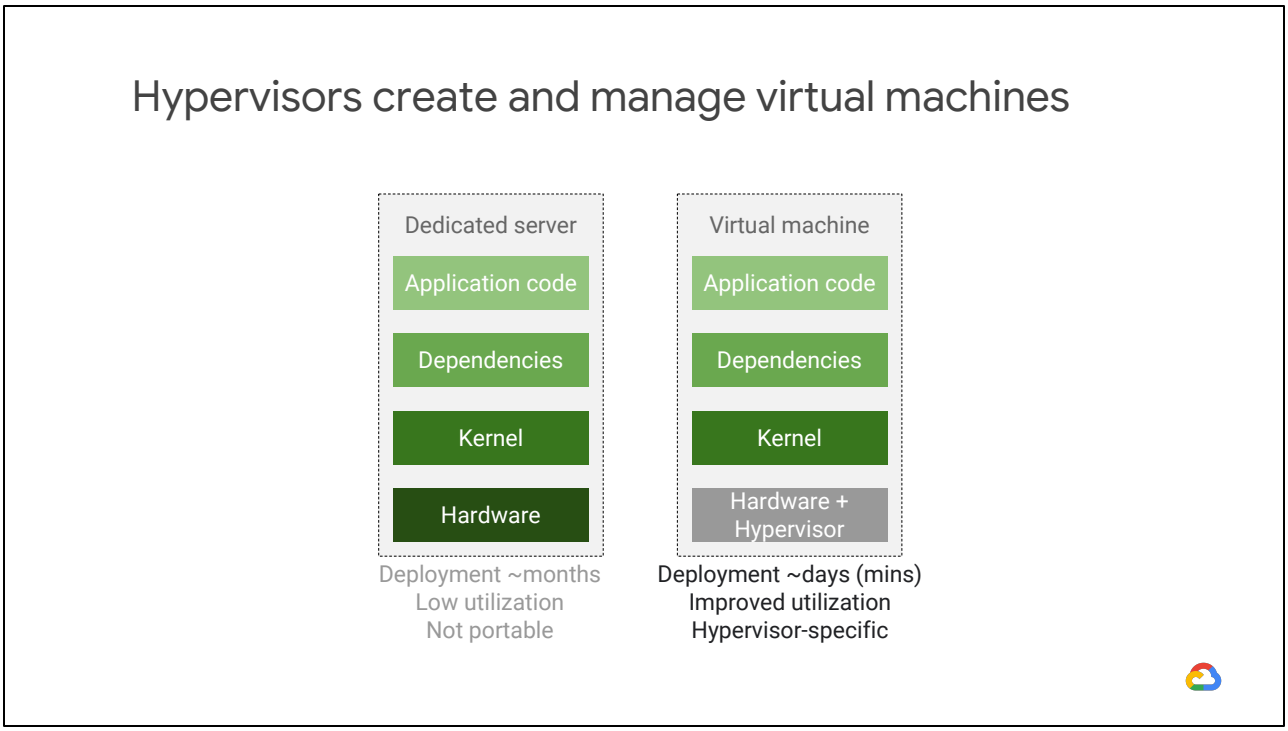
Welcome to the ‘Introduction to Containers and Kubernetes’ module. In this module you will learn what Containers are, what their benefits are for application deployment, how containers are configured and built, what functions container management solutions like Kubernetes provide, and what the advantages of Google Kubernetes Engine are compared to building your own Container Management infrastructure.



Not very long ago, the default way to deploy an application was on its own physical computer. To set one up, you'd find physical space, power, cooling, network connectivity for it, and then install an operating system, any software dependencies, and finally the application. If you need more processing power, redundancy, security, or scalability, add more computers. It was very common for each computer to have a single purpose: for example database, web server, or content delivery.

