Introduction to Containers and Docker

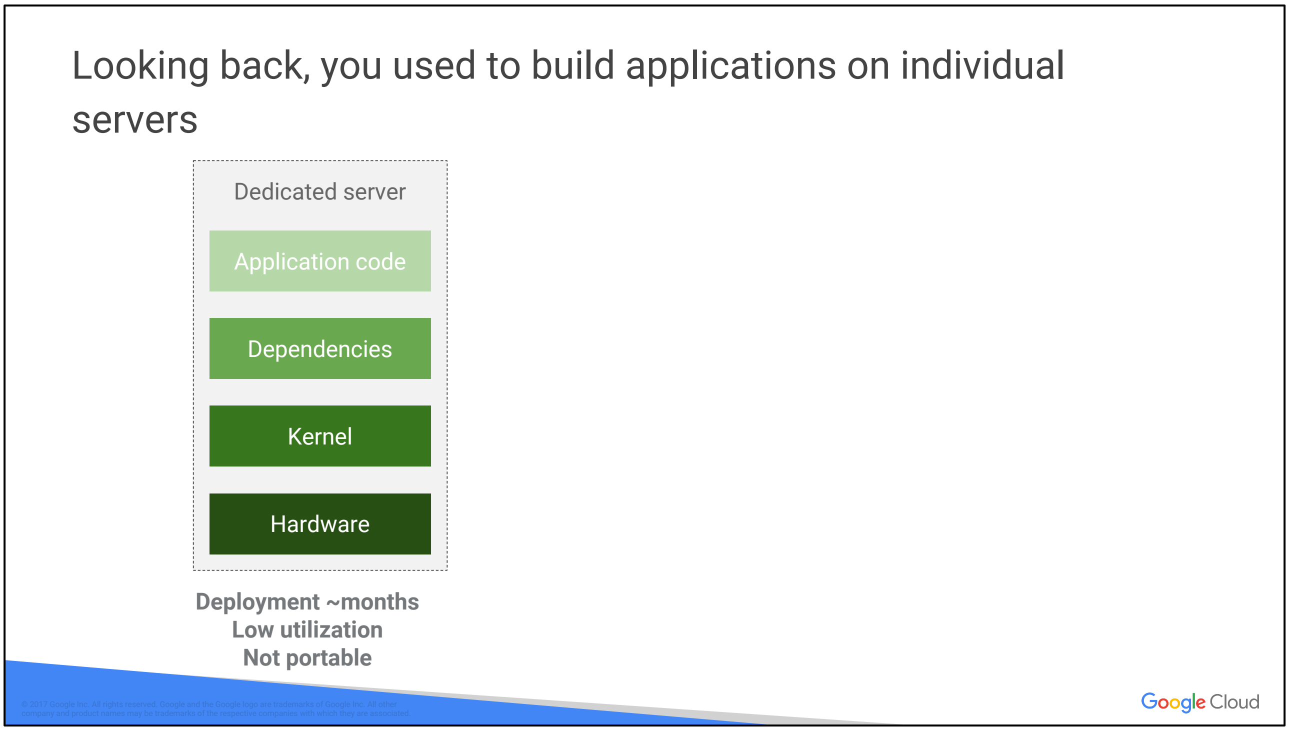
Part 1: Containers

Version 1.6

Google Cloud

[Duration: 20 mins]

Containers are taking the world by storm. Customers have shared with us how it has revolutionized and changed the way they deploy and run applications.



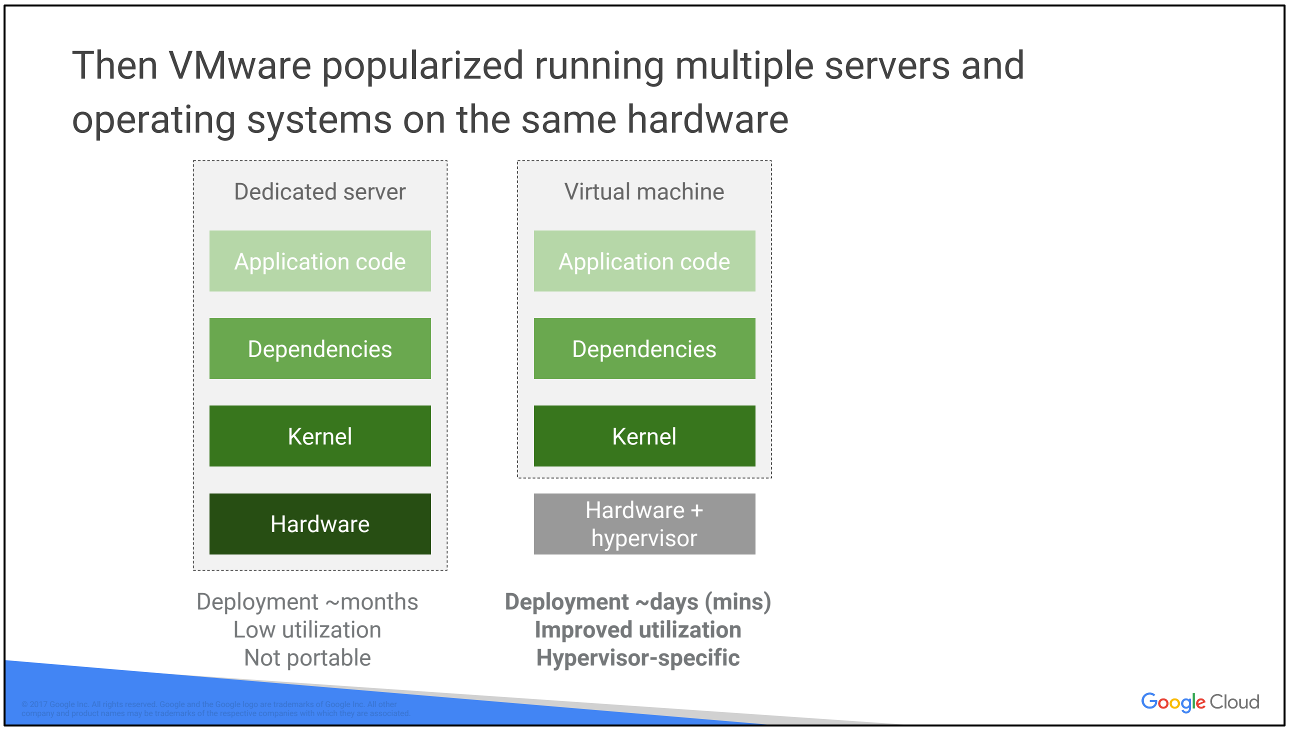
Looking back, you used to build applications on individual servers

Dedicated server
Application code
Dependencies
Kernel
Hardware

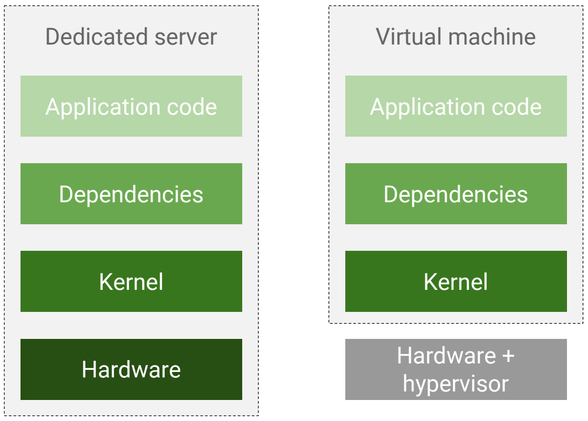
Deployment ~months
Low utilization
Not portable

In this section, you'll walk through what containers are, why they are a small and portable and a good level of abstraction, and how to build and run Docker containers.

