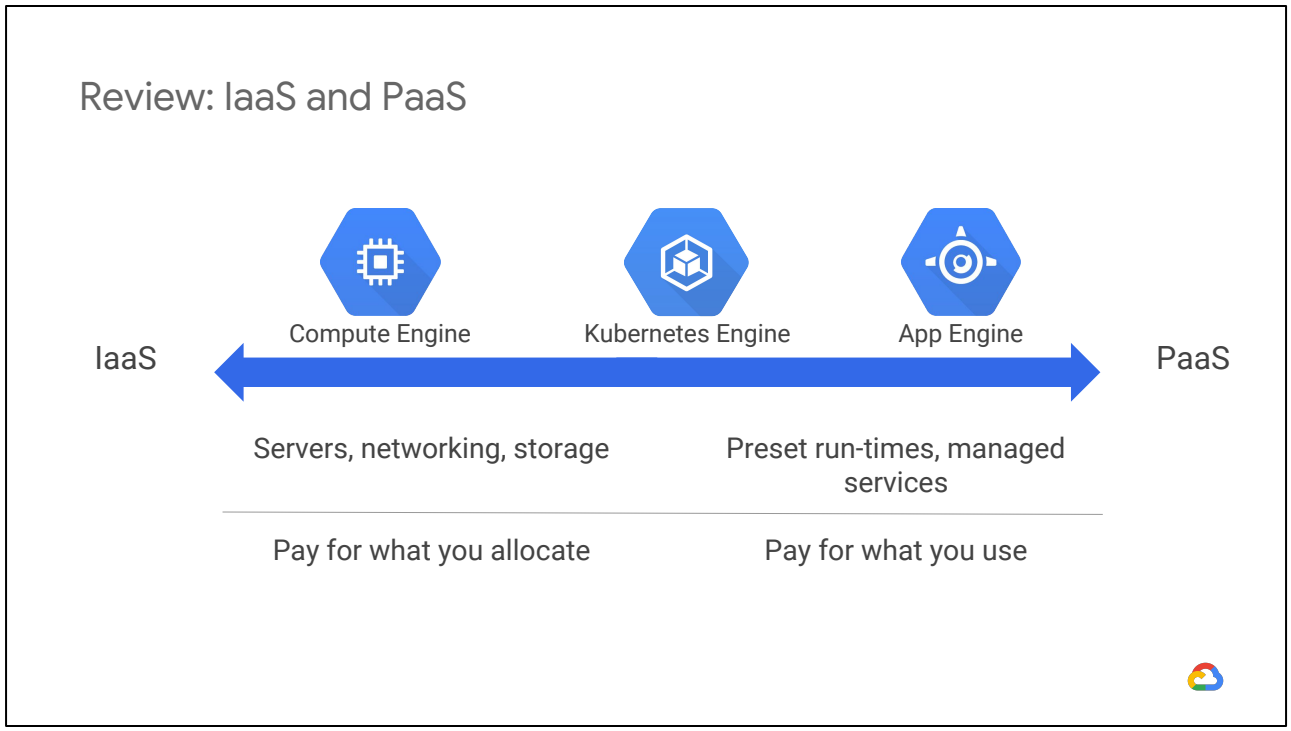
Google Cloud

GCP Fundamentals: Core Infrastructure

Containers in the Cloud

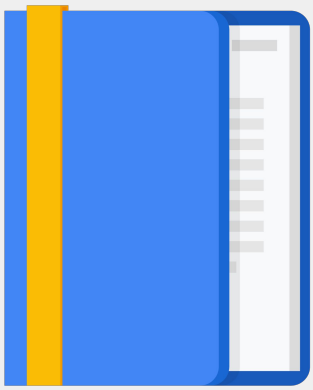


We've already discussed the spectrum between Infrastructure as a Service and Platform as a Service. And We've already discussed Compute Engine, which is GCP's Infrastructure as a Service offering, with access to servers, file systems, and networking.

Now we'll introduce you to containers and Kubernetes Engine which is a hybrid which conceptually sits between the two, offering the managed infrastructure of an IaaS offering with the developer orientation of a PaaS offering.

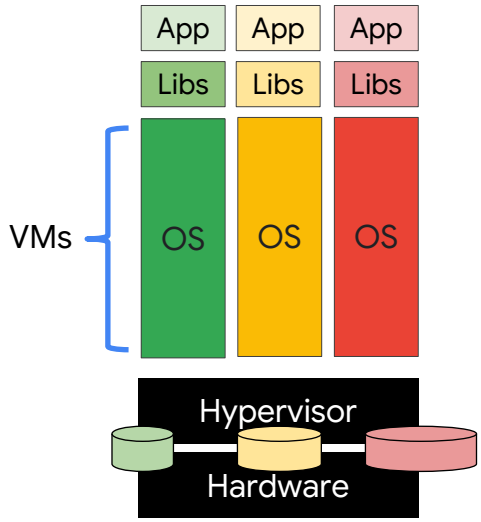
Another choice for organizing your compute is using containers rather than virtual machines, and using Kubernetes Engine to manage them. Containers are simple and interoperable, and they enable seamless, fine-grained scaling. Google launches billions of containers every week.