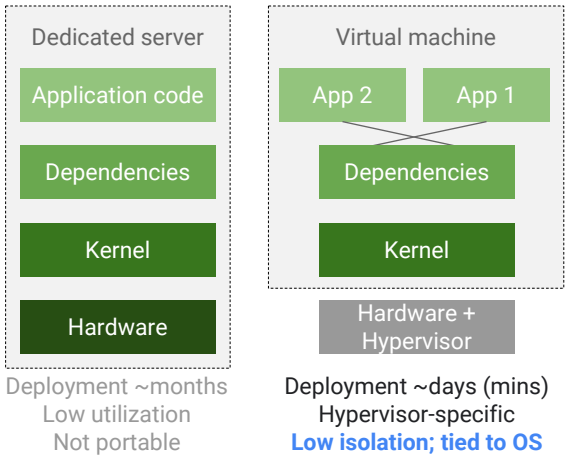
This practice wasted resources and took a lot of time to deploy, maintain, and scale. It also wasn't very portable: applications were built for a specific operating system and sometimes for specific hardware.

Virtualization helped by making it possible to run multiple virtual servers and operating systems on the same physical computer. A hypervisor is the software layer that breaks the dependencies of an operating system on the underlying hardware and allows several virtual machines to share that hardware. KVM is one well-known

hypervisor. Today, you can use virtualization to deploy new servers fairly quickly.

Adopting virtualization means that it takes us less time to deploy new solutions. We waste less of the resources of the physical computers we use. And we get some improved portability, because virtual machines can be imaged and moved around. However, the application, all its dependencies, and operating system are still bundled together. It’s not easy to move a VM from one hypervisor product to another, and every time you start a VM, its operating system takes time to boot up.

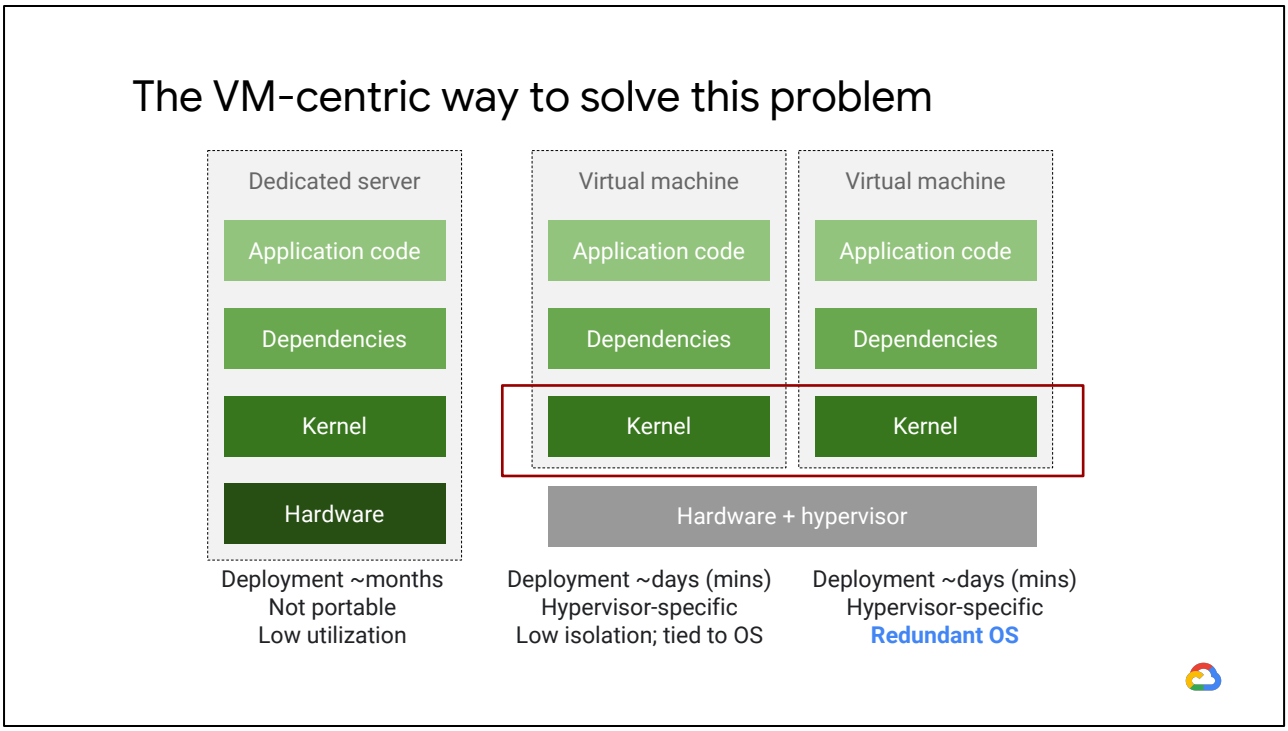
Running multiple apps on a single VM



Running multiple applications within a single VM creates another problem: applications that share dependencies are not isolated from each other. The resource requirements of one application can starve other applications of the resources they need. Also, a dependency upgrade for one application might cause another to stop working.