In the past, do you remember when you got a new server? You would install an OS on it, and then one by one install the various applications that you needed, such as a web server, maybe a database, Java runtime, maybe some web package software, etc. In many ways, it was similar to what you did on your personal computer (whether Windows or Mac or whatever). If you wanted to get fancy and add another web server or database later on for scaling, you bought another server. Each server had a purpose. The problem is that this wasted a lot of resources, took a lot of time to deploy, maintain, and scale, and was not very portable. In addition, many times applications were built assuming a specific OS and even hardware.



Then VMware popularized running multiple servers and operating systems on the same hardware



Deployment ~months
Low utilization
Not portable

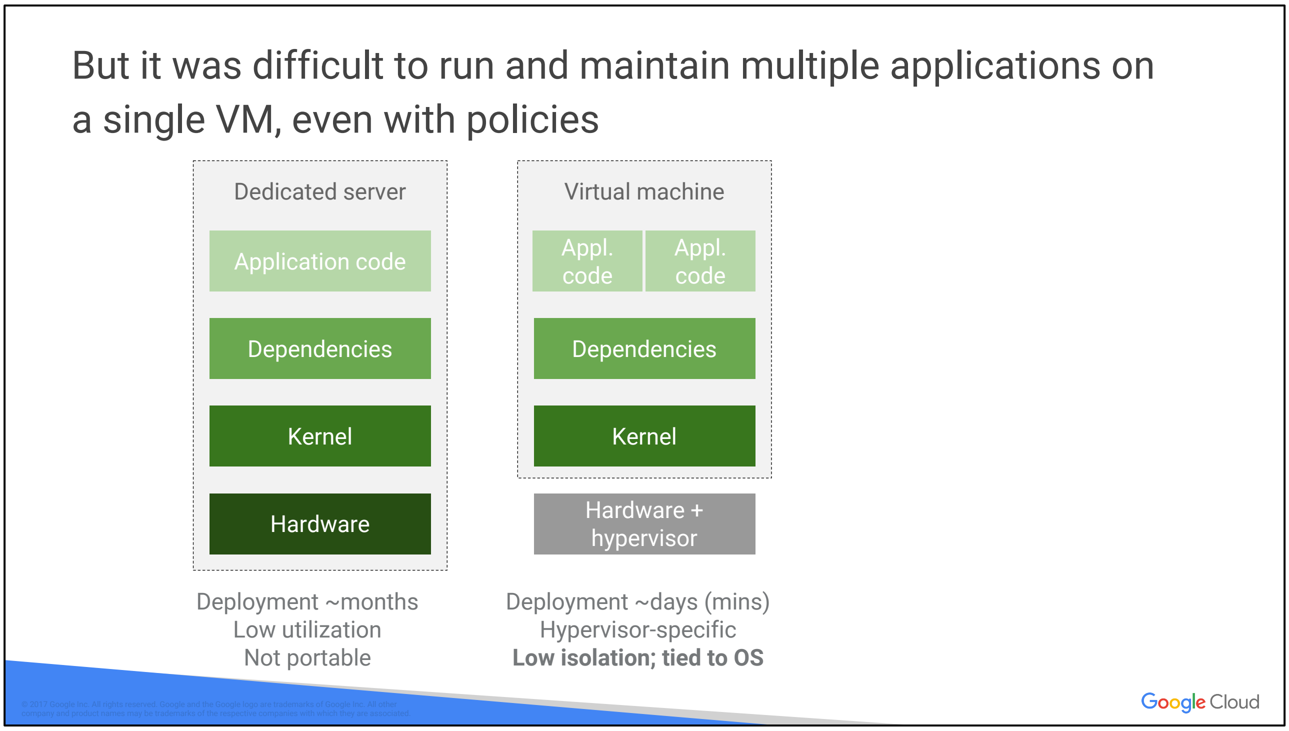
Deployment ~days (mins)
Improved utilization
Hypervisor-specific

Google Cloud

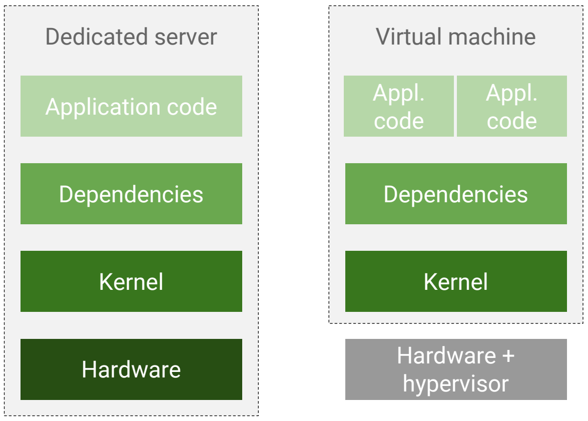
VMware popularized the idea that you could run multiple servers and operating systems on the same physical hardware using a hypervisor. Now you create VM images and deploy them fairly quickly. This improved hardware utilization deployment time and portability. But with a VM, you are still bundling the application, dependencies, and OS together.

This isn't fully portable either because there are many types of hypervisors, so you can't move your VM to an environment running a different hypervisor, such as the cloud or your laptop. And every time you start a VM, the OS has to boot, which takes time.

But this was a great step forward from dedicated servers since it raised the abstraction layer and the VMs allowed you to abstract above the hardware and hypervisor.



But it was difficult to run and maintain multiple applications on a single VM, even with policies