Agenda



- **Introduction to Containers**
- Kubernetes and Kubernetes Engine
- Quiz and Lab

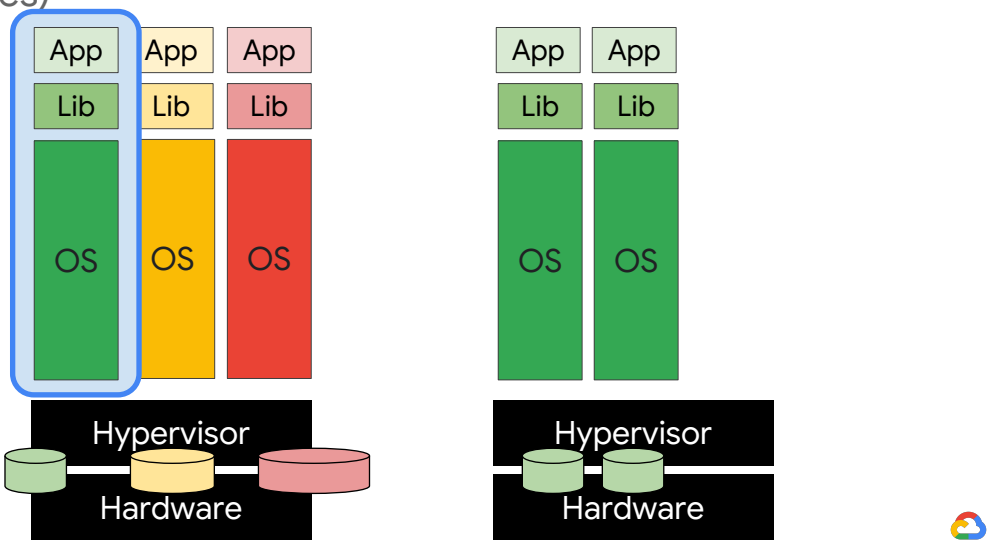
IaaS allows you to share resources by virtualizing the hardware



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 their own runtimes and libraries as well as their own partitions of RAM, file systems, networking interfaces, and so on.

You have your tools of choice on your own 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)

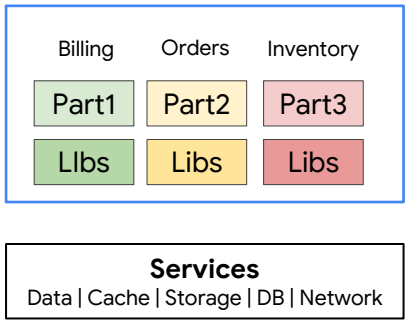


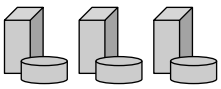
But flexibility comes with a cost. The smallest unit of compute is an app with its VM. The guest OS may be large, even gigabytes in size, and takes minutes to boot.

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.

PaaS provides hosted services and an environment that can scale workloads independently

- All you do is write code in self-contained workloads and include any libraries
- It's easy to decouple code because you're not tied to the underlying hardware, OS, or a lot of the software stacks you used to manage





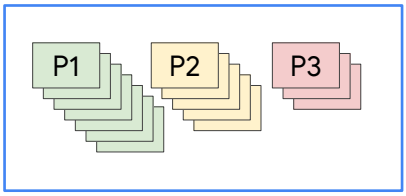
With Platform as a Service you get access to hosted programming services.

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

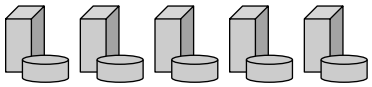
Workloads do not need to represent entire applications. They're easier to decouple because they're not tied to the underlying hardware, OS, or a lot of the software stacks that you used to manage.

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

- This encourages you to build your apps as decoupled microservices
- But you may not be able to fine-tune the underlying architecture to save cost



Services
Data | Cache | Storage | DB | Network



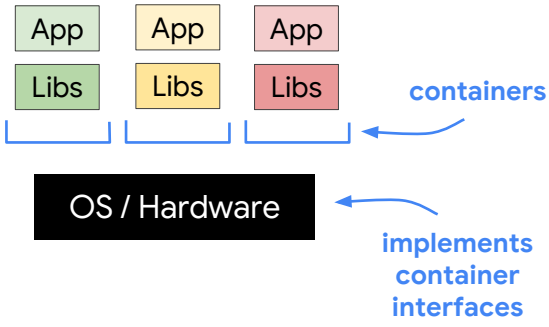
As demand for your app increases, the platform scales your app seamlessly and independently by workload and infrastructure.

This scales rapidly and encourages you to build your applications as decoupled microservices that run more efficiently, 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)



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.