You can try to solve this problem with rigorous software engineering policies. For example, you can lock down the dependencies so that no application is allowed to make changes; but this leads to new problems because dependencies need to be upgraded occasionally.

You can add integration tests to ensure that applications work. Integration tests are great. But dependency problems can cause novel failure modes that are hard to troubleshoot. And it really slows down development if you have to rely on integration tests to confirm the basic integrity of your application environment.

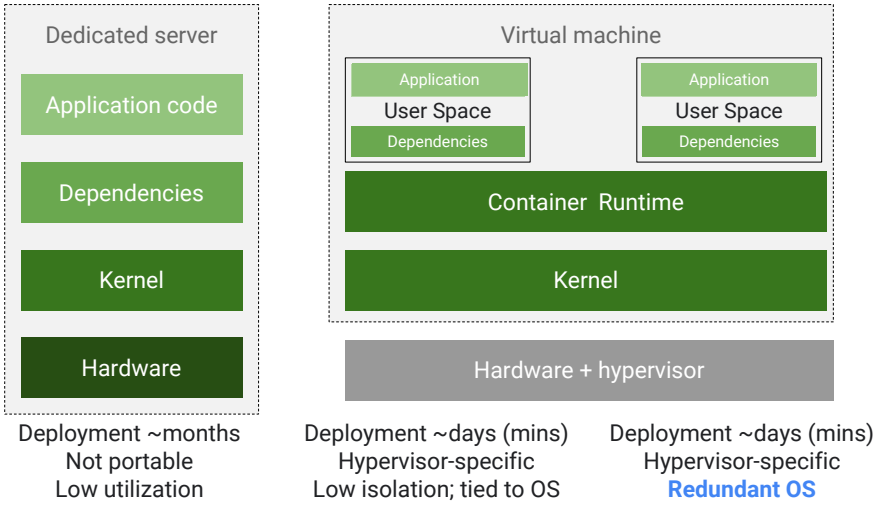


The VM-centric way to solve this problem is to run a dedicated virtual machine for each application.

Each application maintains its own dependencies, and the kernel is isolated so one application won't affect the performance of another. The result is that two complete copies of the kernel are running.

Scale this to hundreds or thousands of applications and you see its limitations; imagine trying to do a kernel update. So, for large systems, dedicated VMs are redundant and wasteful. VMs are also relatively slow to start up, because an entire operating system has to boot.

User space abstraction and containers



A more efficient way to resolve the dependency problem is to implement abstraction at the level of the application and its dependencies. You don't have to virtualize the entire machine, or even the entire operating system, but just the user space. The user space is all of the code that resides above the kernel, and it includes applications and their dependencies.

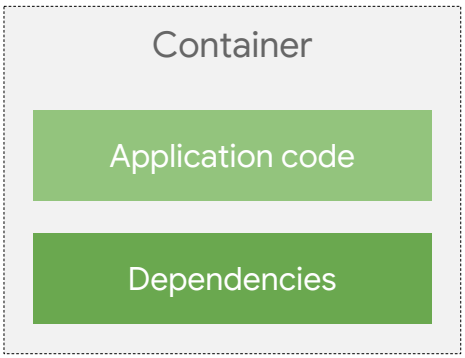
This is what it means to create containers. Containers are isolated user spaces for running application code.

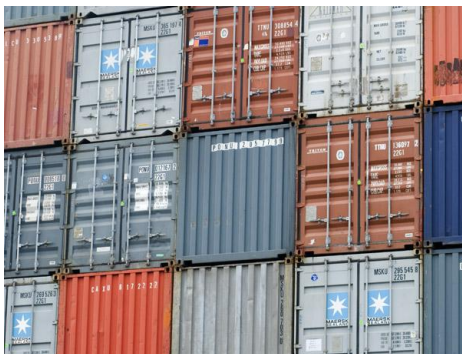
Containers are lightweight because they don't carry a full operating system. They can be scheduled or packed tightly onto the underlying system, which is very efficient. And they can be created and shut down very quickly, because you're just starting and stopping operating system processes, and not booting an entire VM and initializing an operating system for each application.