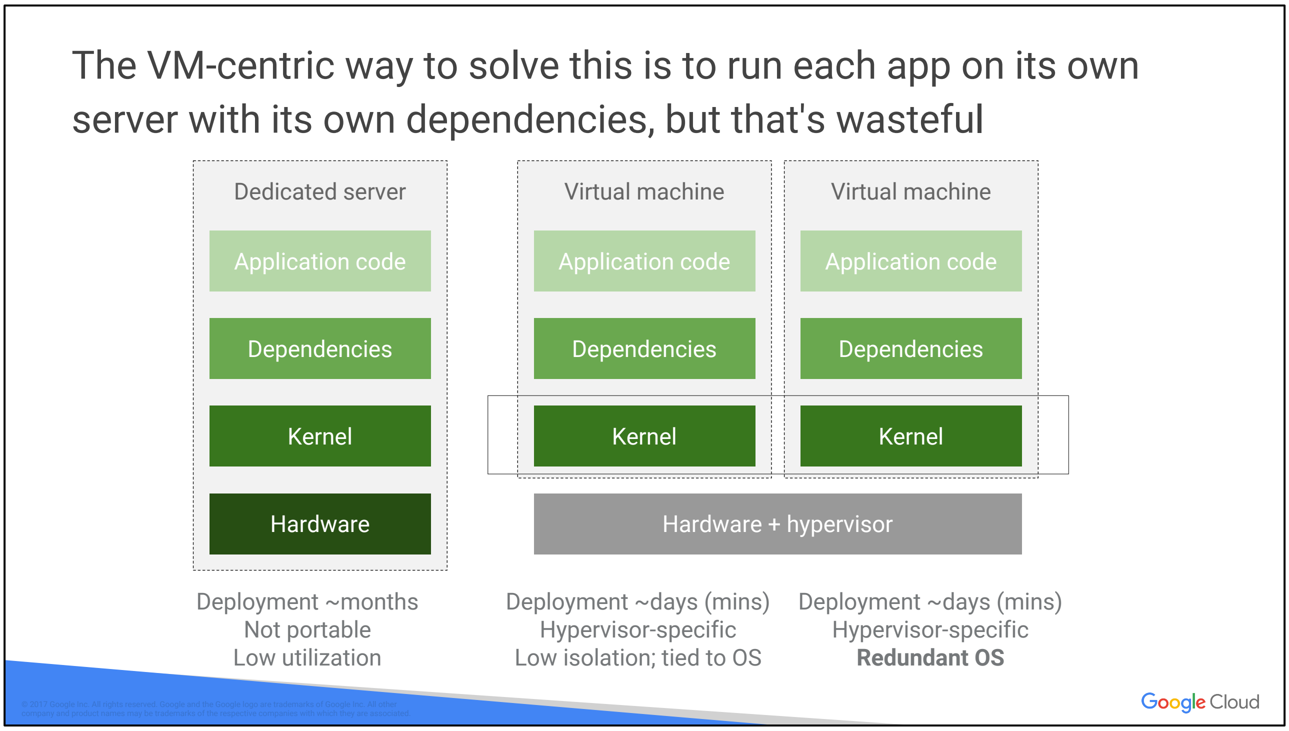


Deployment ~months
Low utilization
Not portable

Deployment ~days (mins)
Hypervisor-specific
Low isolation; tied to OS

But companies quickly ran into this problem. It became clear that it was rather difficult to run multiple applications within a single VM. If there are 2 different applications sharing a common set of dependencies, how can the 2 applications continue with future development without causing potential conflicts? What if one application needs to make a dependency change (upgrade, etc.)? How do you know that the other application will still work? Even apps that share dependencies might not run well because they aren't isolated from each other if they are in the same VM. And of course, one application might get really popular, which slows the VM and affects performance of the other applications running on the VM.

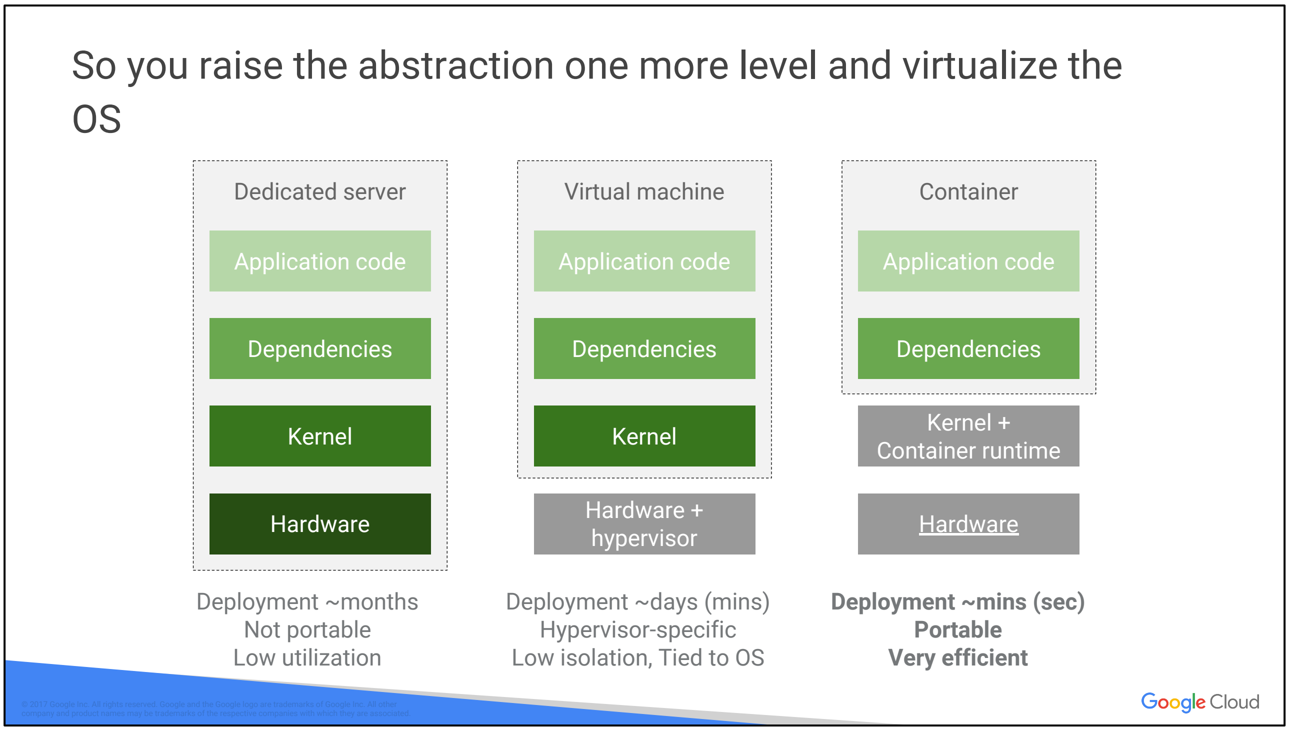
You can try to solve this problem by instituting policies for software engineering. For example, you can lock down the dependencies and say that no application is allowed to make changes, but that doesn't work because dependencies need to be upgraded from time to time. You can also add a lot of integration tests to ensure that applications work ,but this is error prone and time consuming. And in the end, developers aren't happy with either of these solutions because they slow down development.



The VM-centric way to solve this is to run each app on its own server with its own dependencies, but that's wasteful

Dedicated server	Virtual machine	Virtual machine
Deployment ~months	Deployment ~days (mins)	Deployment ~days (mins)
Not portable	Hypervisor-specific	Hypervisor-specific
Low utilization	Low isolation; tied to OS	**Redundant OS**

So the VM-centric way to solve this problem is running a VM for each application, which has its own set of dependencies. So each application is allowed to maintain its own dependencies without impacting any other application, and there is isolation of hardware so one application won't affect the performance of another. But as you can see, the result is that two complete copies of the kernel are running. This becomes much worse if you imagine hundreds or thousands of applications. Imagine trying to do a kernel update across all of your applications. So this is redundant and wasteful.



