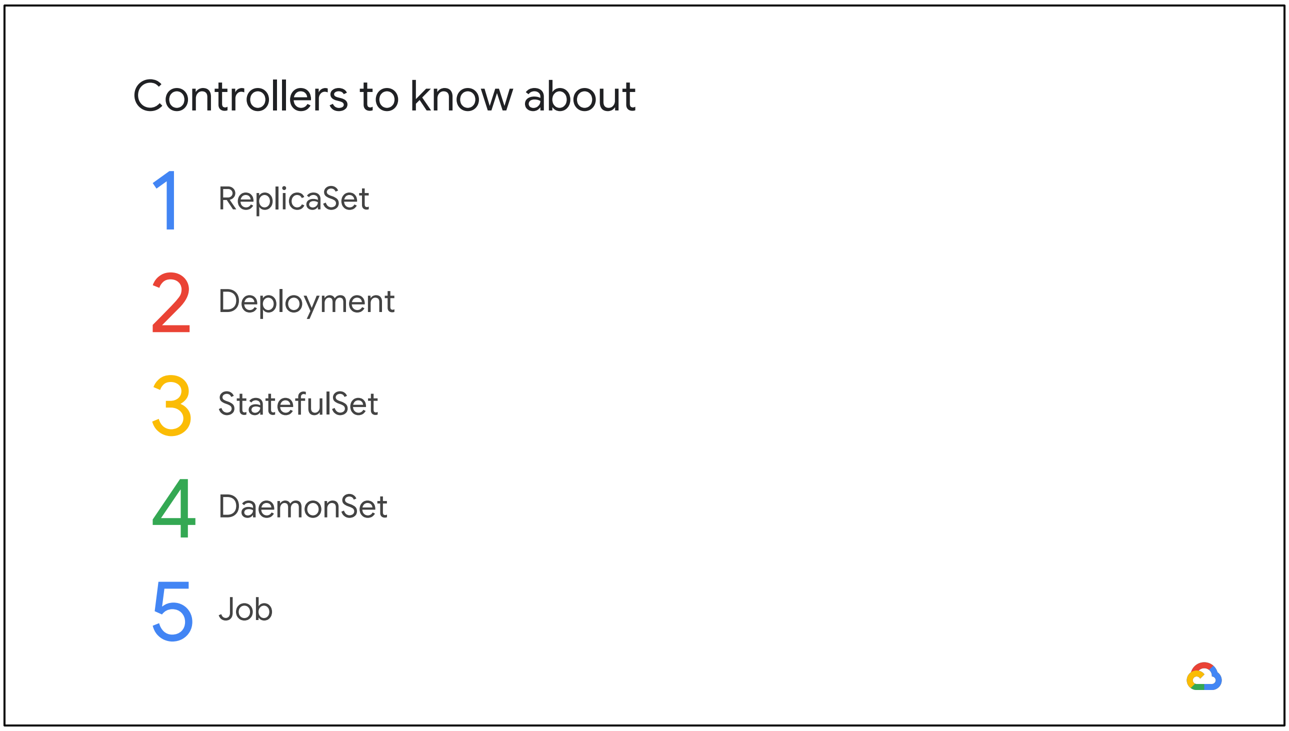
That's why a Kubernetes Volume is used for more persistent storage. Kubernetes Volume is another abstraction.

A Volume is simply a directory that is accessible to all the containers in a Pod.

The requirements for a Volume are defined through the Pod specification. This declares how the directory is created, what storage medium should be used, and its initial contents.

You don't want container failures or restarts don't affect the data within these Volumes. And you want your volume to be shared among multiple containers within a Pod. Docker containers have their own filesystem; therefore, in order to access these Volumes, they must be mounted specifically on each container within a Pod.

However, Pods themselves are also ephemeral. A failing node or deleted Pod could lead to its Volume being deleted too. To avoid this, you can configure Volumes using network based storage from outside of your Pods to provide durable storage that is not lost when a Pod or node fails. You’ll learn about Persistent Volumes later in this specialization.



Controllers to know about

1 ReplicaSet

2 Deployment

3 StatefulSet

4 DaemonSet

5 Job

Other advanced Kubernetes objects are discussed in more depth later in the course. Let’s look at the Controllers you should be aware of.

