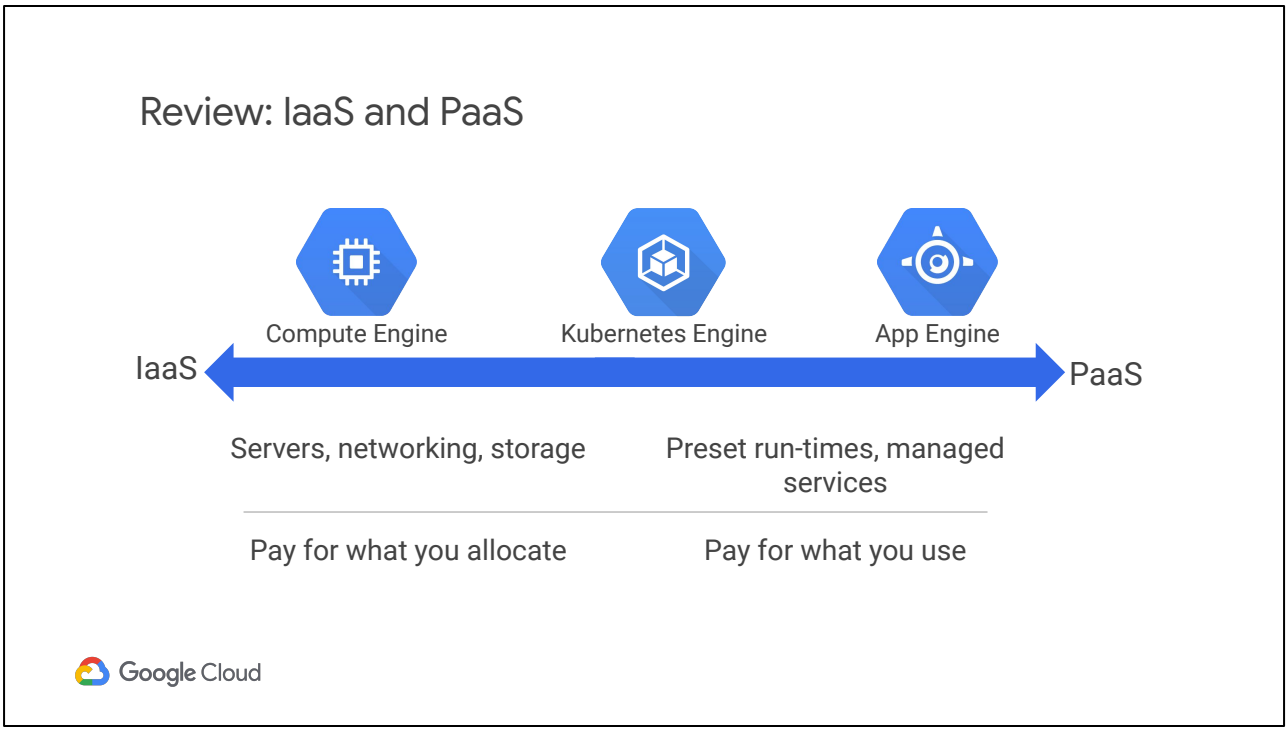
Google Cloud

GCP Fundamentals: Core Infrastructure

Containers in the Cloud



We've already discussed **Compute Engine**, which is GCP's **Infrastructure as a Service** offering, with access to servers, file systems, and networking.

And **App Engine** which is GCP's **PaaS** offering.

Now I'm going to introduce you to containers and Kubernetes Engine which is a hybrid which conceptually sits between the two and benefits from both.

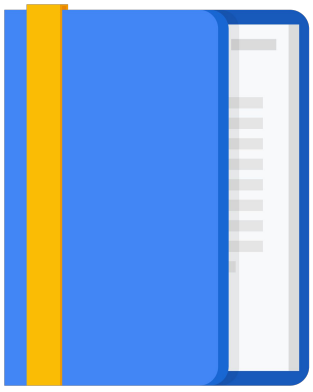
Agenda

Introduction to Containers

Kubernetes and Kubernetes Engine

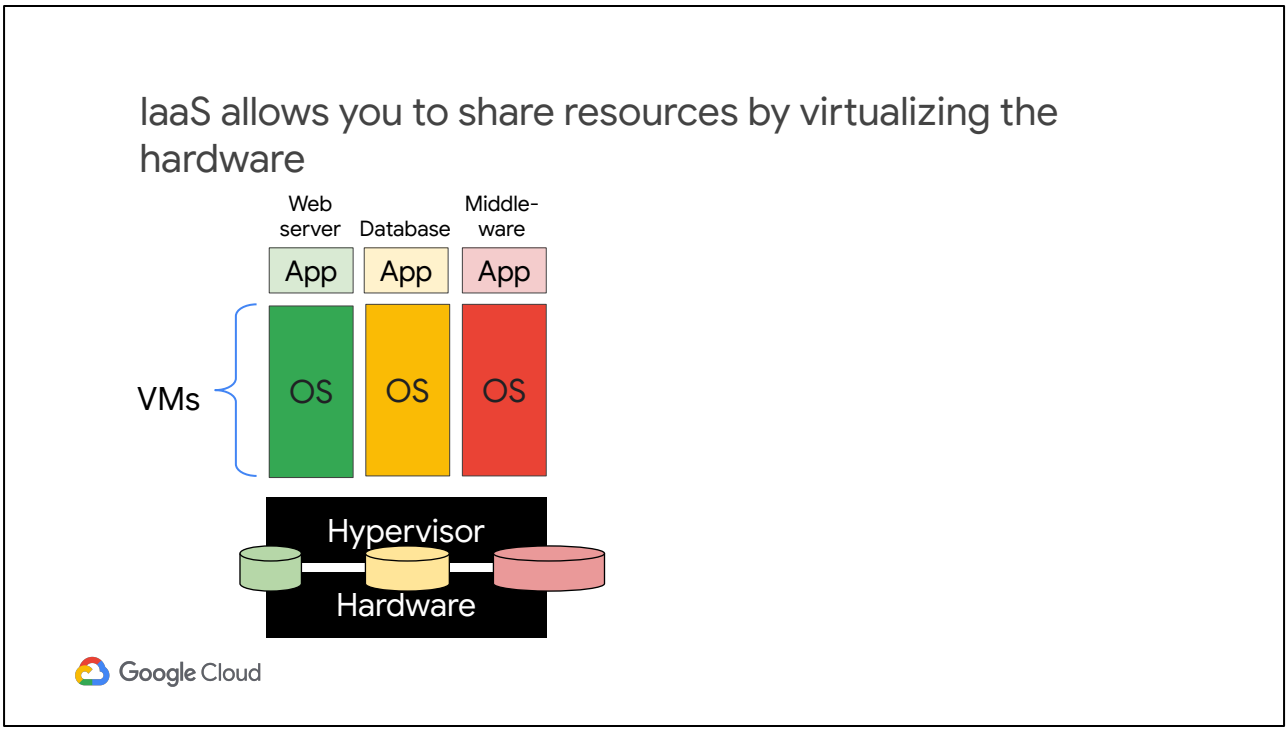
Hybrid and Multi Cloud

Quiz and Lab



Google Cloud

I'll describe why you want to use containers and how to manage them in **Kubernetes Engine**.

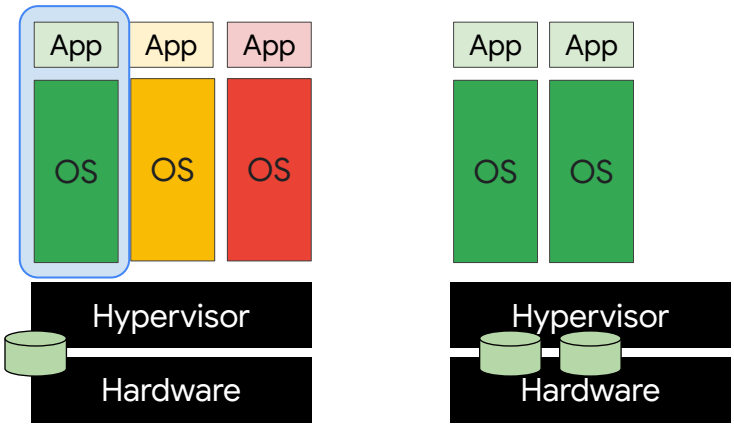


Let's begin, by remembering that **Infrastructure as a Service** allows you to share compute resources with other developers by **virtualizing the hardware** using **virtual machines**. Each developer can deploy their own operating system, access the hardware, and build their applications in a self-contained environment with access to RAM, file systems, networking interfaces, and so on.

But flexibility comes with a cost. The smallest unit of compute is an app with its **VM**. The guest OS may be large, even gigabytes in size, and takes minutes to boot.

But you have your tools of choice on a configurable system. So you can install your favorite runtime, web server, database, or middleware, configure the underlying system resources, such as disk space, disk I/O, or networking and build as you like.

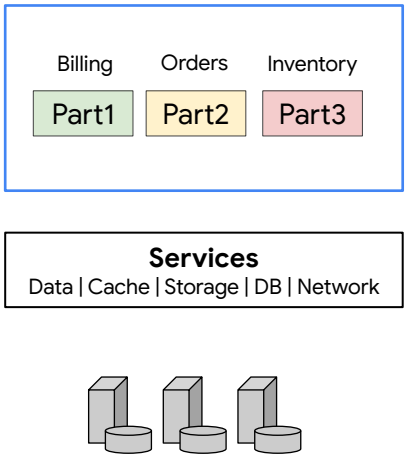
But flexibility has costs in boot time (minutes) and resources (Gigabytes)



Google Cloud

However, as demand for your application increases, you have to copy an entire VM and boot the guest OS for each instance of your app, which can be slow and costly.

App Engine provides access to programming services

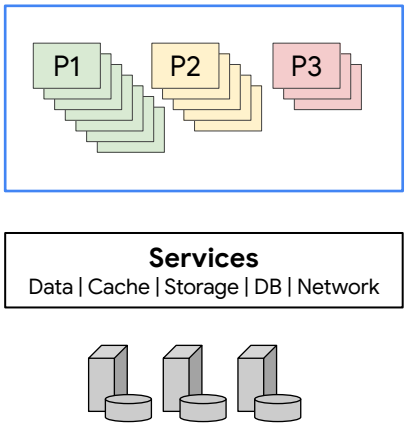


Google Cloud

With App engine, you get access to programming services.

So all you do is write your code in self-contained **workloads** that use these services and include any dependent libraries.