So you raise the abstraction one more level and virtualize the OS

Dedicated server	Virtual machine	Container
Application code	Application code	Application code
Dependencies	Dependencies	Dependencies
Kernel	Kernel	Kernel + Container runtime
Hardware	Hardware + hypervisor	Hardware

Deployment ~months
Not portable
Low utilization

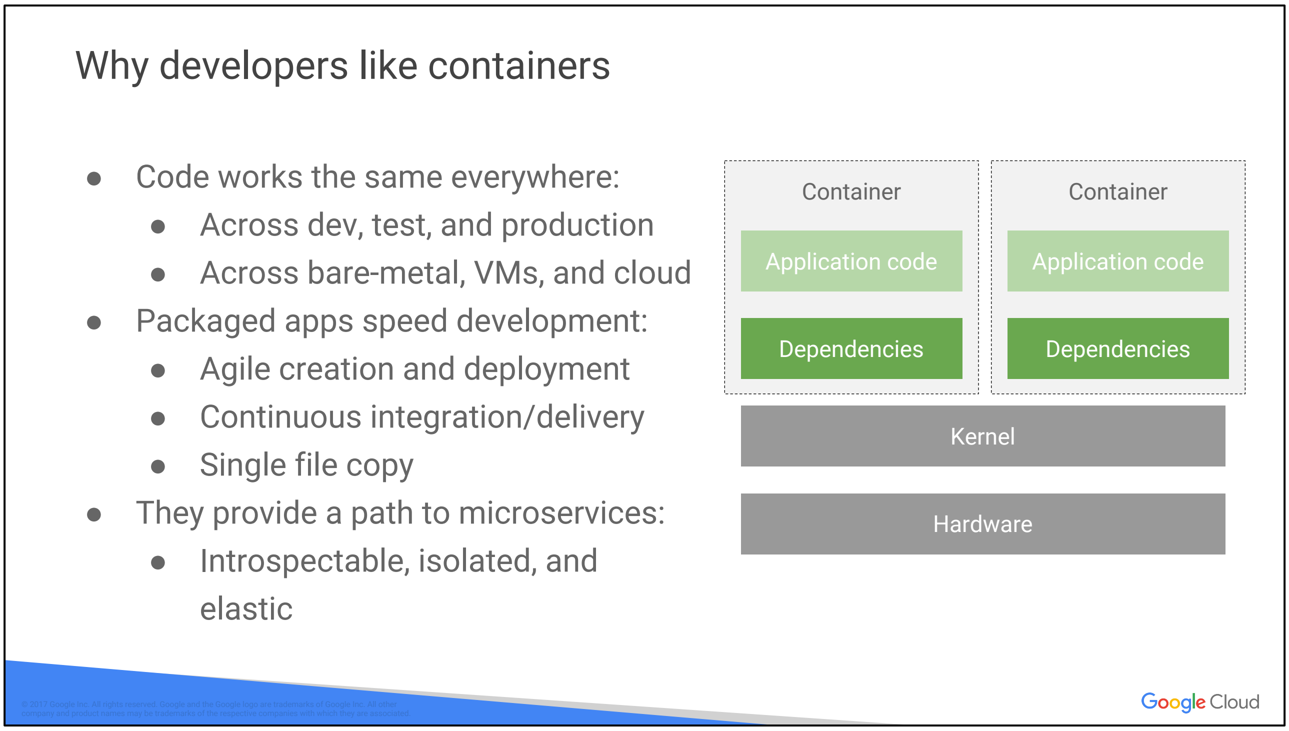
Deployment ~days (mins)
Hypervisor-specific
Low isolation, Tied to OS

Deployment ~mins (sec)
Portable
Very efficient

Google Cloud

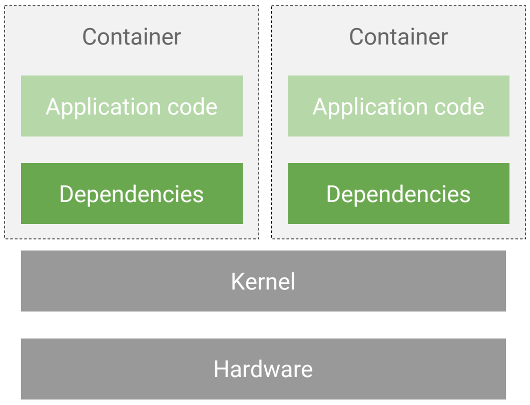
So you raise the abstraction level one more time—this time by virtualizing the OS. You package only your application and dependencies together. This is called a container. You can run this anywhere that runs the Linux kernel—it is ultra portable so you can go from your laptop to the cloud without needing to change or reboot anything. It starts very fast so your app scales very fast. Its lightweight because it doesn't carry a full OS, so it can be scheduled/packed tightly onto the underlying hardware, which results in the best utilization. Startup and shutdown is no more than starting and stopping a process running in the OS.

This is the right level of abstraction and is the next step in the evolution of managing code.



Why developers like containers

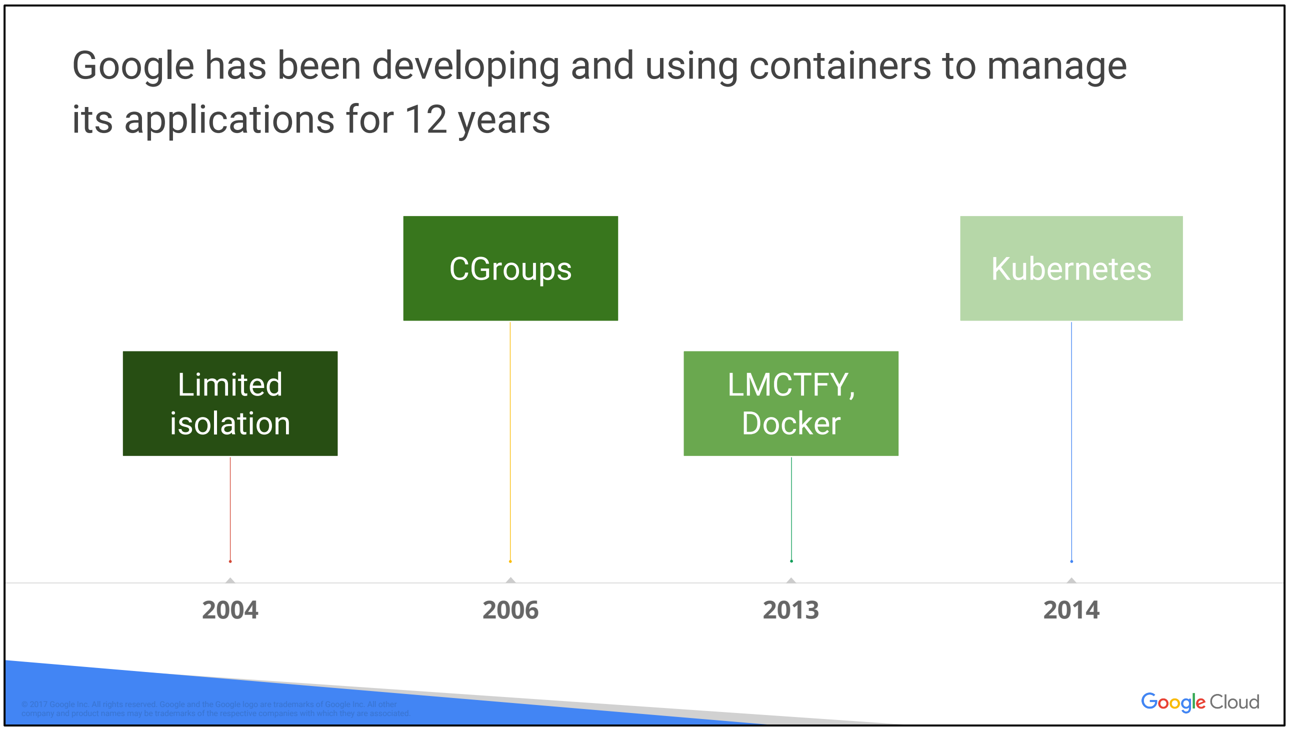
- Code works the same everywhere:
  - Across dev, test, and production
  - Across bare-metal, VMs, and cloud
- Packaged apps speed development:
  - Agile creation and deployment
  - Continuous integration/delivery
  - Single file copy
- They provide a path to microservices:
  - Introspectable, isolated, and elastic



Google Cloud

Developers like containers because:

- Code works the same everywhere with a Linux kernel underneath. You no longer have code that works on your laptop but doesn't work in production; the container is the same and runs anywhere.
- They speed development: You make incremental changes to a container based on a production image, and deploy it very quickly with a single file copy.
- They provide a path to microservices which allows the OS to introspect running work (this makes it easier to debug). And they isolate application code so it's loosely coupled and quick to add or remove containers.