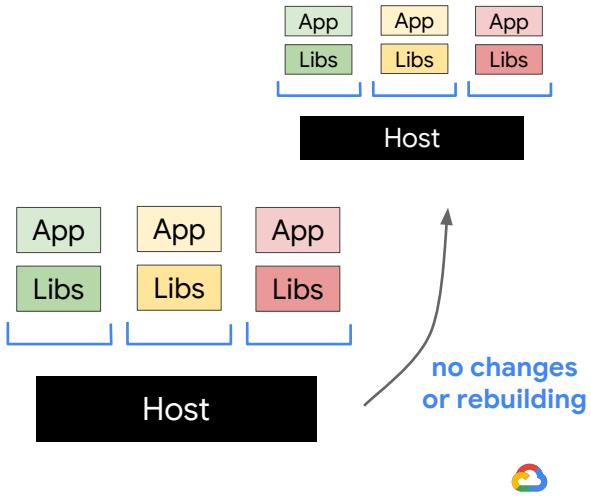
All you need on each host is an OS kernel that supports containers and a container runtime.

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
- Bundle any dependencies
- 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



Because a container can configure its own isolated partition of file system, RAM, and networking, and specify its own dependencies on top of that, it's ultra portable.

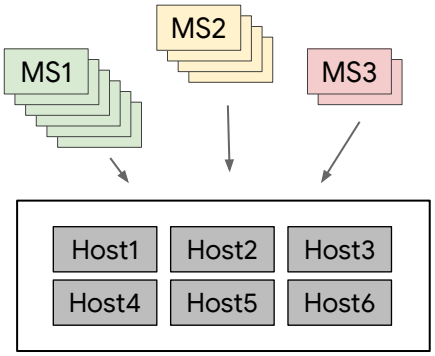
You can effectively treat the OS and hardware as a black box.

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 hundreds or thousands of 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



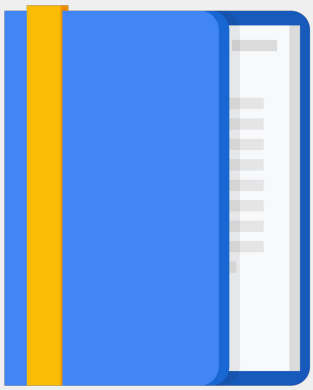
If you want to scale, for example, a web server, you can do so in seconds and deploy dozens or hundreds of them (depending on the size or your workload) on a single host or group of hosts.

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**
- Quiz and Lab

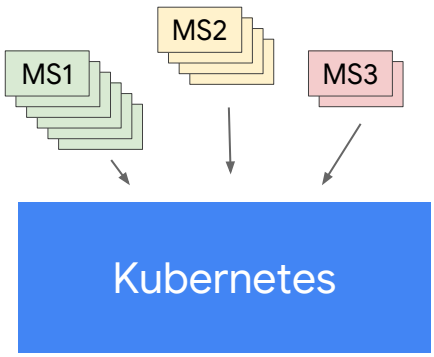
Kubernetes makes it easy to orchestrate many containers on many hosts

It's an open-source container management platform

- It was built by Google to run applications at scale
- Google starts billions of containers with it every week

Kubernetes lets you:

- Install the system on local servers in the cloud
- Manage container networking and data storage
- Deploy rollouts and rollbacks
- Monitor and manage container and host health



A tool that helps you do this well is Kubernetes.

You can install Kubernetes on a group of your own managed servers or run it as a hosted service in GCP on a cluster of managed GCE instances called Kubernetes Engine.

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

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 any 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

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')
```

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

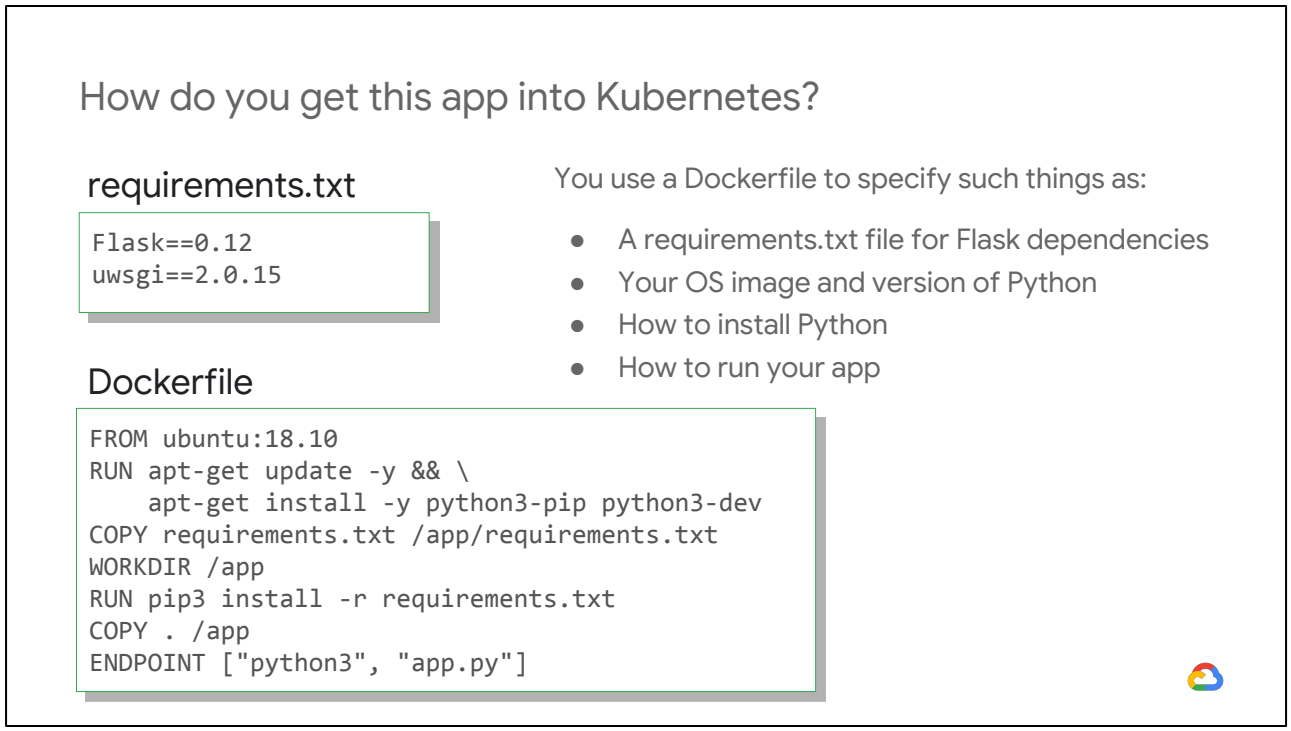
I'll use an open-source tool called **Docker** that defines a format for bundling your application, its dependencies, and machine-specific settings into a container; you could use a different tool like **Google Cloud Build**. It's up to you.

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
```