As demand for your app increases, the platform scales your app rapidly and independently by workload and infrastructure



Google Cloud

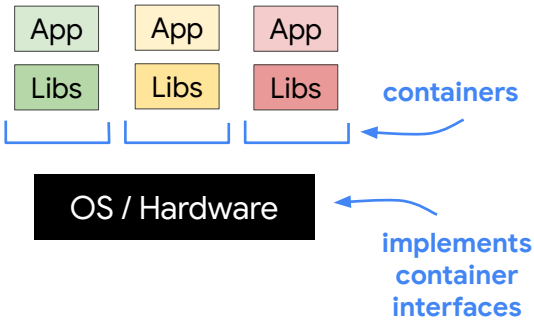
As demand for your app increases, the platform scales your app seamlessly and independently by workload and infrastructure.

This scales rapidly but you won't be able to fine-tune the underlying architecture to save cost.

Containers offer IaaS flexibility and PaaS scalability

Containers provide:

- An abstraction layer of the hardware and OS
- An invisible box with configurable access to isolated partitions of file system, RAM, and networking
- Fast startup (only a few system calls)



Google Cloud

That's where containers come in.

The idea of a container is to give you the independent scalability of workloads in PaaS and an abstraction layer of the OS and hardware in IaaS.

What you get is an invisible box around your code and its dependencies, with limited access to its own partition of the file system and hardware.

It only requires a few system calls to create and it starts as quickly as a process.

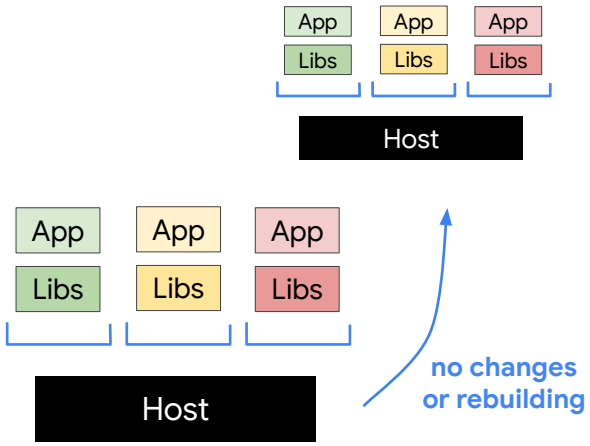
All you need on each host is an OS kernel that supports containers and a container runtime.

In essence, you are virtualizing the OS. It scales like PaaS, but gives you nearly the same flexibility as IaaS.

Containers are configurable, self-contained, and ultra-portable

With containers, you can:

- Define your own hardware, OS, and software stack configurations
- Treat the OS and hardware as a black box and go from dev, to staging, to production, or your laptop to the cloud, without changing or rebuilding anything



Google Cloud

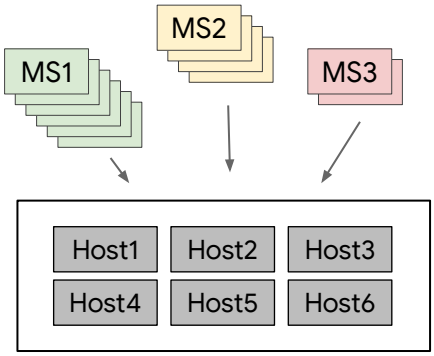
With this abstraction, your code is ultra portable and you can treat the OS and hardware as a black box.

So you can go from development, to staging, to production, or from your laptop to the cloud, without changing or rebuilding anything.

With a common host configuration, you can deploy containers on a group of servers called a cluster

With a cluster, you can:

- Connect containers using network connections
- Build code modularly
- Deploy it easily
- Scale containers and hosts independently for maximum efficiency and savings



Google Cloud

If you want to scale, for example, a web server, you can do so in seconds and deploy dozens or hundreds of them (depending on the size or your workload) on a single host.

Now that's a simple example of scaling one container running the whole application on a single host.

You'll likely want to build your applications using lots of containers, each performing their own function like **microservices**.

If you build them this way, and connect them with network connections, you can make them modular, deploy easily, and scale independently across a group of **hosts**.

And the hosts can scale up and down and start and stop containers as demand for your app changes or as hosts fail.

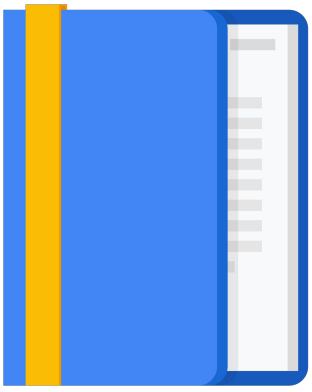
Agenda

Introduction to Containers

Kubernetes and Kubernetes Engine

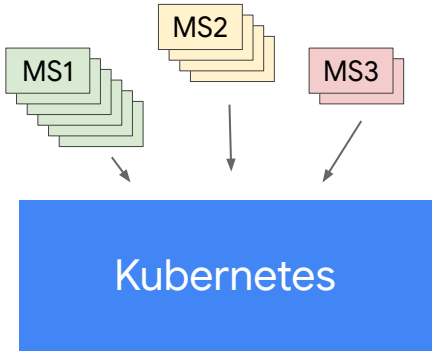
Hybrid and Multi Cloud

Quiz and Lab



Google Cloud

Kubernetes makes it easy to orchestrate many containers on many hosts



Google Cloud

A tool that helps you do this well is **Kubernetes**.

Kubernetes makes it easy to orchestrate many containers on many hosts, scale them as microservices, and deploy rollouts and rollbacks.

First, I'll show you how you build and run containers.

Let's begin by building and running an app as a container

- We'll use an open-source tool called **Docker** that bundles your app, its dependencies, and system settings
  - You could use another tool like **Google Cloud Build**
- Here is an example of some code you may have written:
  - It's a Python app that says 'hello world'
  - Or if you hit the second endpoint, it gives you the version

Google Cloud

app.py

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
  return "Hello World!\n"

@app.route("/version")
def version():
  return "Helloworld 1.0\n"

if __name__ == "__main__":
  app.run(host='0.0.0.0')
```

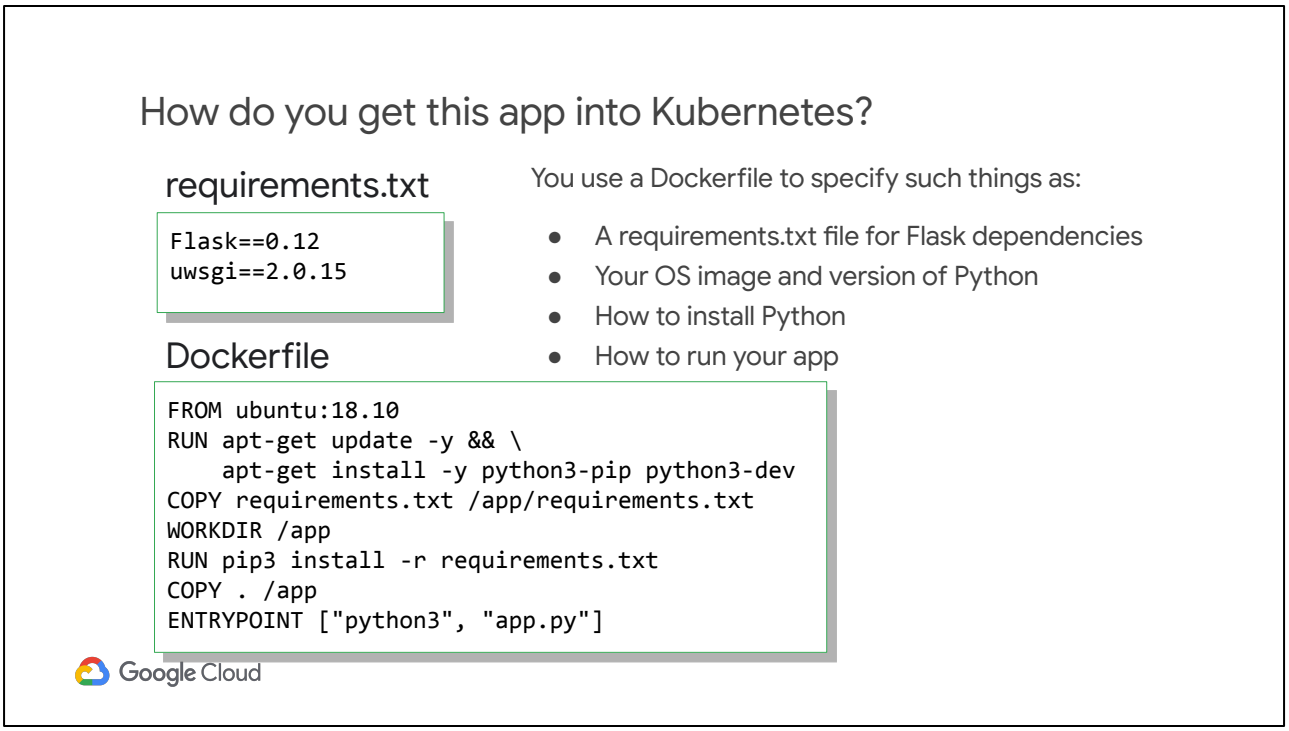
I'll use an open-source tool called **Docker** that defines a format for bundling your application, its dependencies, and machine-specific settings into a container; you could use a different tool like **Google Container Builder**. It's up to you.

Here is an example of some code you may have written.

It's a Python app.

It says "Hello World"

Or if you hit the second endpoint '/version', it gives you the version.



So how do you get this app into Kubernetes?

You have to think about your version of Python, what dependency you have on Flask, how to use the requirements.txt file, how to install Python, and so on.

So you use a **Dockerfile** to specify how your code gets packaged into a container.

For example, if you're a developer, and you're used to using Ubuntu with all your tools, you start there.

You can install Python the same way you would on your dev environment.

You can take that requirements file from Python that you know.

And you can use tools inside **Docker** or **Cloud Build** to install your dependencies the way you want.

Eventually, it produces an app, and you run it with the ENDPOINT command.

Then you build and run the container as an image