Google has been developing and using containers to manage its applications for 12 years

One of the reasons Google started with containers back in 2004 is that they helped Google get started in providing isolation. But at the time, isolation was limited, so they came up with an important container concept called *cgroups*. Google became successful in part because of how quickly they could serve search results pages. In order to provide search results, Google first builds an index which takes a lot of processing power. In fact, the search index has long-running jobs that use all of the processing power available if they can.

But Google also has jobs that need to run queries. If you put jobs that build the index and run queries on the same server, jobs that build the index will use all of the processing power, and jobs that run queries will not have the processing power they need. So Google had to separate the servers that built the index from those that served results.

Having different servers for different jobs becomes hard to manage. So Google developed a feature that strongly restricts the amount of resources that are available to jobs on a server. That was originally called a *process container* and then renamed a *control group* (or *cgroup*)—one of the bases for modern-day containers. The purpose of a cgroup is to limit the proportion of CPU and RAM capabilities a process can use.

In addition to cgroups, Google also needed a way to allocate portions of disk and network resources which you'll learn about later.

With cgroups, Google could guarantee batch work wouldn't steal all resources from interactive services, so they could combine these two jobs into a single server or sets of servers—called a *cluster*—and they built a cluster management system designed to get the most utilization out of the machines at all times.

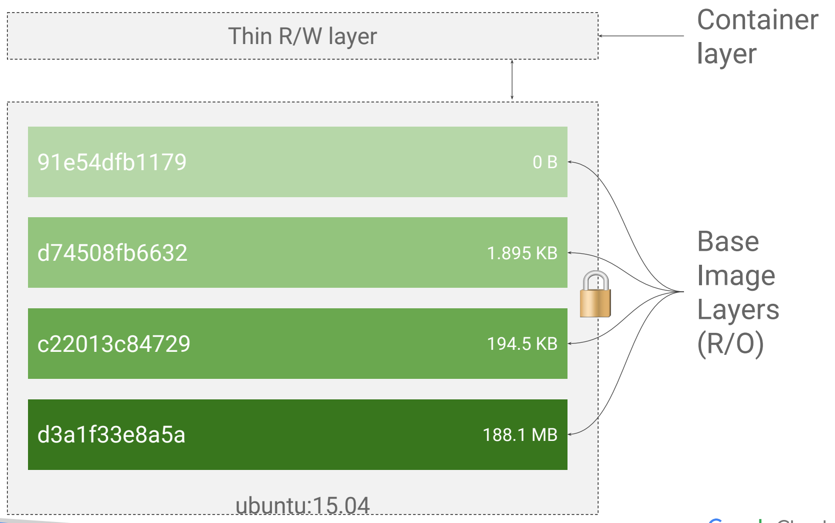
The system that Google built to manage all of this was called *Borg*. With Borg, you declare what you want (for example, 10,000 CPUs running the Gmail front end), and it will find space on the thousands of machines in that cluster to run your job—and more importantly, make sure it keeps running.

Over time, there were other implementations of these ideas. In 2013, a company named Docker open-sourced and popularized the container as a compact image-packaging format for developers. Soon after that, a team of Googlers developed, and released as open source, a container orchestration system for Docker containers. This system was inspired by Borg, and it is called *Kubernetes*. In 2014, they announced and released Kubernetes to the open source community.