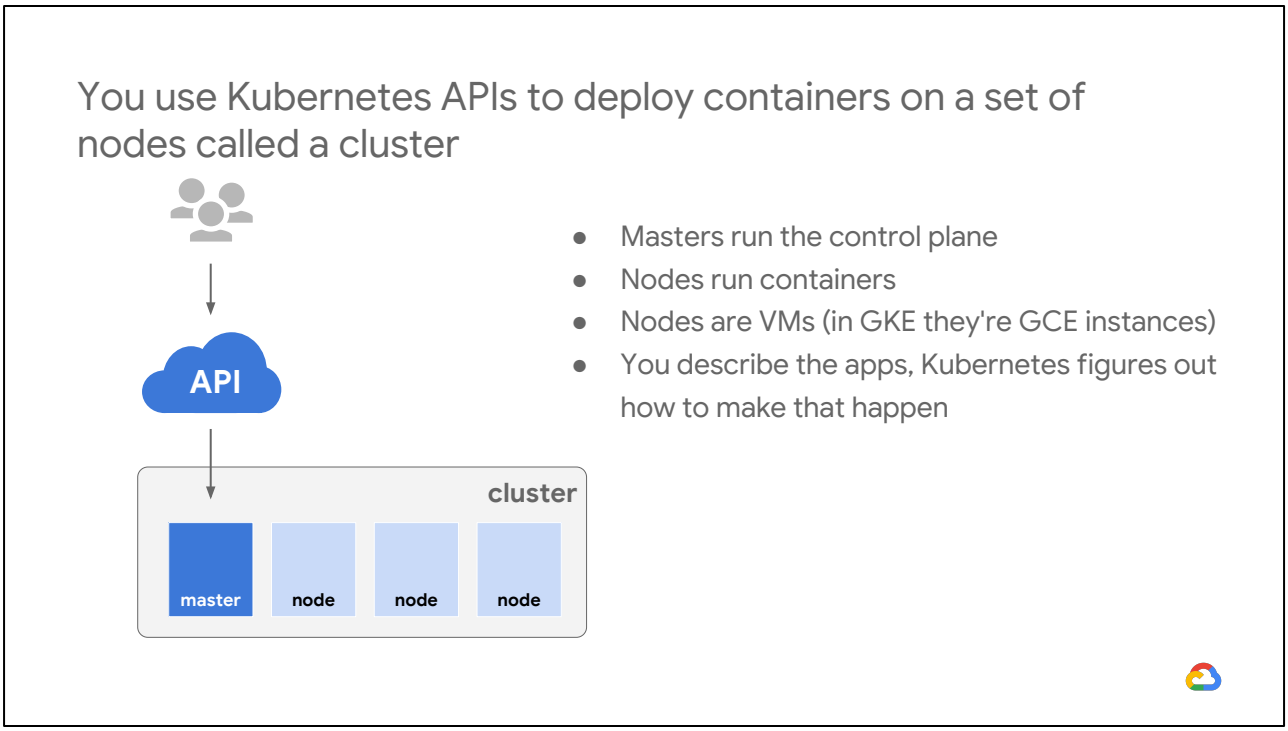
- **`docker build`** builds a container and stores it locally as a runnable image
- You can upload images to a registry service (like **Google Container Registry**) for sharing
- **`docker run`** starts the container image

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.