Developers appreciate this level of abstraction because they don't want to worry about the rest of the system. Containerization is the

next step in the evolution of managing code.

Containers are lightweight, standalone, resource-efficient, portable, executable packages



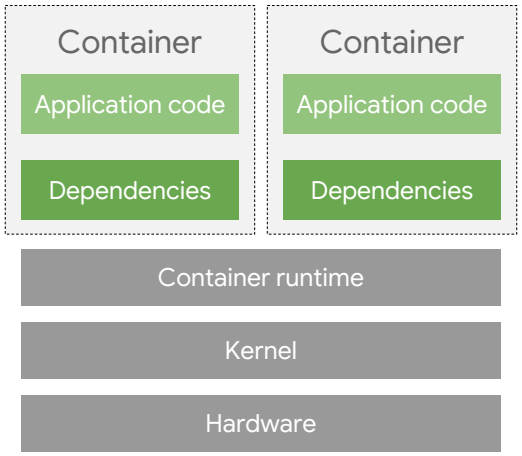


You now understand containers as delivery vehicles for application code. They're lightweight, standalone, resource-efficient, portable, executable packages.

You develop application code in the usual way: on desktops, laptops, and servers. The container allows you to execute your final code on VMs without worrying about software dependencies like application runtimes, system tools, system libraries, and settings. You packaged your code with all the dependencies it needs, and the engine that executes your container is responsible for making them available at runtime.

Image source:
By Steve Gibson hosted at https://commons.wikimedia.org/wiki/File:Shipping_containers_at_Clyde.jpg

Why developers like containers



Containers appeal to developers because they're an application-centric way to deliver high-performing, scalable applications.

Containers also allow developers to safely make assumptions about the underlying hardware and software.

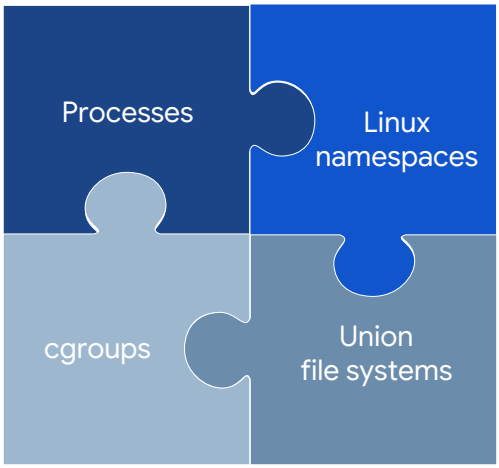
With a Linux kernel underneath, you no longer have code that works on your laptop but doesn't work in production; the container is the same and runs anywhere. If you make incremental changes to a container based on a production image, you can deploy it very quickly with a single file copy. This speeds development.

Finally, containers make it easier to build applications that use the microservices design pattern: that is, loosely coupled, fine-grained components. This modular design pattern allows the operating system to scale and upgrade components of an application without affecting the application as a whole.

An application and its dependencies are called an *image*. A container is simply a running instance of an image. By building software into container images, developers can easily package and ship an application without worrying about the system it will be running on.

You need software to build container images and to run them. Docker is one tool that does both. Docker is an open-source technology that allows you to create and run applications in containers, but it doesn't offer a way to orchestrate those applications at scale as Kubernetes does. In this course, we will use Google's Cloud Build to create Docker-formatted container images.

Containers use a varied set of Linux technologies



Containers are not an intrinsic, primitive feature of Linux. Instead, their power to isolate workloads is derived from the composition of several technologies.