Containers use a layered file system with only the top layer writable

Dockerfile

```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```



Google Cloud

Containers use the concept of a layered filesystem.

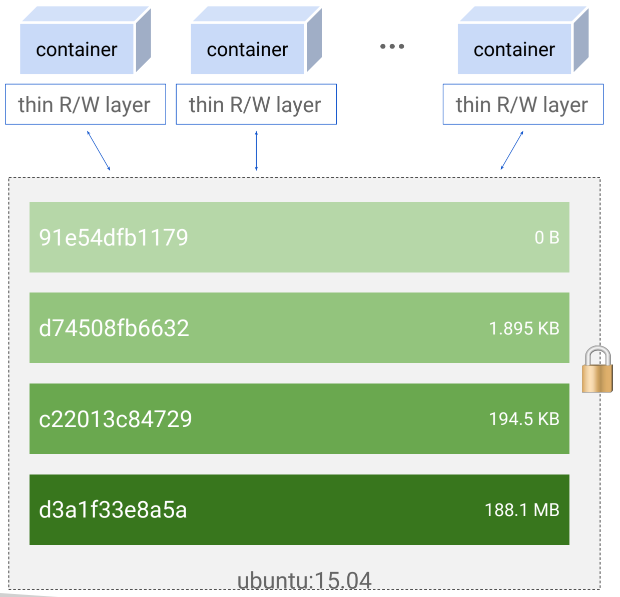
A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the last one is read-only, and you make your updates to a small read-write layer on top of the image in a container. Consider the Dockerfile on the slide:

This Dockerfile contains four commands, each of which creates a layer. The FROM statement starts out by creating a layer from the ubuntu:15.04 image. The COPY command adds some files from your Docker client's current directory. The RUN command builds your application using the make command. Finally, the last layer specifies what command to run within the container.

Each layer is only a set of differences from the layer before it. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the *container* layer. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer.

The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged.

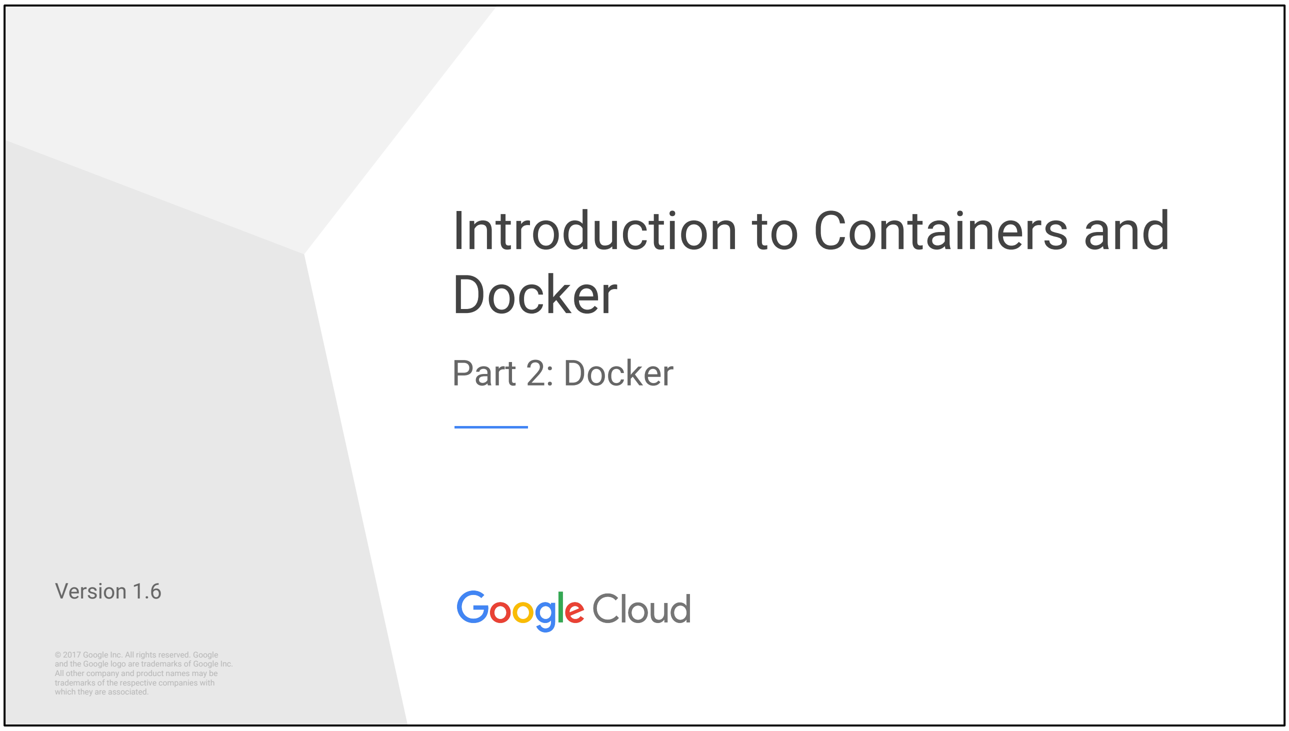
Containers promote smaller shared images



Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram shows multiple containers sharing the same Ubuntu 15.04 image.

Also, each layer is only a set of differences from the layer before it, so sharing promotes smaller images.

As an example, your base application image may be 200 MB, but the diff to the next point release may be only 200 KB. When you build a container, instead of copying the whole image, it creates a layer with just the diff. When you run a container, Docker pulls down whatever layers it needs. When you update, you only need to copy a small diff. This is much faster than running a new VM.



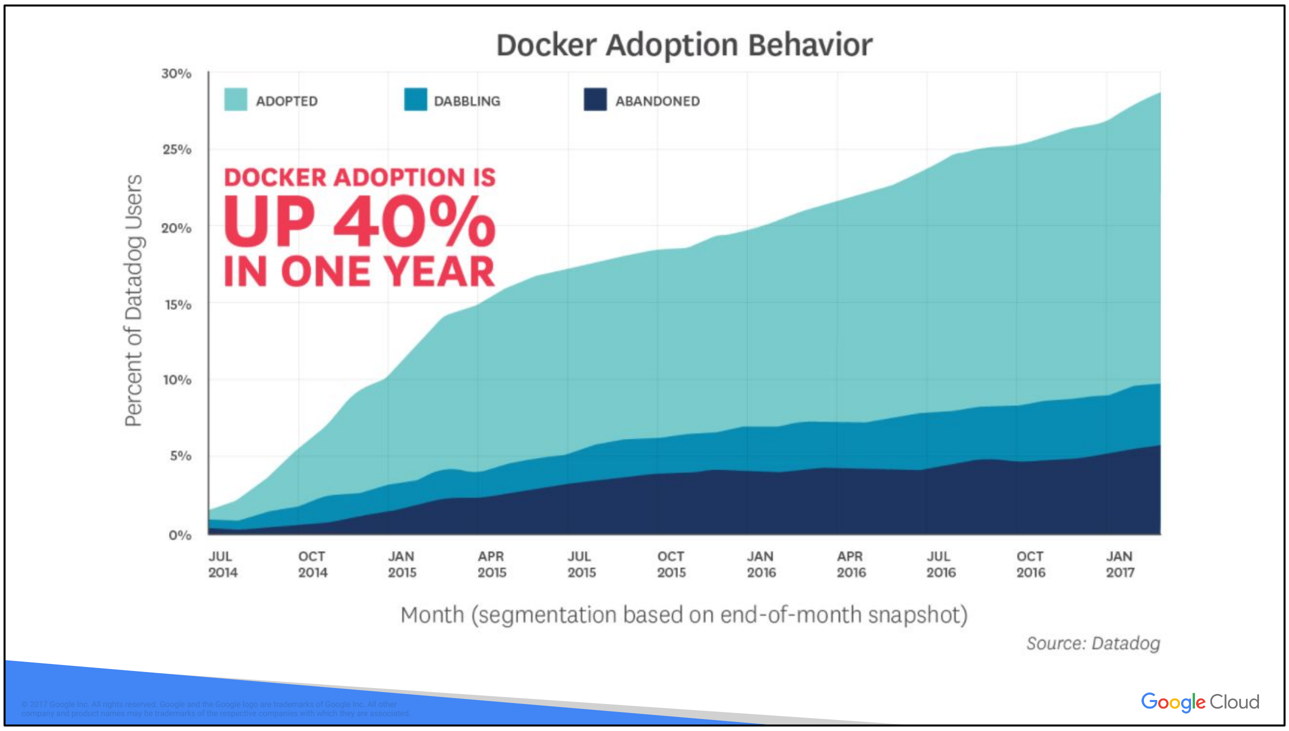
Introduction to Containers and Docker

Part 2: Docker

Version 1.6

Google Cloud

Now we'll take a look at how Docker helps build, run, and manage containers.

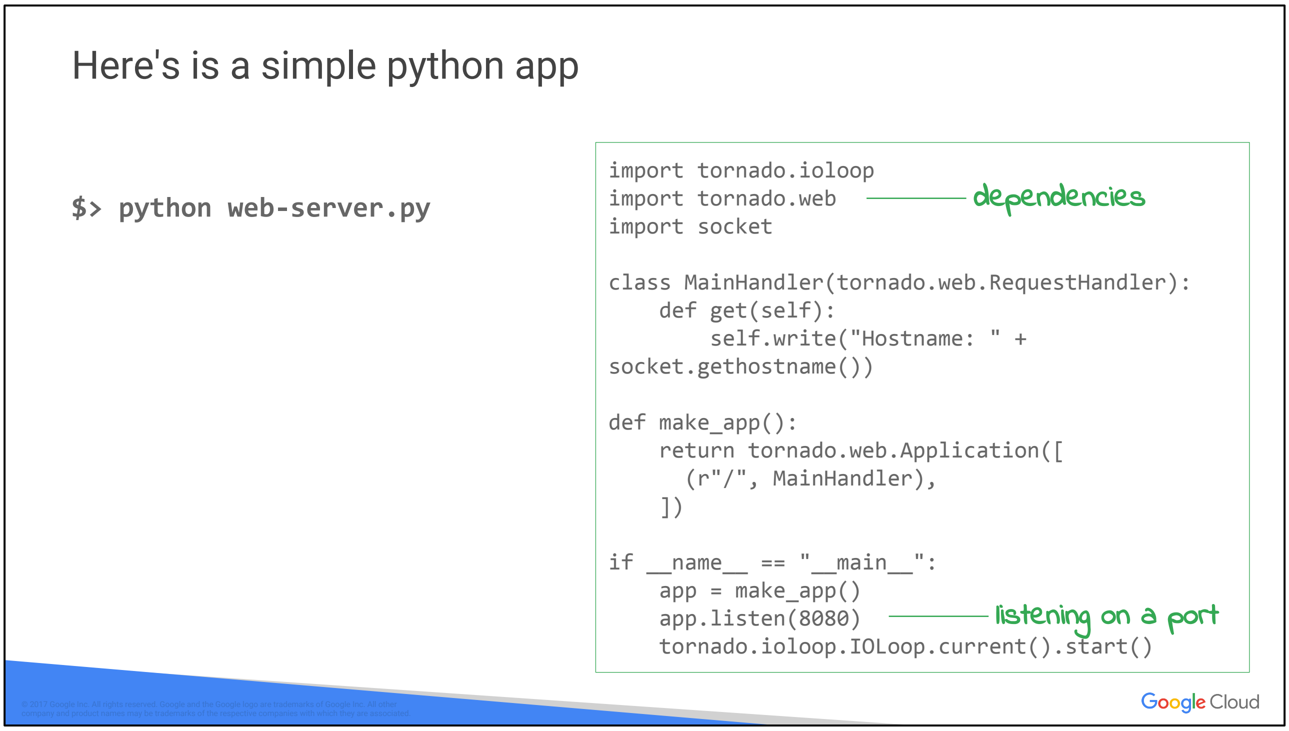


Docker helps you describe what's in a container: which software packages, which versions, where to get them, and how to install and start them.

Docker has made containers easy to pick up and has done an excellent job of advancing the industry.

Docker enables you to build a container that will run on the Docker container runtime. Their tools make it very easy to build a container that will run on any machine that has the Docker runtime installed, such as your laptop, an on-premises server, or a virtual server in the cloud. You will build a container in your lab today.

Image source from Datadog at https://www.datadoghq.com/docker-adoption/



Here's is a simple python app