```
$> docker build -t py-server .
$> docker run -d py-server
```

- **`docker build`** builds a container and stores it locally as a runnable image
- You can upload images to a registry service (like Google Container Registry) for sharing
- **`docker run`** starts the container image

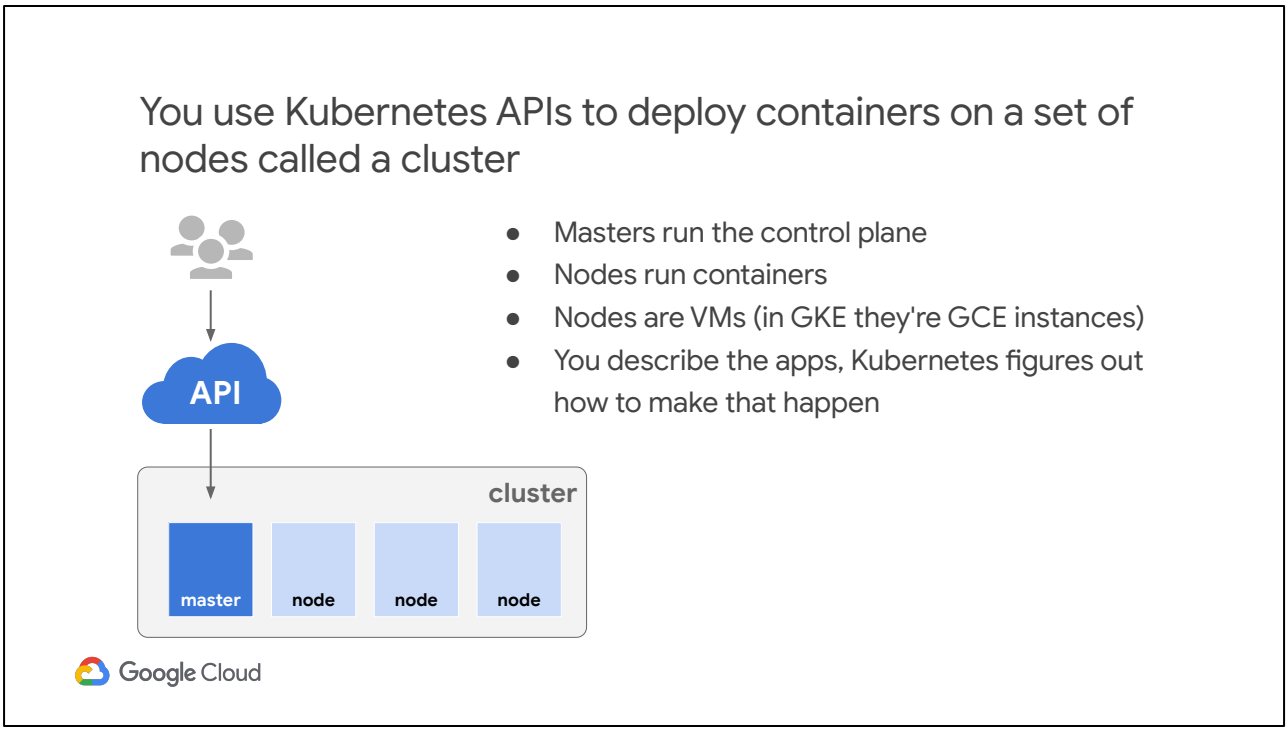
Google Cloud

Then you use the "**docker build**" command to build the container.

This builds the container and stores it locally as a runnable **image**. You can save and upload the image to a container registry service and share or download it from there.

Then you use the "**docker run**" command to run the image.

As it turns out, packaging applications is only about 5% of the issue. The rest has to do with: application configuration, service discovery, managing updates, and monitoring. These are the components of a reliable, scalable, distributed system.



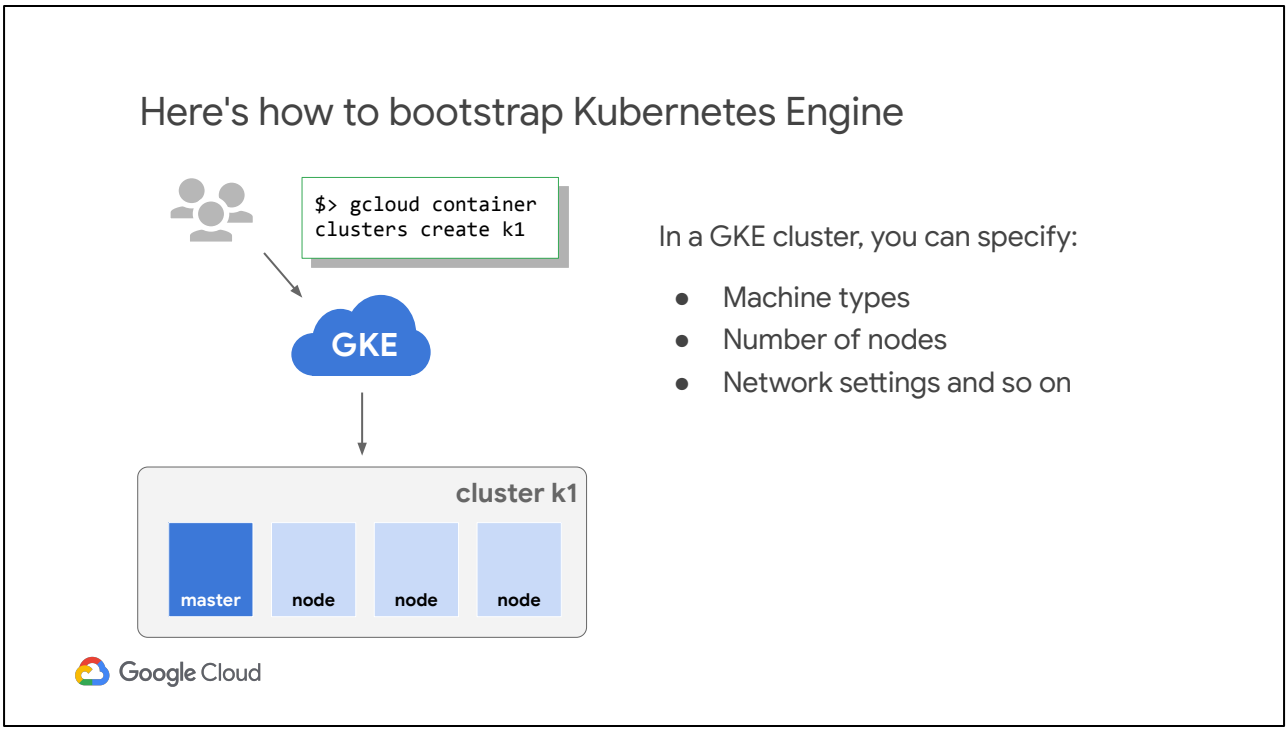
Now, I'll show you where **Kubernetes** comes in.

Kubernetes is an open-source **orchestrator** that abstracts containers at a higher level so you can better manage and scale your applications.

At the highest level, Kubernetes is a set of APIs that you can use to deploy containers on a set of **nodes** called a **cluster**.

The system is divided into a set of **master** components that run as the control plane and a set of **nodes** that run containers. In Kubernetes, a node represents a computing instance, like a machine. In Google Cloud, nodes are virtual machines running in Compute Engine.

You can describe a set of applications and how they should interact with each other and Kubernetes figures how to make that happen



Now that you've built a container, you'll want to deploy one into a **cluster**.

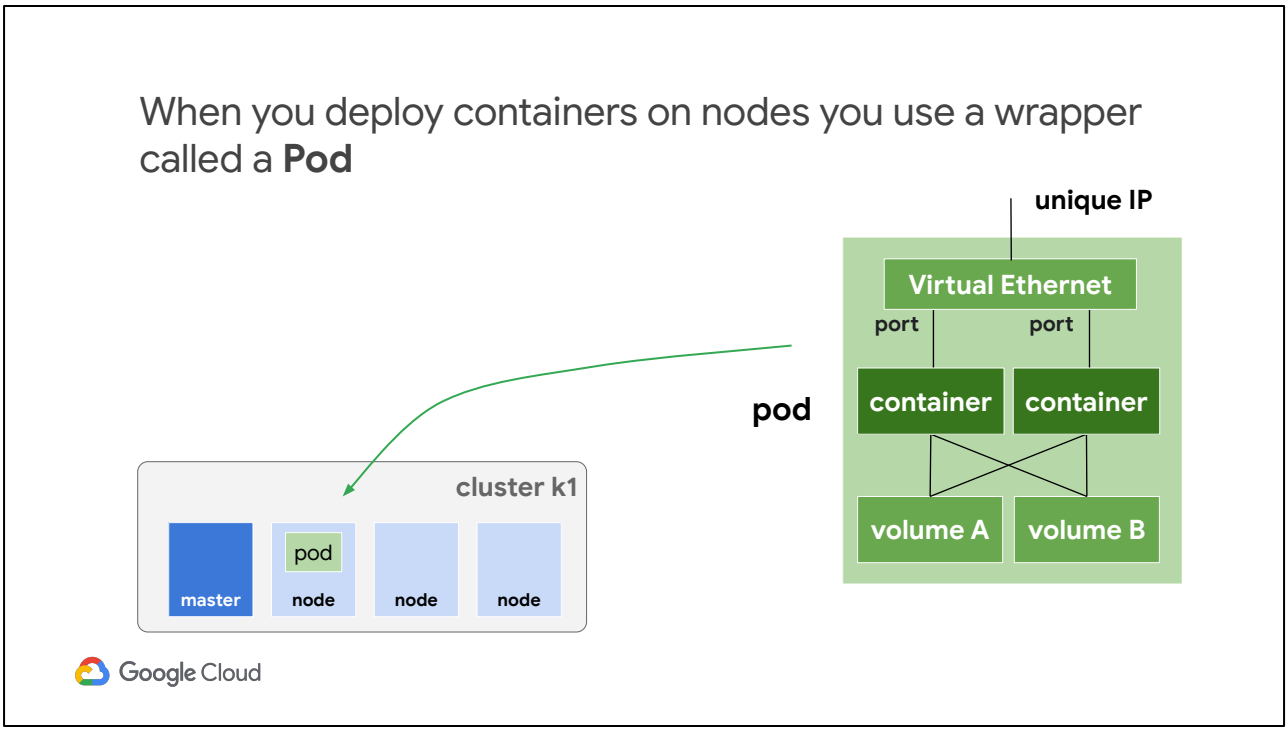
Kubernetes can be configured with many options and add-ons, but can be time consuming to bootstrap from the ground up. Instead, you can bootstrap Kubernetes using **Kubernetes Engine** or (GKE).

GKE is a hosted Kubernetes by Google. GKE clusters can be customized and they support different machine types, number of nodes, and network settings.

To start up Kubernetes on a **cluster** in GKE, all you do is run this command: **`$> gcloud container clusters create k1`**

At this point, you should have a cluster called 'k1' configured and ready to go.

You can check its status in admin console.



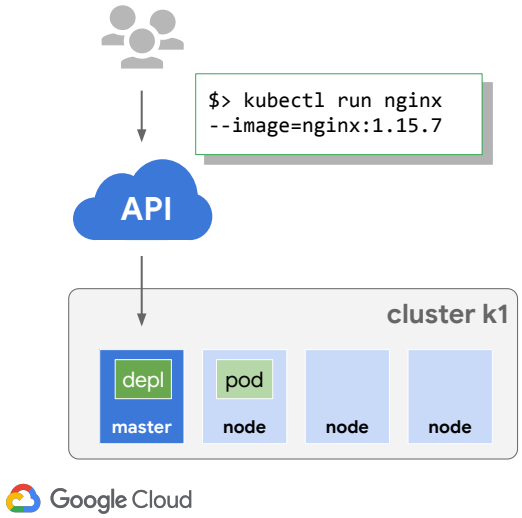
Then you deploy **containers** on nodes using a wrapper around one or more containers called a Pod.

A **Pod** is the smallest unit in Kubernetes that you create or deploy. A Pod represents a running process on your cluster as either a component of your application or an entire app.

Generally, you only have one container per pod, but if you have multiple containers with a hard dependency, you can package them into a single pod and share networking and storage. The Pod provides a unique network IP and set of ports for your containers, and options that govern how containers should run.

Containers inside a Pod can communicate with one another using localhost and ports that remain fixed as they're started and stopped on different nodes.

You can run a container in a Pod using `kubectl run`



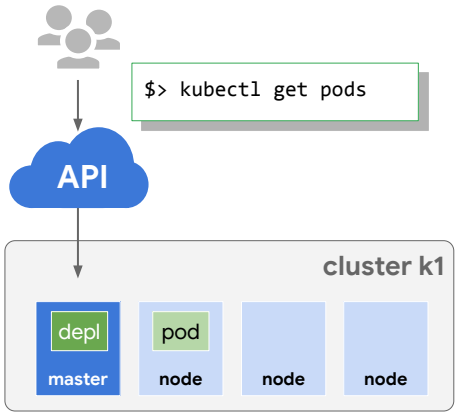
- **`kubectl`** is the command-line client to the Kubernetes API
- This command starts a **Deployment** with a container running in a Pod
- In this case, the container is an image of the NGINX server

Google Cloud

One way to run a container in a Pod in Kubernetes is to use the **kubectl run** command. We'll learn a better way later in this module, but this gets you started quickly.

This starts a **Deployment** with a container running in a **Pod** and in this case, the container inside the Pod is an image of the nginx server.

You use a **Deployment** to manage a set of replica Pods for an app or workload and make sure the desired number is running and healthy



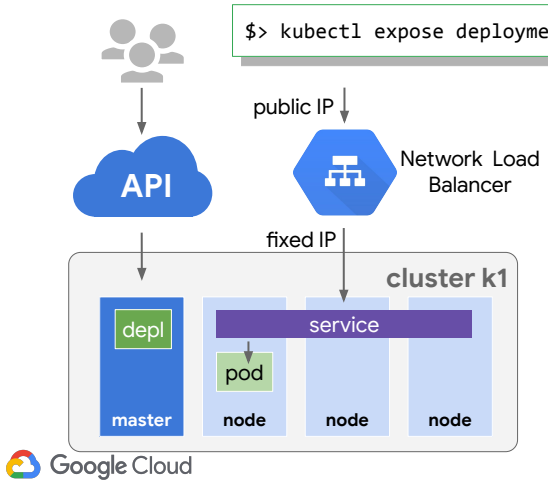
Google Cloud

A **Deployment** represents a group of replicas of the same Pod and keeps your Pods running even when nodes they run on fail. It could represent a component of an application or an entire app. In this case, it's the nginx web server.

To see the running nginx Pods, run the command:

```
$ kubectl get pods
```

By default, Pods are only available inside a cluster and they get ephemeral IPs



- To make them publicly available with a fixed IP, you can connect a load balancer to your Deployment running **`kubectl expose`**
- Kubernetes creates a Service with a fixed IP for your Pods, and a controller says "I need to attach an external load balancer with a public IP address"

By default, Pods in a Deployment are only accessible inside your GKE cluster.

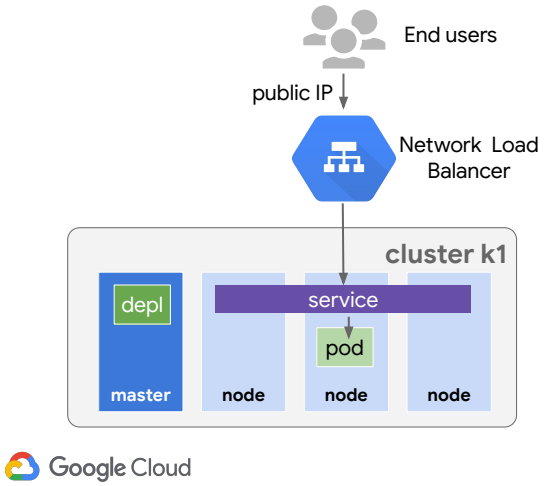
To make them publicly available, you can connect a load balancer to your Deployment by running the **kubectl expose** command:

Kubernetes creates a **Service** with a fixed IP for your Pods,

and a controller says "I need to attach an external **load balancer** with a public IP address to that **Service** so others outside the cluster can access it".

In **GKE**, the load balancer is created as a **Network Load Balancer**.

Any client hitting that IP will be routed to a Pod behind the Service



Google Cloud

- For example, if you create two sets of Pods called frontend and backend, and put them behind their own Services, backend Pods may change, but frontend Pods are not aware of this. They simply refer to the backend Service.

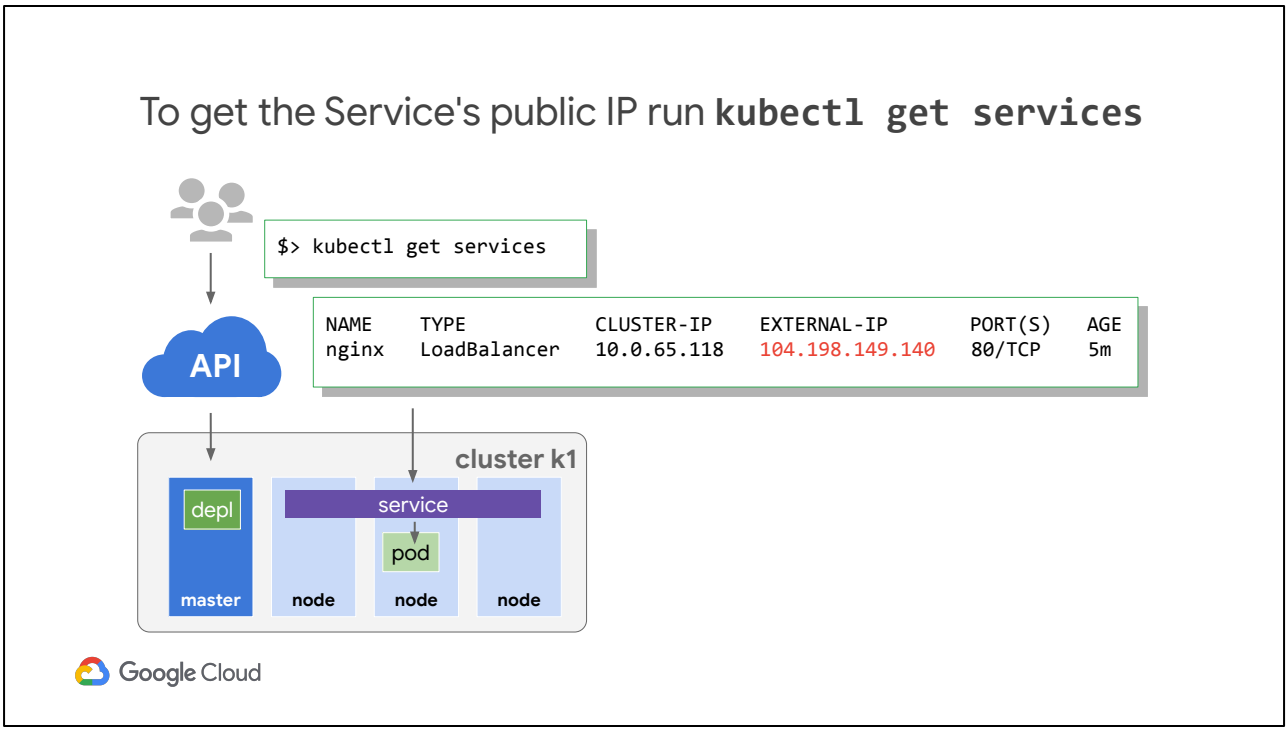
Any client that hits that IP address will be routed to a Pod behind the Service, in this case there is only one--your simple nginx Pod.

A **Service** is an abstraction which defines a logical set of Pods and a policy by which to access them.

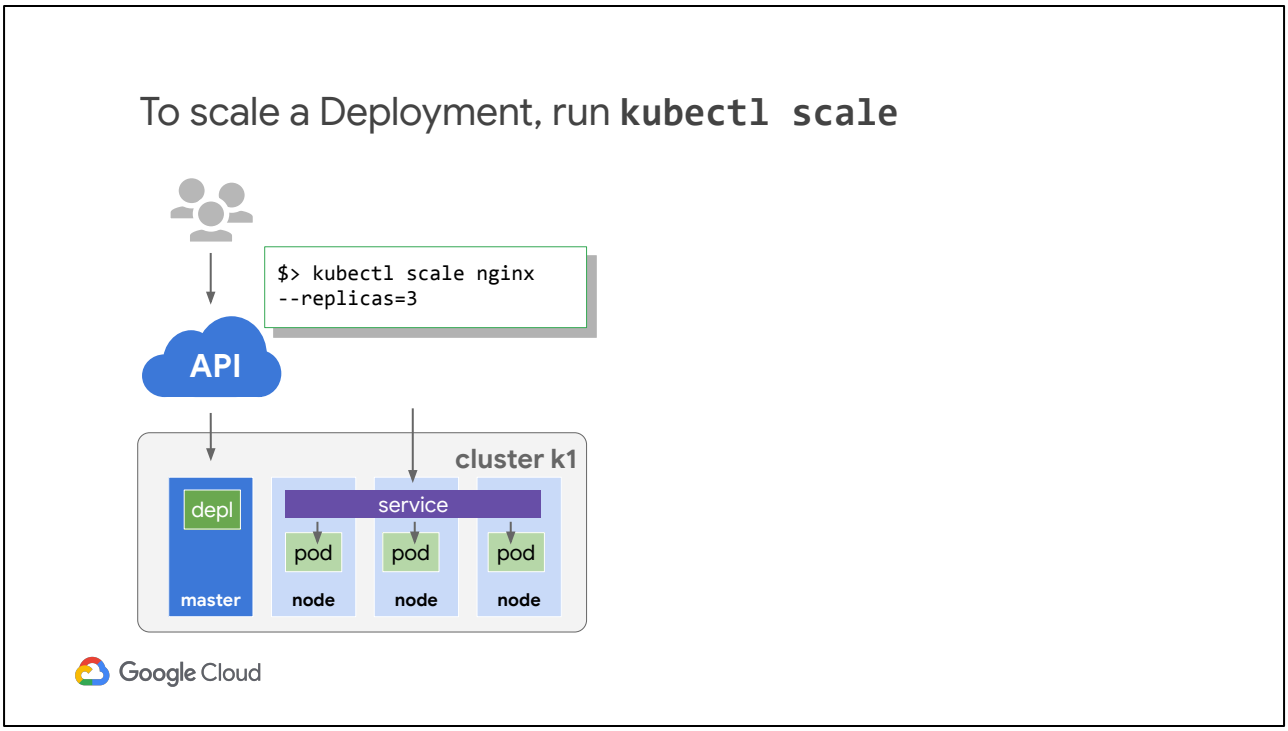
As Deployments create and destroy Pods, Pods get their own IP address. But those addresses don't remain stable over time.

A Service groups a set of Pods and provides a stable endpoint (or fixed IP) for them.