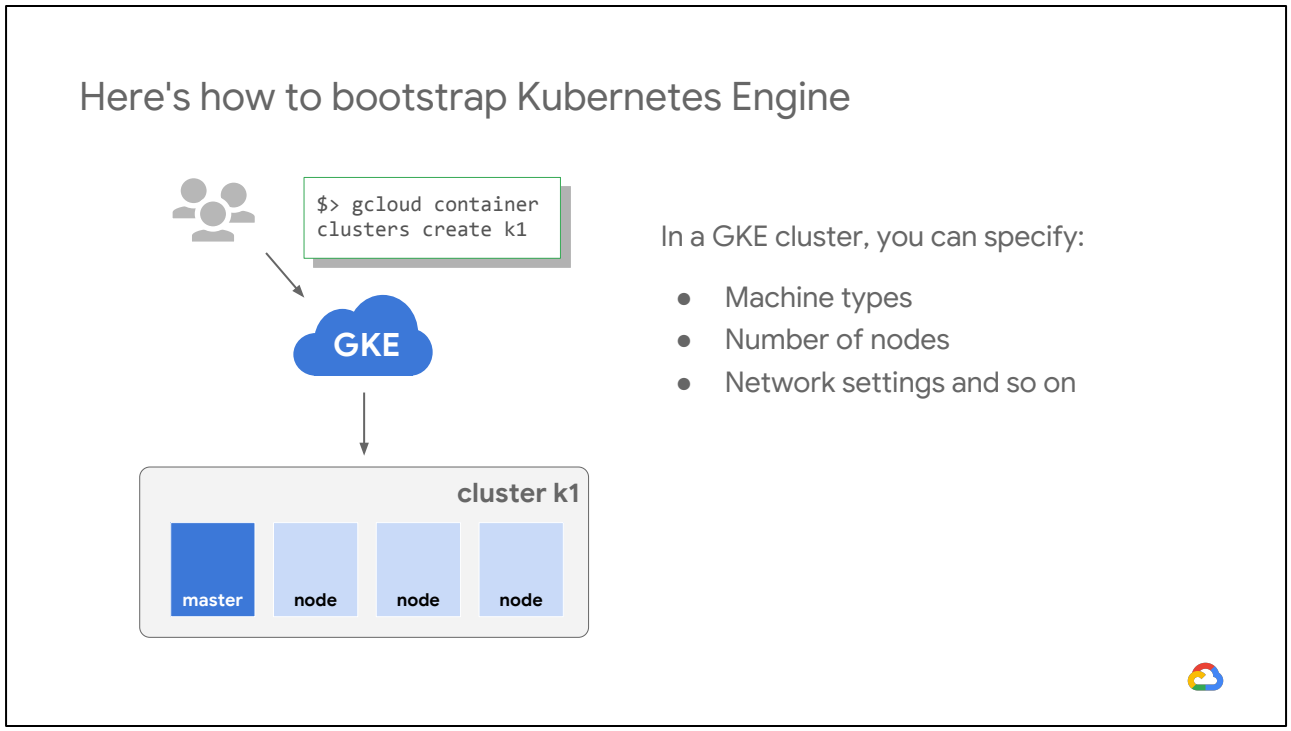
Now, I'll show you where **Kubernetes** comes in.

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).

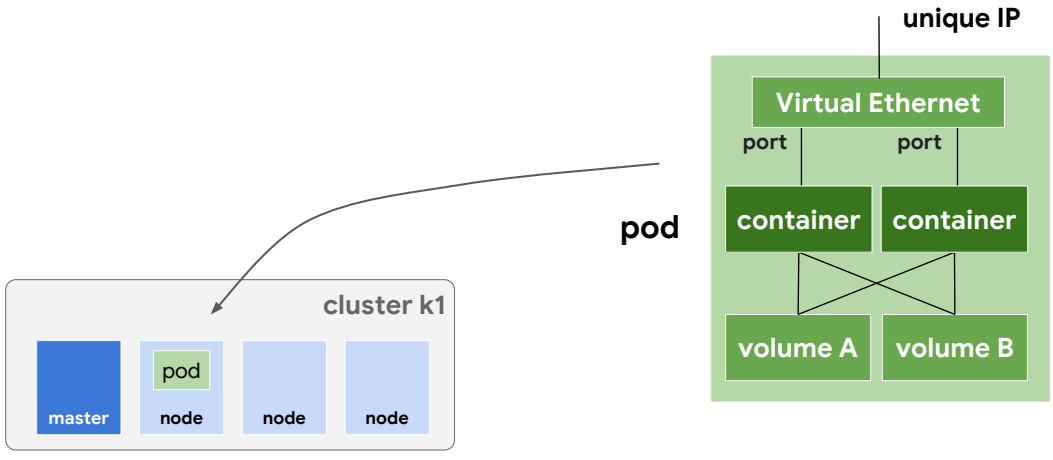
GKE is a hosted Kubernetes by Google. GKE clusters can be customized and they support different machine types, number of nodes, and network settings.

To start up Kubernetes on a **cluster** in GKE, all you do is run this command:

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

You can check its status in admin console.

When you deploy containers on nodes you use a wrapper called a **Pod**



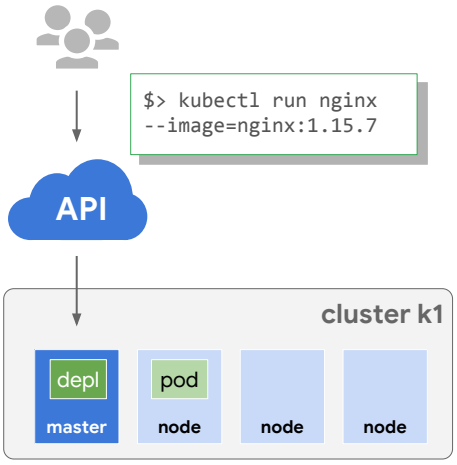
Then you deploy **containers** on nodes using a wrapper around one or more containers called a Pod.