A ReplicaSet controller ensures that a population of Pods, all identical to one another, are running at the same time. Deployments let you do declarative updates to ReplicaSets and Pods. In fact, Deployments manage their own ReplicaSets to achieve the declarative goals you prescribe, so you will most commonly work with Deployment objects.

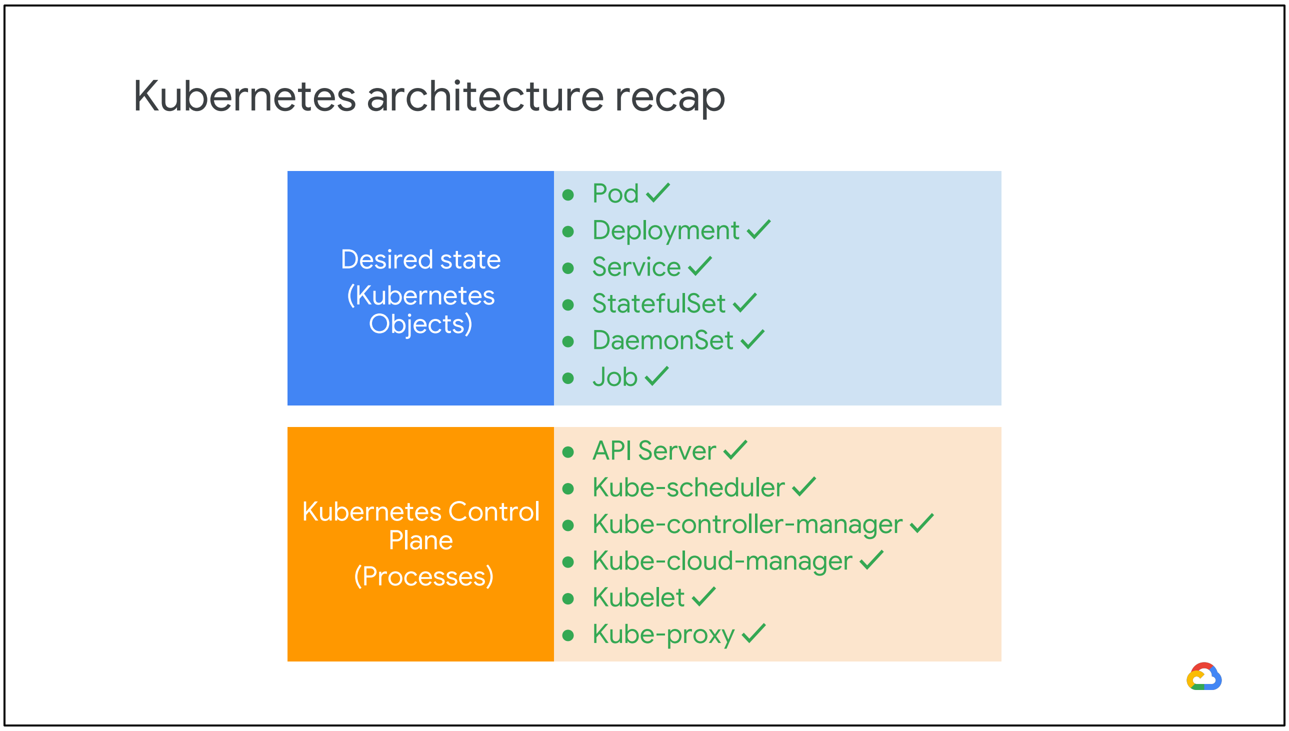
Deployments let you create, update, roll back, and scale Pods, using ReplicaSets as needed to do so. For example, when you perform a rolling upgrade of a Deployment, the Deployment object creates a second ReplicaSet, and then increases the number of Pods in the new ReplicaSet as it decreases the number of Pods in its original ReplicaSet.

Replication Controllers perform a similar role to the combination of ReplicaSets and Deployments, but their use is no longer recommended. Because Deployments provide a helpful "front end" to ReplicaSets, this training course chiefly focuses on Deployments.

If you need to deploy applications that maintain local state, StatefulSet is a better option. A StatefulSet is similar to a Deployment in that the Pods use the same container spec. The Pods created through Deployment are not given persistent identities, however; by contrast, Pods created using StatefulSet have unique persistent identities with stable network identity and persistent disk storage.