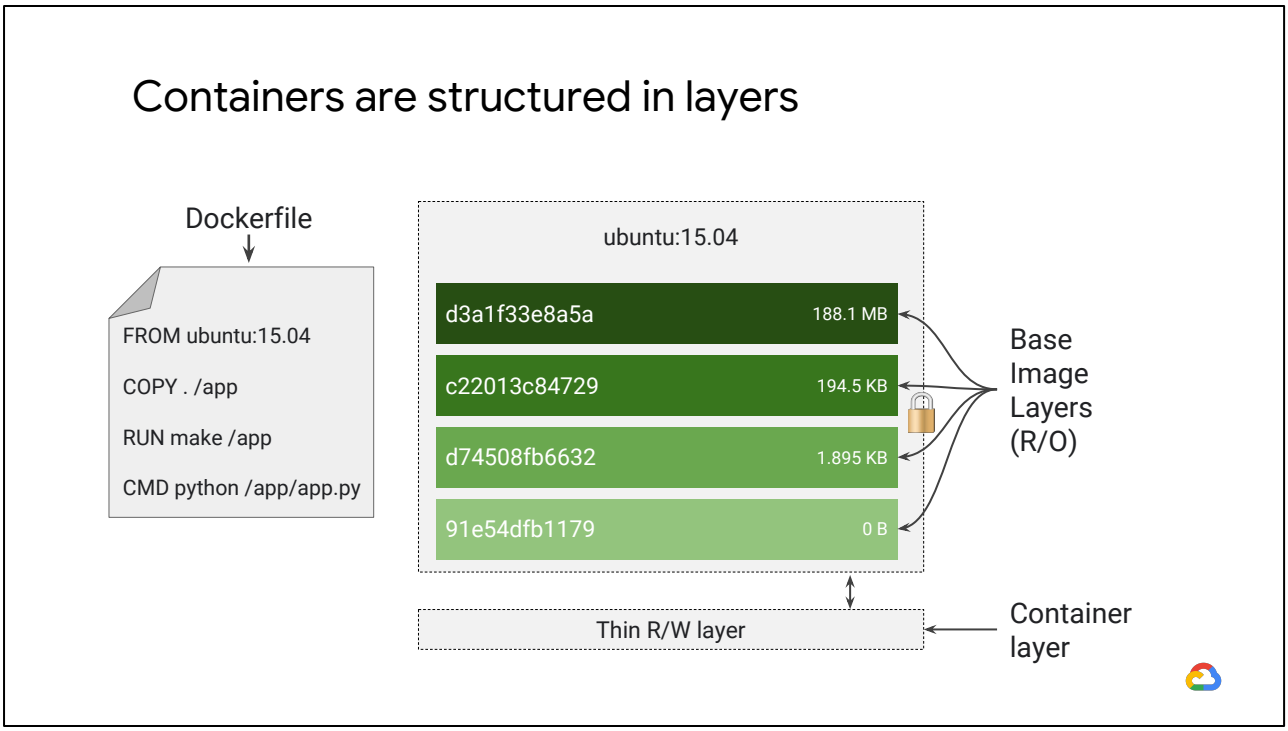
One foundation is the Linux process. Each Linux process has its own virtual memory address space, separate from all others, and Linux processes are rapidly created and destroyed.

Containers use Linux namespaces to control what an application can see: process ID numbers, directory trees, IP addresses, and more. By the way, Linux namespaces are not the same thing as Kubernetes namespaces, which you will learn about later in this course.

Containers use Linux cgroups to control what an application can use: its maximum consumption of CPU time, memory, I/O bandwidth, and other resources.

Finally, containers use union file systems to efficiently encapsulate

applications and their dependencies into a set of clean, minimal layers. Now let’s see how this works.



A container image is structured in layers. The tool you use to build the image reads instructions from a file called the “container manifest.” In the case of Docker-formatted container images, that’s called a Dockerfile. Each instruction in the Dockerfile specifies a layer inside the container image. Each layer is read-only. (When a container runs from this image, it will also have a writable, ephemeral topmost layer.)

Let’s look at a simple Dockerfile. This Dockerfile will contain four commands, each of which creates a layer. (At the end of this discussion, I’ll explain why this Dockerfile is a little oversimplified for modern use.)