For example, if you create two sets of Pods called frontend and backend, and put them behind their own Services, backend Pods may change, but frontend Pods are not aware of this. They simply refer to the backend Service.



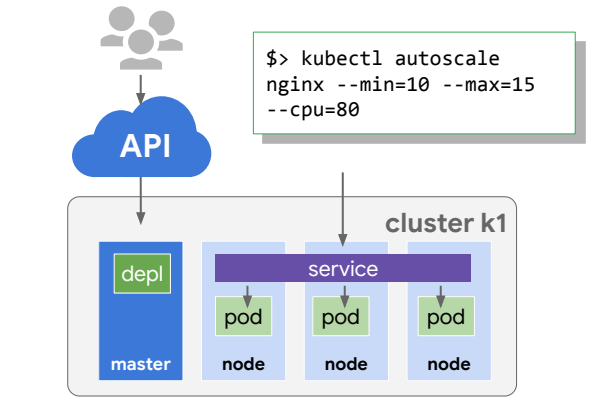
You can run the **kubectl get services** command to get the public IP to hit the nginx container remotely.



To scale a Deployment, run the **kubectl scale** command.

In this case, three Pods are created in your Deployment and they're placed behind the Service and share one fixed IP.

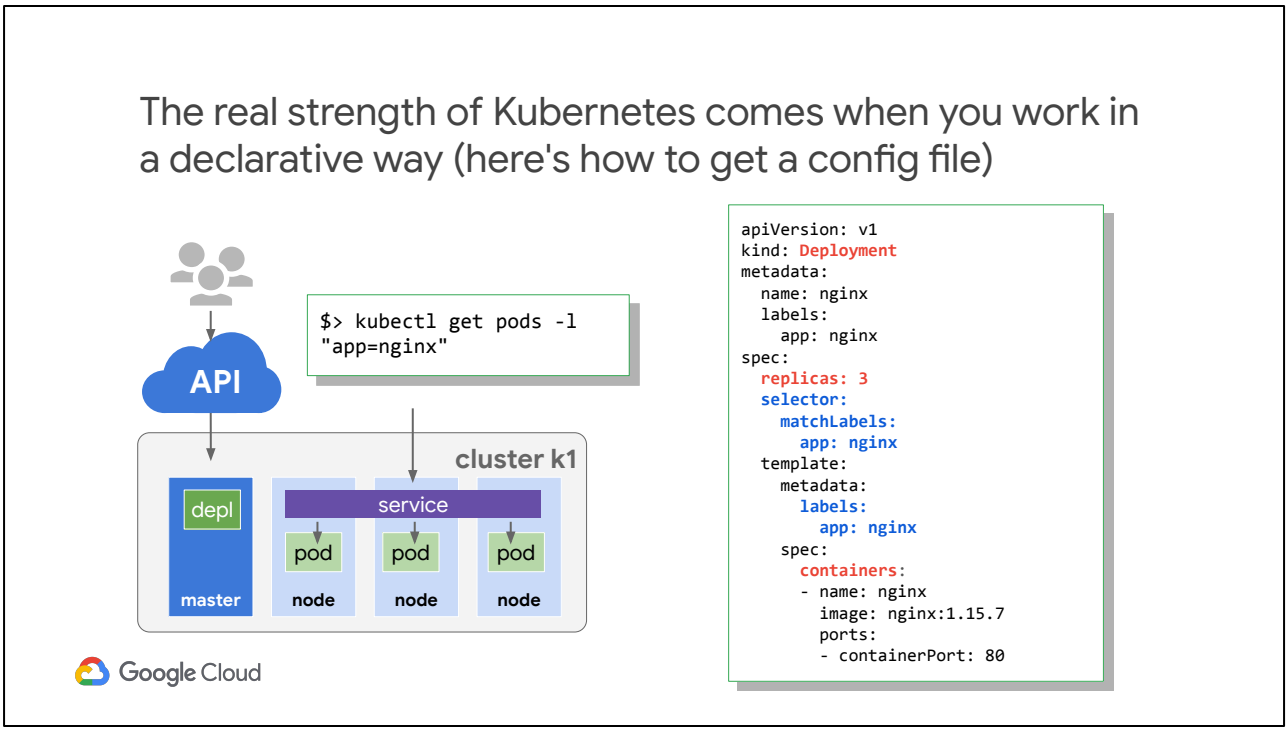
You can also run autoscaling with all kinds of parameters or put it behind programming logic for intelligent management



Google Cloud

You could also use **autoscaling** with all kinds of parameters.

Here's an example of how to autoscale the Deployment to between 10 and 15 Pods when CPU utilization reaches 80 percent.



So far, I've shown you how to run **imperative** commands like **expose** and **scale**. This works well to learn and test Kubernetes step-by-step.

But the real strength of Kubernetes comes when you work in a **declarative** way.

Instead of issuing commands, you provide a configuration file that tells Kubernetes what you want your desired state to look like, and Kubernetes figures out how to do it.

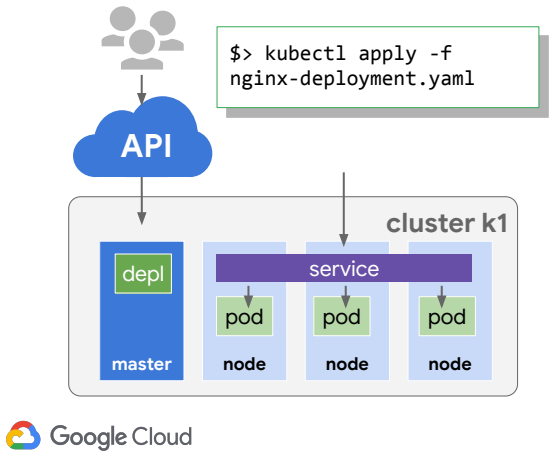
Let me show you how to scale your Deployment using an existing Deployment config file.

To get the file, you can run a kubectl get pods command like the following, and you'll get a Deployment configuration file like the following.

In this case, it declares you want three replicas of your nginx Pod.

It defines a **selector** field so your Deployment knows how to group specific Pods as replicas, and you add a **label** to the Pod template so they get selected.

To declaratively apply changes run **kubectl apply -f**

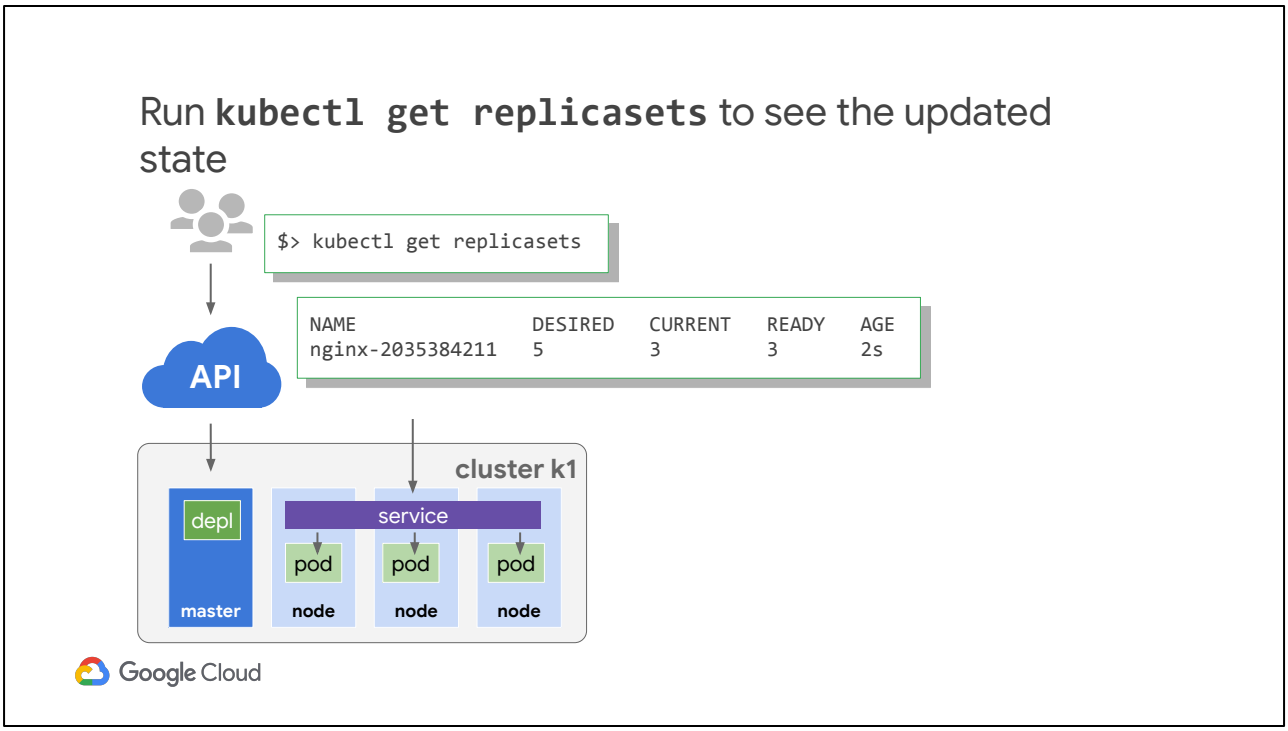