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`

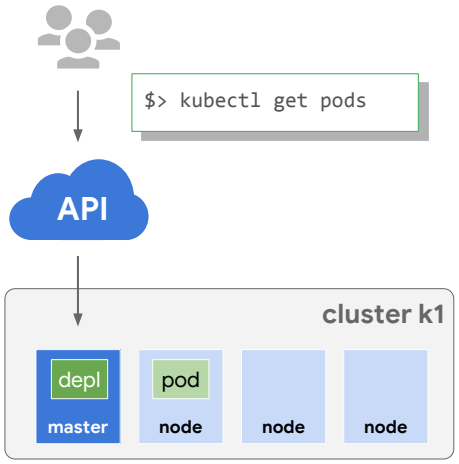


- **`kubectl`** is the command-line client to the Kubernetes API
- This command starts a **Deployment** with a container running in a Pod
- In this case, the container is an image of the NGINX server

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

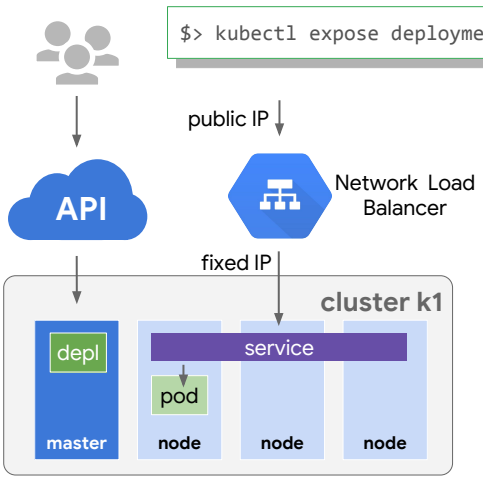


A **Deployment** manages a set of replica Pods representing an app or workload and makes sure the number of desired is running and healthy. It keeps your Pods running even when nodes they run on fail. It could represent a component of an application or an entire app. In this case, it's the nginx web 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"

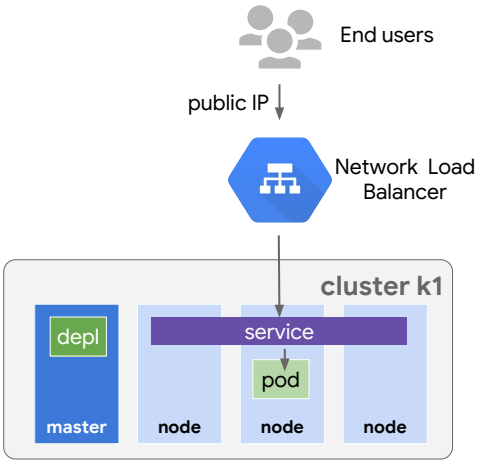
By default, Pods in a Deployment are only accessible by default in the cluster and have ephemeral IP addresses.

To make them publicly available with a fixed IP, 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



- 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.

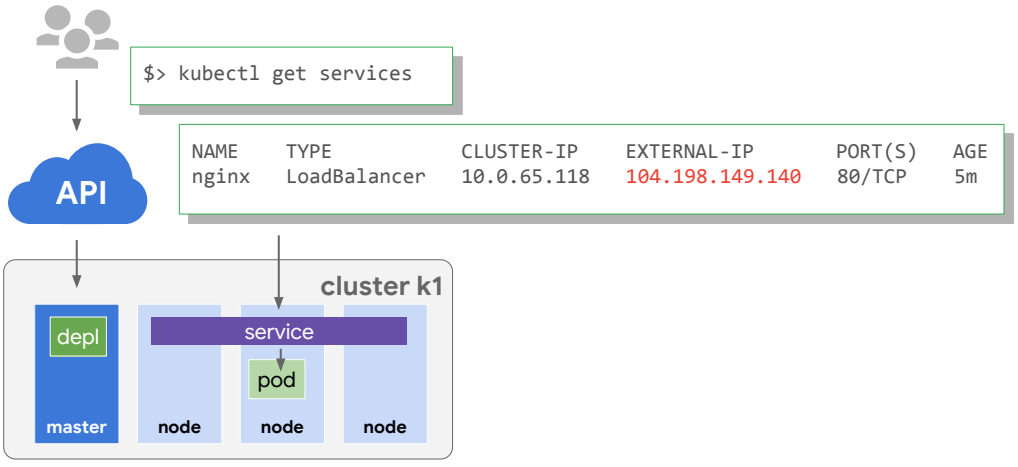
A **Service** is an abstraction which defines a logical set of Pods and a policy by which to access them.

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.