The FROM statement starts out by creating a base layer, pulled from a public repository. This one happens to be the Ubuntu Linux runtime environment of a specific version.
The COPY command adds a new layer, containing some files copied in from your build tool’s current directory.
The RUN command builds your application using the “make”

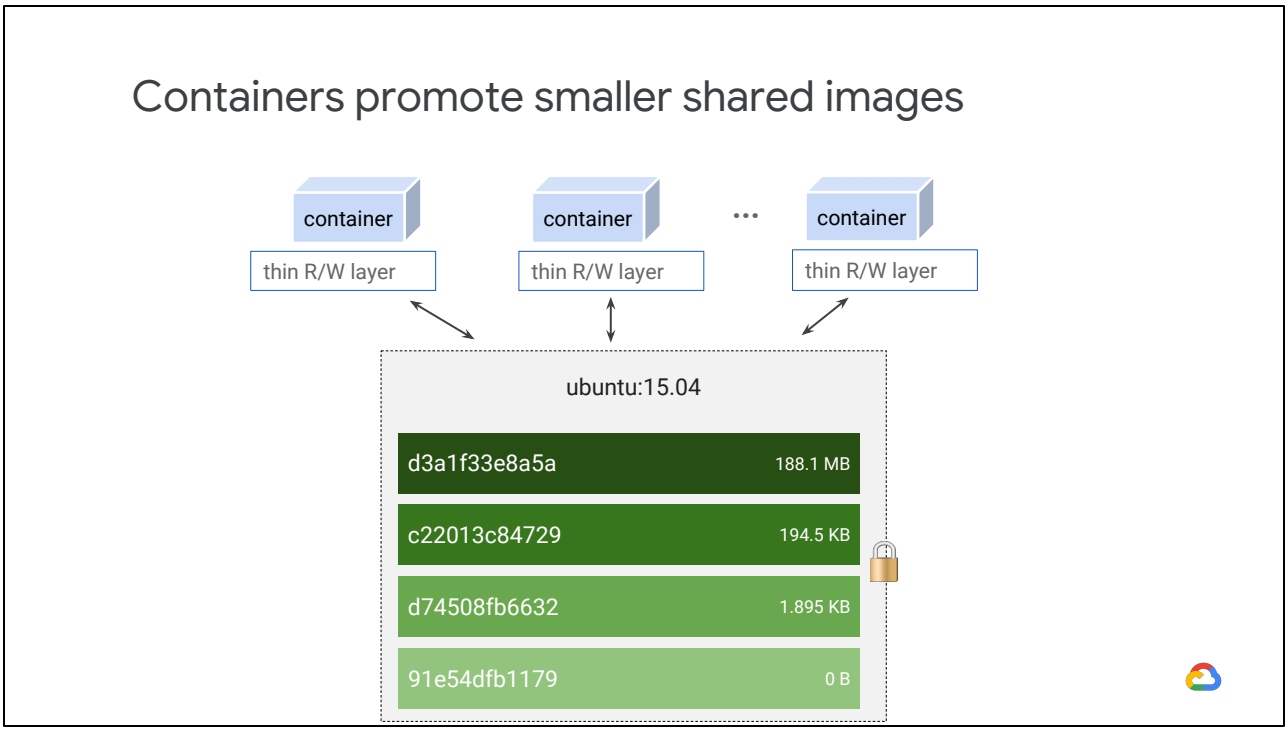
command and puts the results of the build into a third layer.

And finally, the last layer specifies what command to run within the container when it is launched. Each layer is only a set of differences from the layer before it. When you write a Dockerfile, you should organize from the layers likely to change, through to the layers most likely to change.

By the way, I promised that I would explain how this Dockerfile example is oversimplified. These days, the best practice is *not* to build your application in the very same container that you ship and run. After all, your build tools are at best just clutter in a deployed container, and at worst they are an additional attack surface. Today, application packaging relies on a multi-stage build process, in which one container *builds* the final executable image, and a separate container receives only what is needed to *run* the application. Fortunately for us, the tools we use support this practice.

When you launch a new container from an image, the container runtime adds 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. And they're ephemeral: When the container is deleted, the contents of this writable layer are lost forever. The underlying container image remains unchanged. This fact about containers has an implication for your application design: whenever you want to store data permanently, you must do so somewhere *other than* a running container image. You will learn about several choices in this specialization.



Because each container has its own writable container layer, and all changes are stored in this 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.

Because each layer is only a set of differences from the layer before it, you get smaller images.