```
apiVersion: v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 5
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.10.0
        ports:
        - containerPort: 80
```

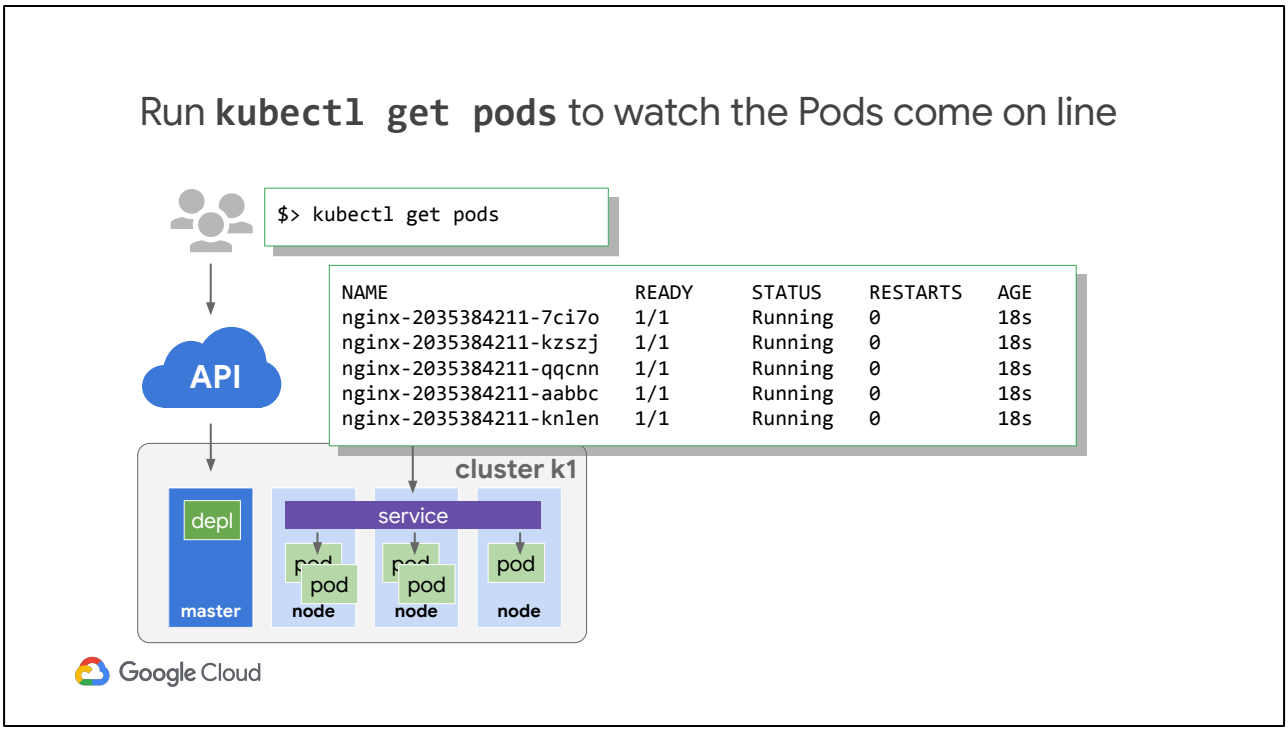
Google Cloud

To run five replicas instead of three, all you do is update the Deployment config file.

And run the **kubectl apply** command to use the config file.

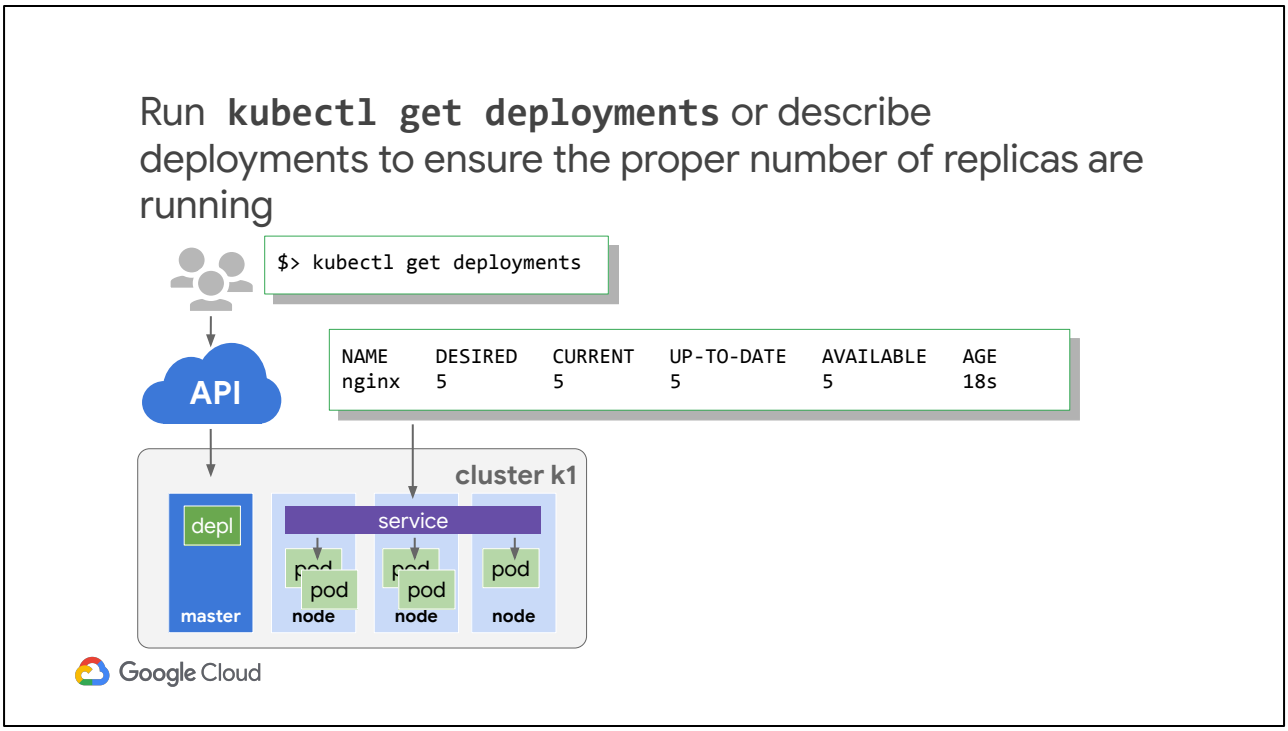


Now look at your replicas to see their updated state.

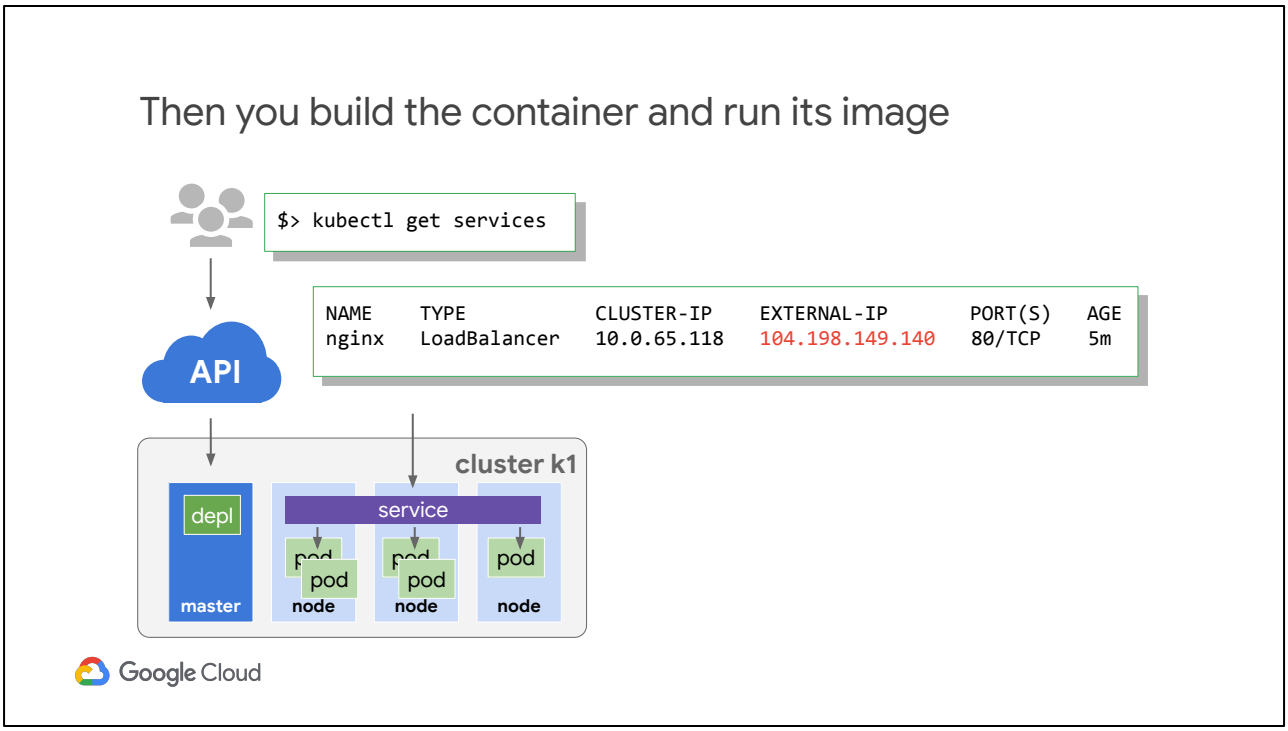


Then use the **kubectl get pods** command to watch the pods come on line.

In this case, all five are **READY** and **RUNNING**.



And check the Deployment to make sure the proper number of replicas are running using either $ kubectl **get deployments** or $ kubectl **describe deployments**. In this case, all five Pod replicas are **AVAILABLE**.



And you can still hit your endpoint like before using $ kubectl **get services** to get the external IP of the Service, and hit the public IP from a client.

At this point, you have five copies of your nginx Pod running in GKE, and you have a single Service that's proxying the traffic to all five Pods. This allows you to share the load and scale your Service in Kubernetes.

To update to a new version of your app, you can use a variety of rollout strategies