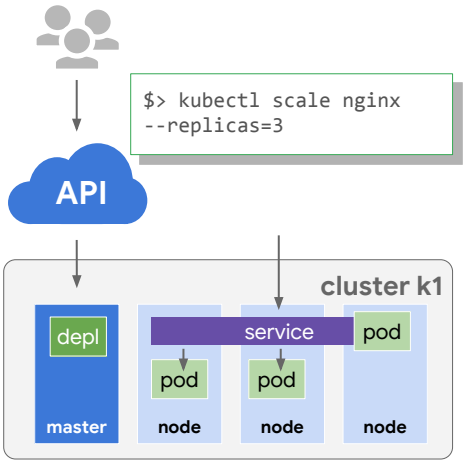
For example, if you create two sets of Pods called frontend and backend, and put them behind their own Services, backend Pods may change, but frontend Pods are not aware of this. They simply refer to the backend Service.

To get the Service's public IP run **`kubectl get services`**



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

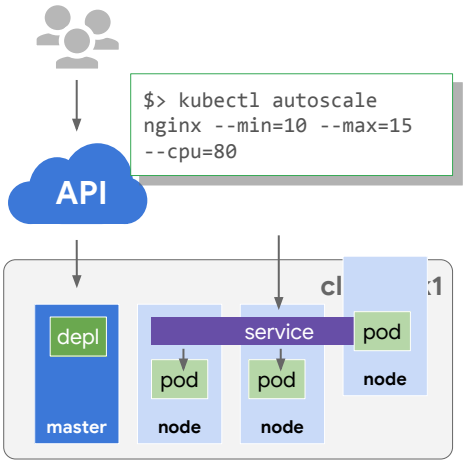
To scale a Deployment, run `kubectl scale`



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.

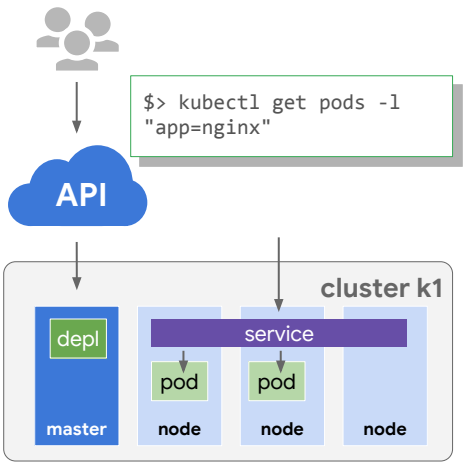
You can also run autoscaling with all kinds of parameters or put it behind programming logic for intelligent management



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.

The real strength of Kubernetes comes when you work in a declarative way (here's how to get a config file)




```
apiVersion: v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15.7
        ports:
        - containerPort: 80
```

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.

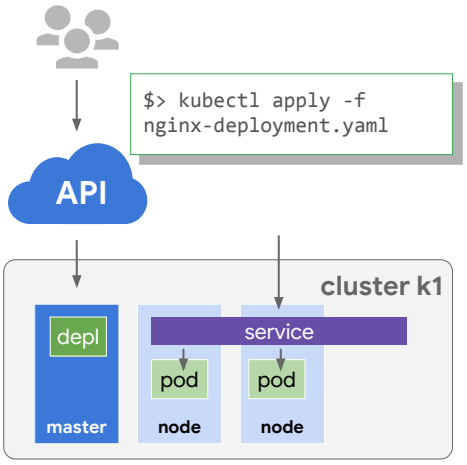
But the real strength of Kubernetes comes when you work in a **declarative** way.

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.