```
$> python web-server.py
```

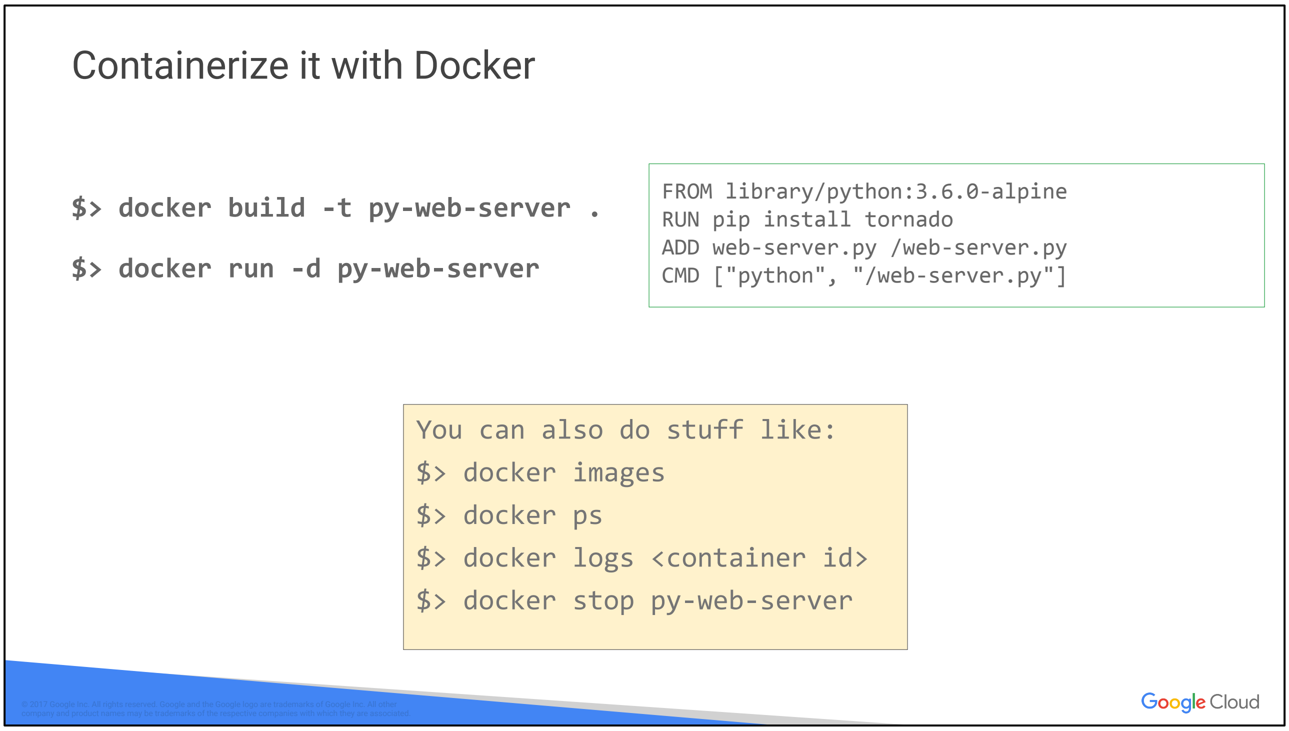
```
import tornado.ioloop
import tornado.web
import socket

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hostname: " +
socket.gethostname())

def make_app():
    return tornado.web.Application([
      (r"/", MainHandler),
    ])

if __name__ == "__main__":
    app = make_app()
    app.listen(8080)
    tornado.ioloop.IOLoop.current().start()
```

dependencies
listening on a port

Google Cloud

Here is a simple python file. All it does is listen to port 8080 and respond by writing the hostname.

We can test this the way that you would test any python app by typing in "python web-server.py," and then if you curl or go to a browser to port 8080, you'd see your hostname.



Containerize it with Docker

```
$> docker build -t py-web-server .

$> docker run -d py-web-server
```

```
FROM library/python:3.6.0-alpine
RUN pip install tornado
ADD web-server.py /web-server.py
CMD ["python", "/web-server.py"]
```