```
spec:
  # ...
  replicas: 5
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
  # ...
```

The last question is what happens when you want to update a new version of your app?

You want to update your container to get new code out in front of users, but it would be risky to roll out all those changes at once.

So you use **kubectl rollout** or change your deployment configuration file and apply the change using **kubectl apply**.

New Pods will be created according to your update strategy. Here is an example configuration that will create new version Pods one by one, and wait for a new Pod to be available before destroying one of the old Pods.

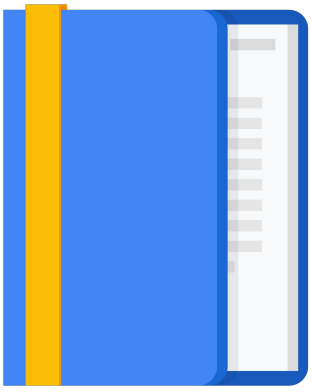
Agenda

Introduction to Containers

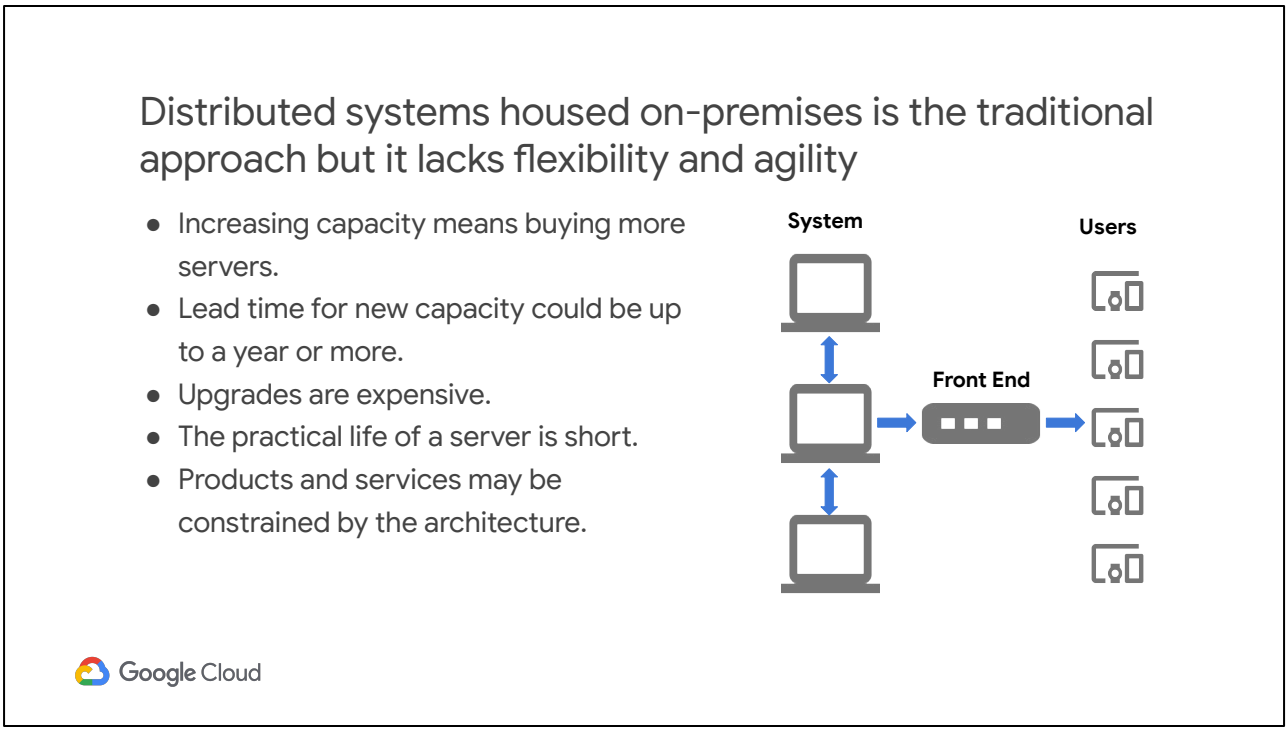
Kubernetes and Kubernetes Engine

Hybrid and Multi-Cloud

Quiz and Lab



Google Cloud



Now that you understand containers, let's take that understanding a step further and talk about using them in a modern hybrid cloud and multi-cloud architecture.

But before we do that, however, let's have a quick look at a typical on-premises distributed systems architecture, which is how businesses traditionally met their enterprise computing needs before Cloud computing.

As you may know, most enterprise-scale applications are designed as distributed systems, spreading the computing workload required to provide services over two or more networked servers. Over the past few years, containers have become a popular way to break these workloads down into "microservices" so they can be more easily maintained and expanded.

Traditionally, these enterprise systems (and their workloads, containerized or not) have been housed on-premises, which means they are housed on a set of high-capacity servers running somewhere within the company's network, or within a company-owned data center.

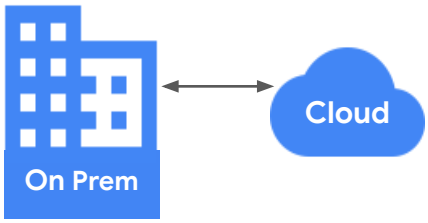
When an application's computing needs begin to outstrip its available computing resources, a company using on-premises systems would need to requisition more (or more powerful) servers, install them on the company network (after any necessary

network changes or expansion), configure the new servers, and finally load the application and its dependencies onto the new servers, before resource bottlenecks could be resolved.

The time required to complete an on-premises upgrade of this kind could be anywhere from several months to one or more *years*. It may also be quite costly, especially when you consider the useful lifespan of the average server is only 3-5 years.

Modern distributed systems allow a more agile approach to managing your compute resources

- Move only some of your compute workloads to the Cloud if you wish.
- Migrate these workloads at your own pace.
- Quickly take advantage of Cloud's flexibility, scalability and lower computing costs.
- Add specialized services to your compute resources stack.



Google Cloud

But what if you need more computing power *now*, not *months* from now?

What if your company wants to begin to relocate some workloads away from on-premises, to the cloud, to take advantage of lower costs and higher availability, but is unwilling (or unable) to move the entire enterprise application from the on-premises network?

What if you want to use specialized products and services that are only available in the Cloud?

This is where a modern hybrid or multi-cloud architecture can help. It allows you to:

- Keep parts of your systems infrastructure on-premises, while moving other parts to the Cloud, creating an environment that is uniquely suited to your company's needs.
- Move only specific workloads to the Cloud at your own pace, because a full scale migration is not required for it to work.
- Take advantage of the flexibility, scalability and lower computing costs offered by Cloud services for running the workloads you decide to migrate.
- Add specialized services such as machine learning, content caching, data analysis, long-term storage, and IoT to your computing resources toolkit.

Anthos is Google's modern solution for hybrid and multi-cloud systems and services management

- Kubernetes and GKE On-Prem create the foundation.
- On-premises and Cloud environments stay in sync.
- A rich set of tools is provided for:
  - Managing services on-premises and in the Cloud.
  - Monitoring systems and services.
  - Migrating applications from VMs into your clusters.
  - Maintaining consistent policies across all clusters, whether on-premises or in the Cloud.

Google Cloud

You may have heard a lot of discussions recently concerning the adoption of "hybrid" architecture for powering distributed systems and services. You may have even heard discussion of Google's answer to modern hybrid and multi-cloud distributed systems and services management, called "Anthos."

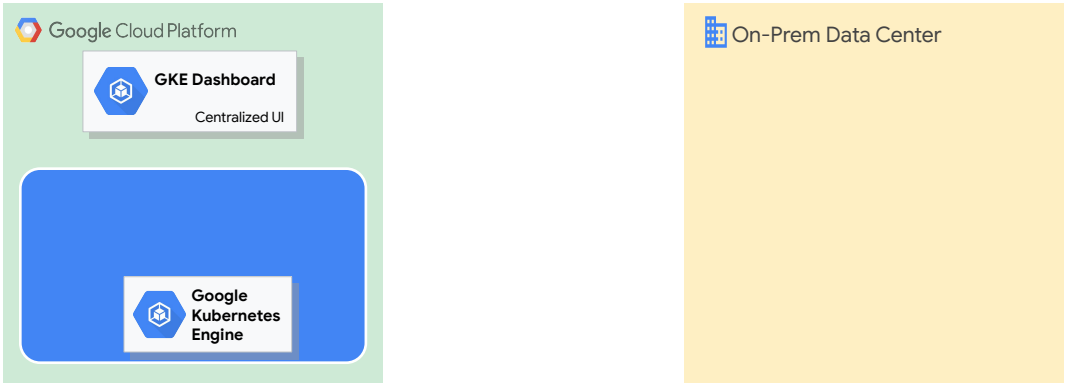
But, what is Anthos?

Anthos is a hybrid and multi-cloud solution powered by the latest innovations in distributed systems and service management software from Google. The Anthos framework rests on Kubernetes and GKE On-Prem, which provides the foundation for an architecture that is fully integrated, with centralized management through a central control plane that supports policy-based application lifecycle delivery across hybrid and multiple cloud environments. Anthos also provides a rich set of tools for monitoring and maintaining the consistency of your applications across all of your network, whether on-premises, in the cloud, or in multiple clouds.

Building a modern hybrid infrastructure, step by step

Let’s take a deeper look at this framework, as we build a modern hybrid infrastructure stack, step by step, with Anthos.