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

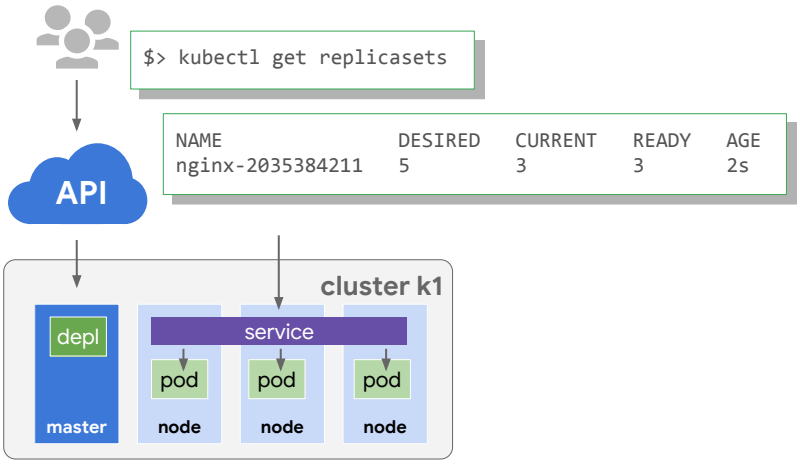



```
apiVersion: v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 5
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.10.0
        ports: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

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

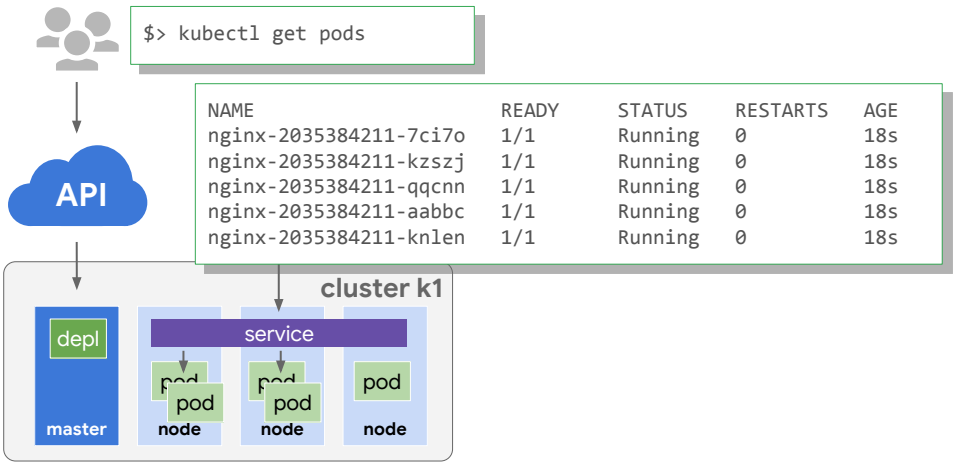
And run the **kubectl apply** command to use the config file.

Run `kubectl get replicasets` to see the updated state



Now look at your replicas to see their updated state.

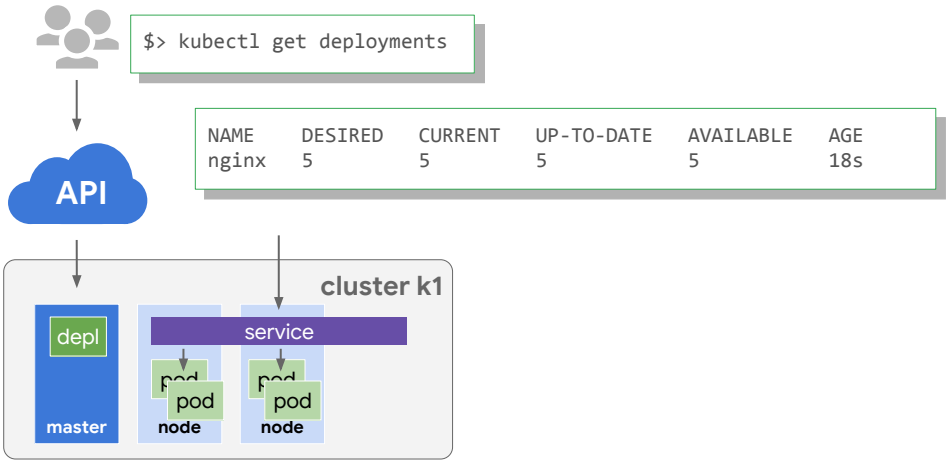
Run **kubectl get pods** to watch the Pods come on line



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

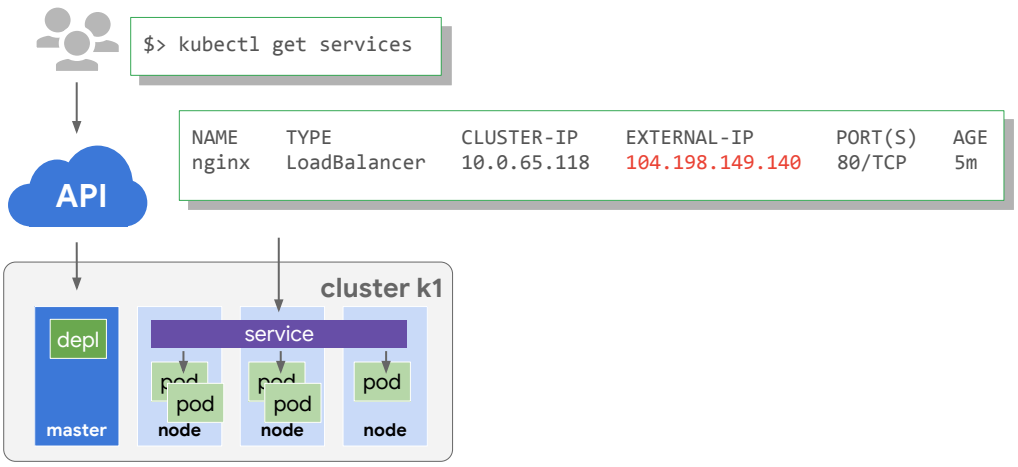
In this case, all five are **READY** and **RUNNING**.

Run **`kubectl get deployments`** or describe deployments to ensure the proper number of replicas are 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**.

Then you build the container and run its image



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
  # ...
```

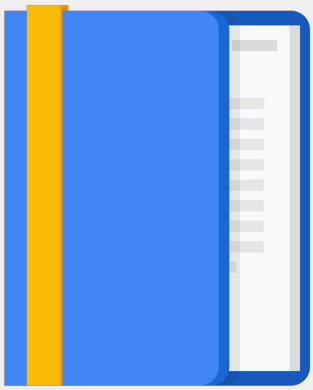
The last question is what happens when you want to update a new version of your app?

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
- **Quiz and Lab**

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.

Objectives

- Provision a Kubernetes cluster using Kubernetes Engine
- Deploy and manage Docker containers using `kubectl`

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/