```
You can also do stuff like:
$> docker images
$> docker ps
$> docker logs <container id>
$> docker stop py-web-server
```

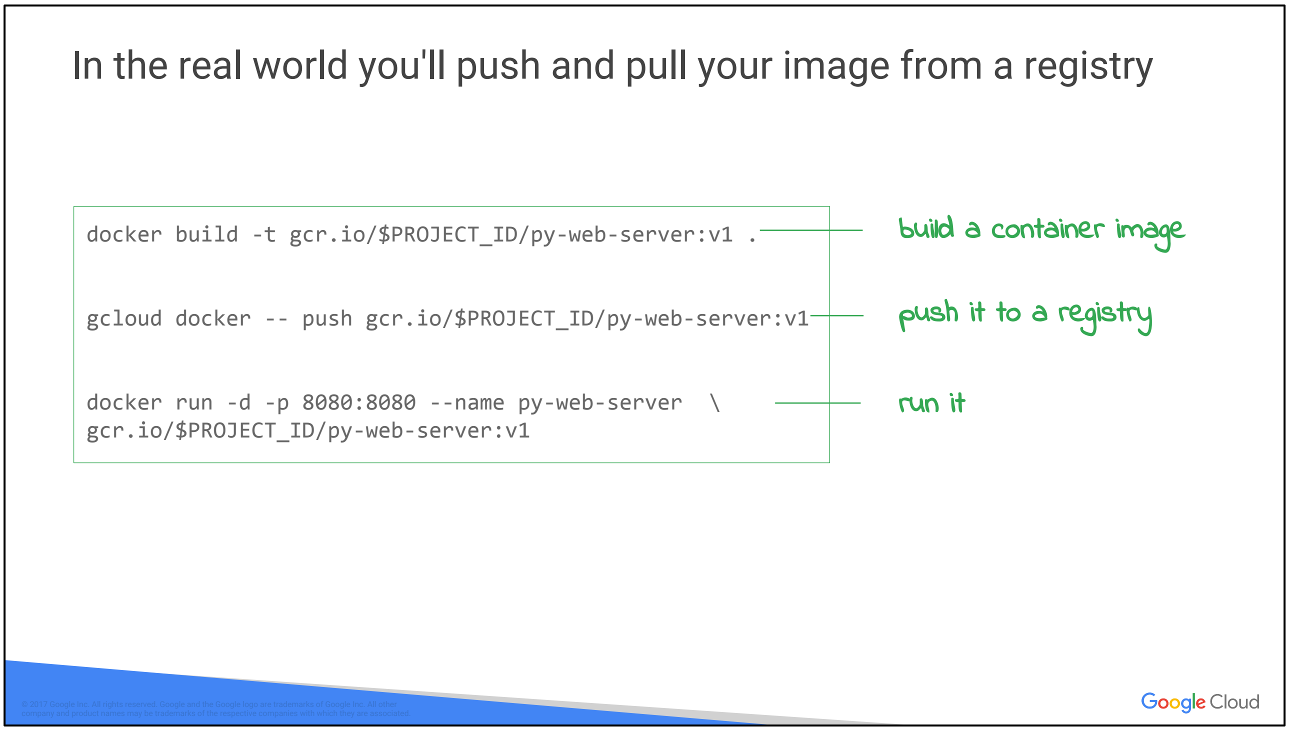
Google Cloud

Now containerize it. Here is the Dockerfile. It says that you want to use python version 3.6 and run "python web-server.py"

You use the "docker build" command to build the container.

You use the "docker run" command to run the container.

There are also other docker commands that you can use to manage your docker containers.



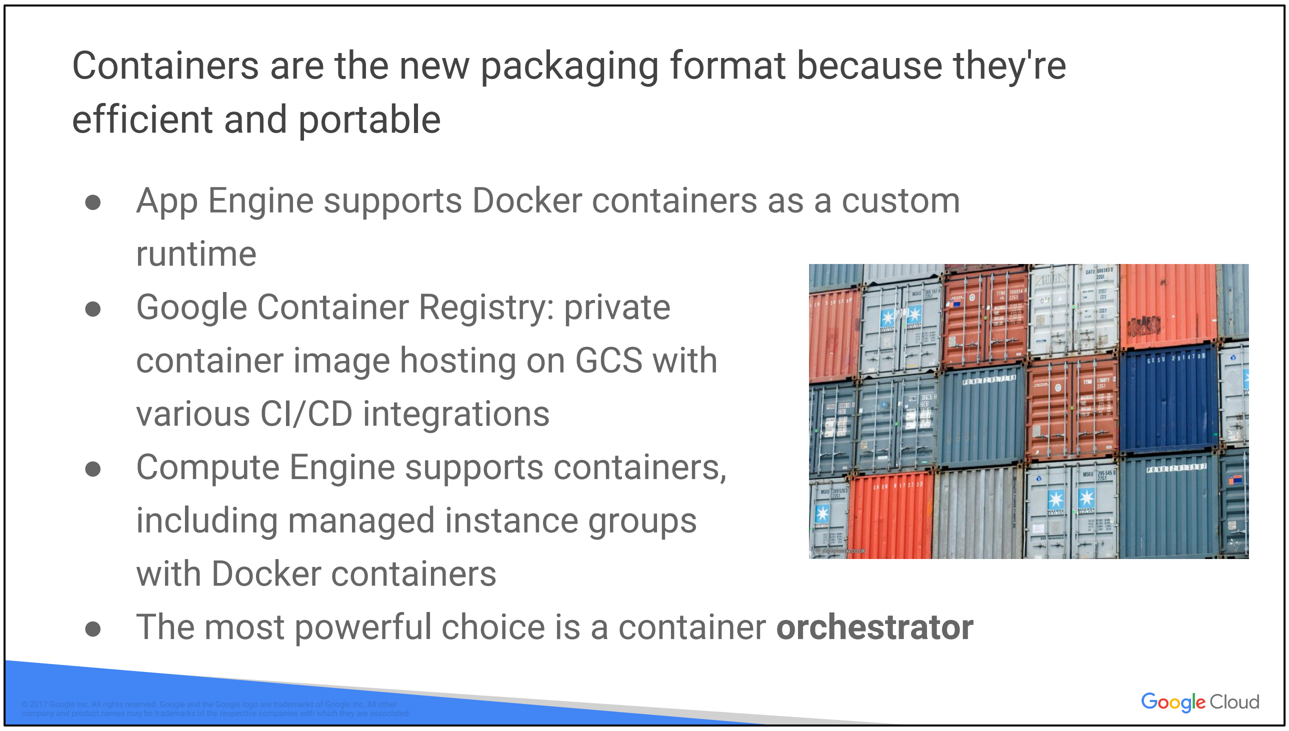
In the real world you'll push and pull your image from a registry

```
docker build -t gcr.io/$PROJECT_ID/py-web-server:v1 .

gcloud docker -- push gcr.io/$PROJECT_ID/py-web-server:v1

docker run -d -p 8080:8080 --name py-web-server  \
gcr.io/$PROJECT_ID/py-web-server:v1
```

build a container image
push it to a registry
run it

Google Cloud

In the real world, you likely wouldn't just create a docker image locally and run it, etc. You'd want to push and pull your image to and from a container registry. There are many public container registries that you can use, or you can create a private container registry to store container images for your organization. Google Container Registry is one container registry.



Containers are the new packaging format because they're efficient and portable

- App Engine supports Docker containers as a custom runtime
- Google Container Registry: private container image hosting on GCS with various CI/CD integrations
- Compute Engine supports containers, including managed instance groups with Docker containers
- The most powerful choice is a container **orchestrator**

Google Cloud

So as you can see, the container packaging format is really efficient and portable.

In fact, it's so efficient and portable that it is used quite widely. In the Google Cloud Platform, the Google App Engine flexible environment uses them. On GCE, a customer can choose to use containers in a MIG. But if you plan to run containers in production, the best choice is in a cluster on Google Kubernetes Engine.

You also saw how, using Docker, you can easily build and run containers. The Docker command line tools are fine for development, but what if you want to run your containers in a production environment? Well, you need a way to quickly scale containers, run constant health checks to ensure that your containers are up and running, etc. In other words, you need a way to manage and orchestrate your containers. This is where Kubernetes comes in.

Lab