Google Kubernetes Engine for production ready apps

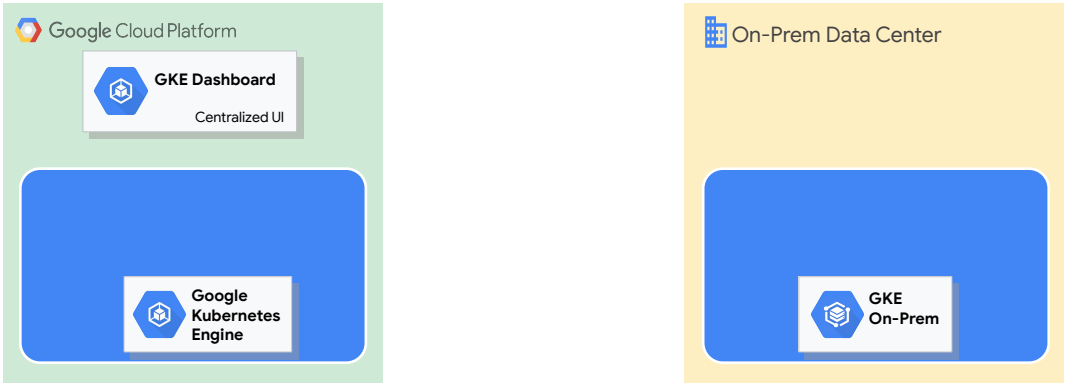


First, let's look at Google Kubernetes Engine on the Cloud side of our hybrid network.

Google Kubernetes Engine:

- Is a managed, production-ready environment for deploying containerized applications.
- Operates seamlessly with high availability and an SLA.
- Runs Certified Kubernetes, ensuring portability across clouds and on-premises.
- Includes auto node repair, auto upgrade, auto scaling.
- Uses regional clusters for high availability with multiple masters, node storage replication across multiple zones (as of Oct. 2019 the number of zones is 3.)

GKE On-Prem is turn-key production-grade Kubernetes

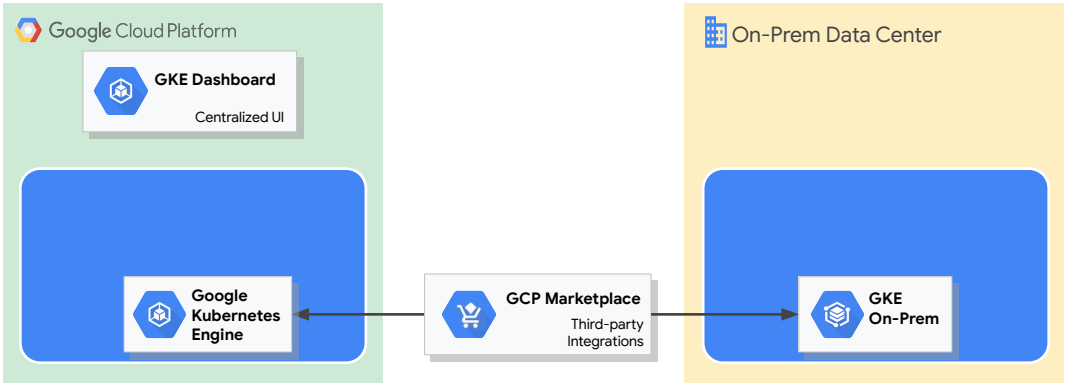


Its counterpart on the on-premises side of our hybrid network is GKE On-Prem.

GKE On-Prem:

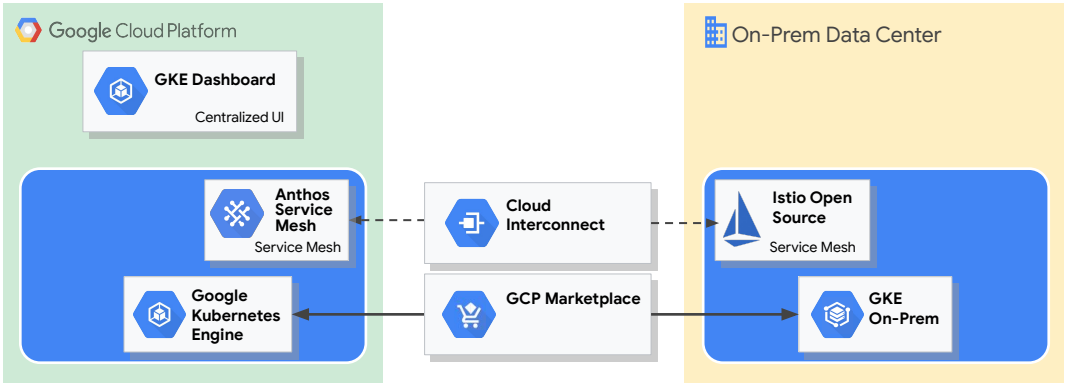
- Is a turn-key, production-grade, conformant version of Kubernetes with a best-practice configuration already pre-loaded.
- Provides an easy upgrade path to the latest Kubernetes releases that have been validated and tested by Google.
- Provides access to Container services on GCP such as Cloud Build, Container Registry, Audit Logging, and more.
- Integrates with Istio, Knative, and Marketplace Solutions.
- Ensures a consistent Kubernetes version and experience across Cloud and on-premises environments.

Marketplace applications are available to all clusters



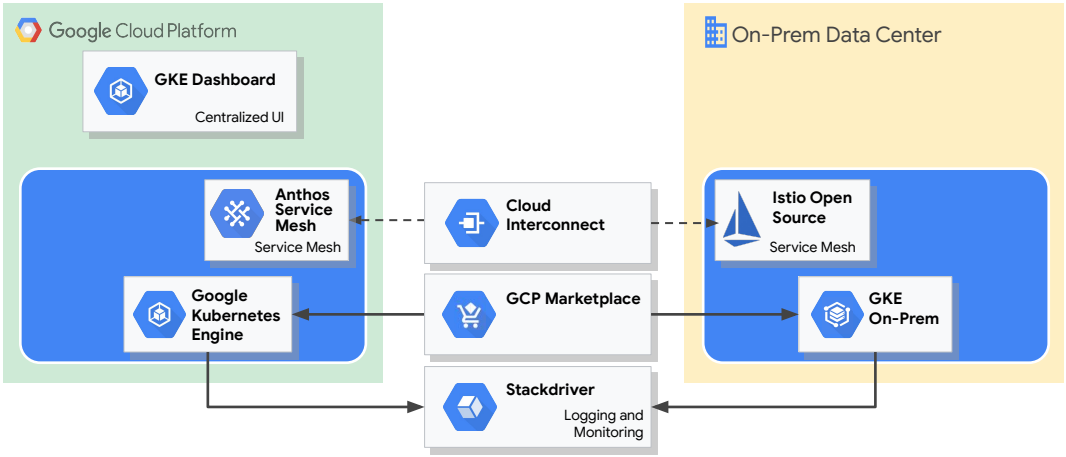
As mentioned, both Google Kubernetes Engine and GKE On-Prem integrate with Marketplace, so that all of the clusters in your network, whether on-premises or in the Cloud, have access to the same repository of containerized applications. This allows you to use the same configurations on both sides of the network, reducing time spent developing applications (write once, replicate anywhere) and maintaining conformity between your clusters.

Service Meshes make apps more secure & observable



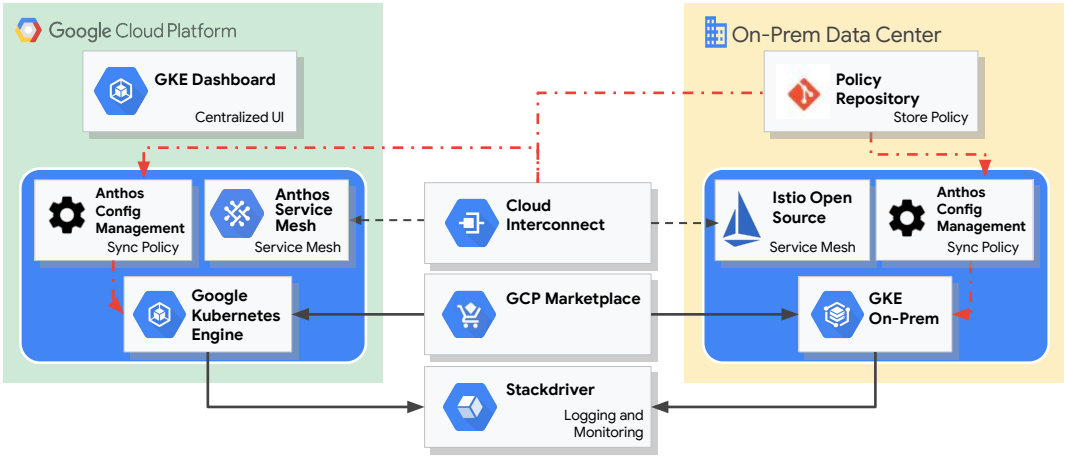
Enterprise applications may use hundreds of microservices to handle computing workloads. Keeping track of all of these services and monitoring their health can quickly become a challenge. Anthos and Istio Open Source service meshes take all of the guesswork out of managing and securing your microservices. These service mesh layers communicate across the hybrid network using Cloud Interconnect, as shown, to sync and pass their data.

Stackdriver Logging and Monitoring watches all sides



Stackdriver is the built-in logging and monitoring solution for GCP. Stackdriver offers a fully managed logging, metrics collection, monitoring, dashboarding, and alerting solution that watches all sides of your hybrid or multi-cloud network. Stackdriver is the ideal solution for customers wanting a single, easy to configure, powerful cloud-based observability solution that also gives you a single pane of glass dashboard to monitor all of your environments.

Configuration Manager is the single source of truth



Lastly, Anthos Configuration Management provides a single source of truth for your clusters configuration. That source of truth is kept in the Policy Repository, which is actually a git repository (in this illustration this repository happened to be located on premises, but it can also be hosted in the cloud.) The Anthos Configuration Management agents use the Policy Repository to enforce configurations locally in each environment, managing the complexity of owning clusters across environments.

Anthos Configuration Management also provides administrators and developers the ability to deploy code changes with a single repository commit, and the option to implement configuration inheritance by using Namespaces.

You can learn more about Anthos from these links

Anthos General Overview:
https://cloud.google.com/anthos/

Anthos Technical Documentation:
https://cloud.google.com/anthos/docs/

Google Cloud

If you would like to learn more about Anthos, here are some more resources to get you started!

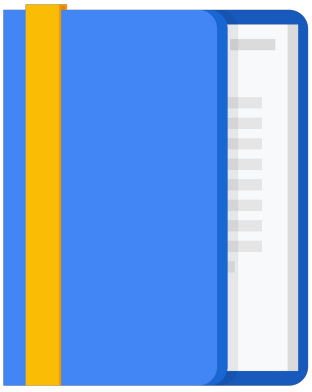
Agenda

Introduction to Containers

Kubernetes and Kubernetes Engine

Hybrid and Multi Cloud

Quiz and Lab



Google Cloud

Question #1

Name two reasons for deploying applications using containers.

Question #1

Name two reasons for deploying applications using containers.

1. Portability across development, testing, and production environments
2. Simpler to migrate workloads
3. Loose coupling
4. Agility

Question #2

True or False: Kubernetes lets you manage container clusters in multiple cloud providers.

Question #2

True or False: Kubernetes lets you manage container clusters in multiple cloud providers.

True: But Kubernetes Engine only runs in GCP

Question #3

True or False: GCP provides a private, high-speed container image storage service for use with Kubernetes Engine.

Question #3

True or False: GCP provides a private, high-speed container image storage service for use with Kubernetes Engine.

True: It's called Google Container Registry

Lab

In this lab you will create a Kubernetes Engine cluster containing several containers, each containing a web server. You'll place a load balancer in front of the cluster and view its contents.

Lab Objectives

Provision a Kubernetes cluster using Kubernetes Engine

Deploy and manage Docker containers using `kubectl`

Google Cloud

More resources

Kubernetes Engine https://cloud.google.com/kubernetes-engine/docs/

Kubernetes http://kubernetes.io/

Google Cloud Build https://cloud.google.com/cloud-build/docs/

Google Container Registry https://cloud.google.com/container-registry/docs/