If you need to run certain Pods on all the nodes within the cluster or on a selection of nodes, use DaemonSet. DaemonSet ensures that a specific Pod is always running on all or some subset of the nodes. If new nodes are added, DaemonSet will automatically set up Pods in those nodes with the required specification. The word "daemon" is a computer science term meaning a non-interactive process that provides useful services to other processes. A Kubernetes cluster might use a DaemonSet to ensure that a logging agent like fluentd is running on all nodes in the cluster.

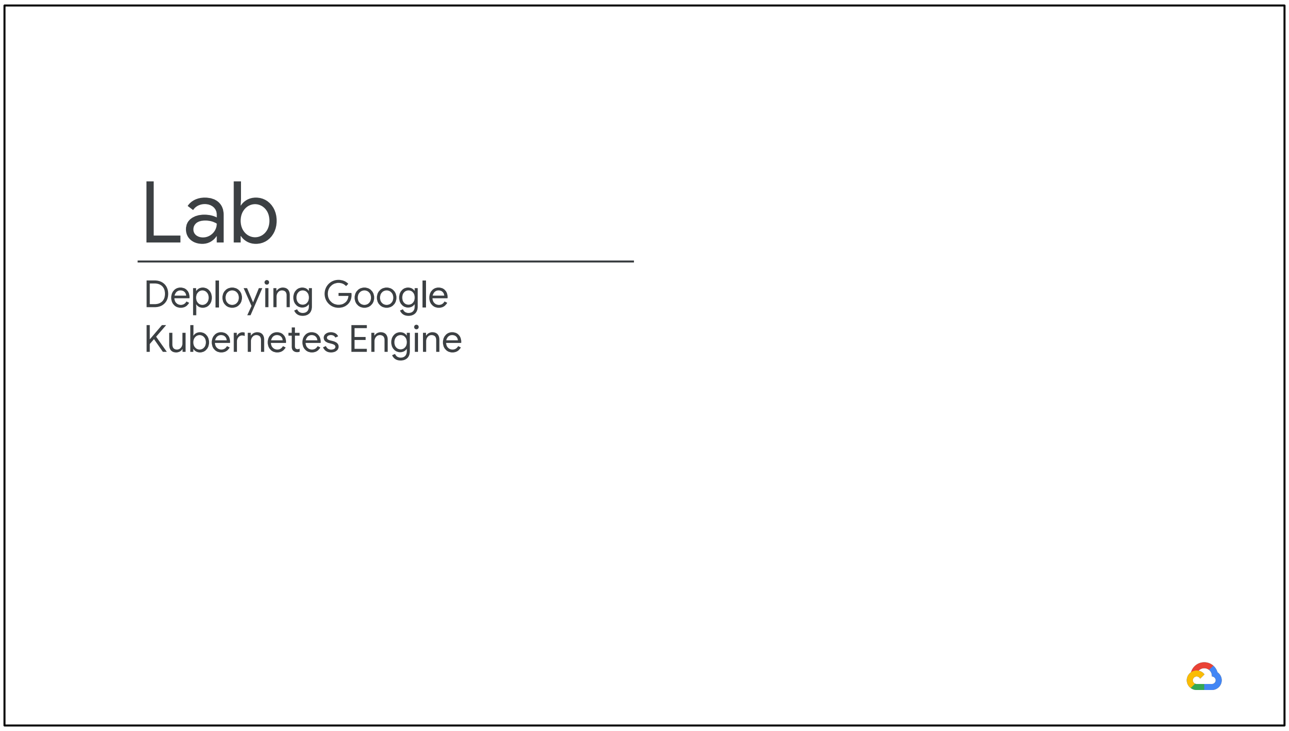
The Job controller creates one or more Pods required to run a task. When the task is completed, Job will then terminate all those Pods. A related controller is CronJob, which runs Pods on a time-based schedule.



Kubernetes architecture recap

Desired state (Kubernetes Objects)	• Pod ✓ • Deployment ✓ • Service ✓ • StatefulSet ✓ • DaemonSet ✓ • Job ✓
Kubernetes Control Plane (Processes)	• API Server ✓ • Kube-scheduler ✓ • Kube-controller-manager ✓ • Kube-cloud-manager ✓ • Kubelet ✓ • Kube-proxy ✓

That wraps up this lesson. You've learned how the desired state is declared through Kubernetes objects, such as Pods and Deployments. You also saw how different components of the Control Plane work in coordination to achieve the desired state of the cluster.



Lab

Deploying Google Kubernetes Engine

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 GCP Console to build and manipulate GKE clusters, deploy a Pod, and examine the cluster and Pods.