For example, your base application image may be 200 MB, but the difference to the next point release might only be 200 KB. When you build a container, instead of copying the whole image, it creates a layer with just the difference. When you run a container, the container runtime pulls down the layers it needs. When you update, you only need to copy the difference. This is much faster than running a new virtual machine.

How can you get or create containers?

Download containerized software from a container registry such as gcr.io.

`docker`

Build your own container using the open-source docker command.

Build your own container using Cloud Build.

It's very common to use publicly available open-source container images as the base for your own images, or for unmodified use. For example, you've already seen the "ubuntu" container image, which provides a Ubuntu Linux environment inside a container. "Alpine" is a popular Linux environment in a container, noted for being very small. The nginx web server is frequently used in its container packaging.

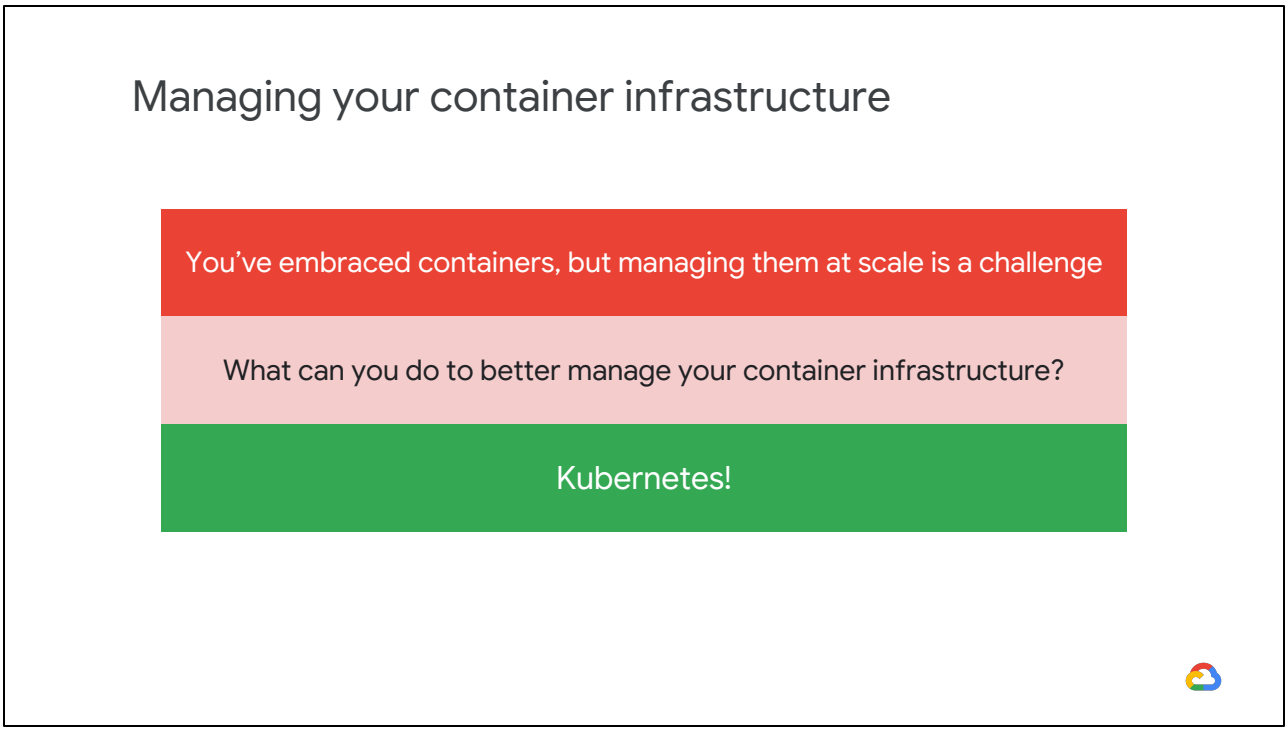
Google maintains a container registry, gcr.io. This registry contains many public, open-source images, and Google Cloud customers also use it to store their private images in a way that integrates with Cloud IAM. Google Container Registry is integrated with Cloud IAM, so, for example, you can use it to store images that aren't public -- instead, they are private to your project.

You can also find container images in other public repositories: Docker Hub Registry, GitLab, and others.

The open-source docker command is a popular way to build your

own container images. It's widely known and widely available.

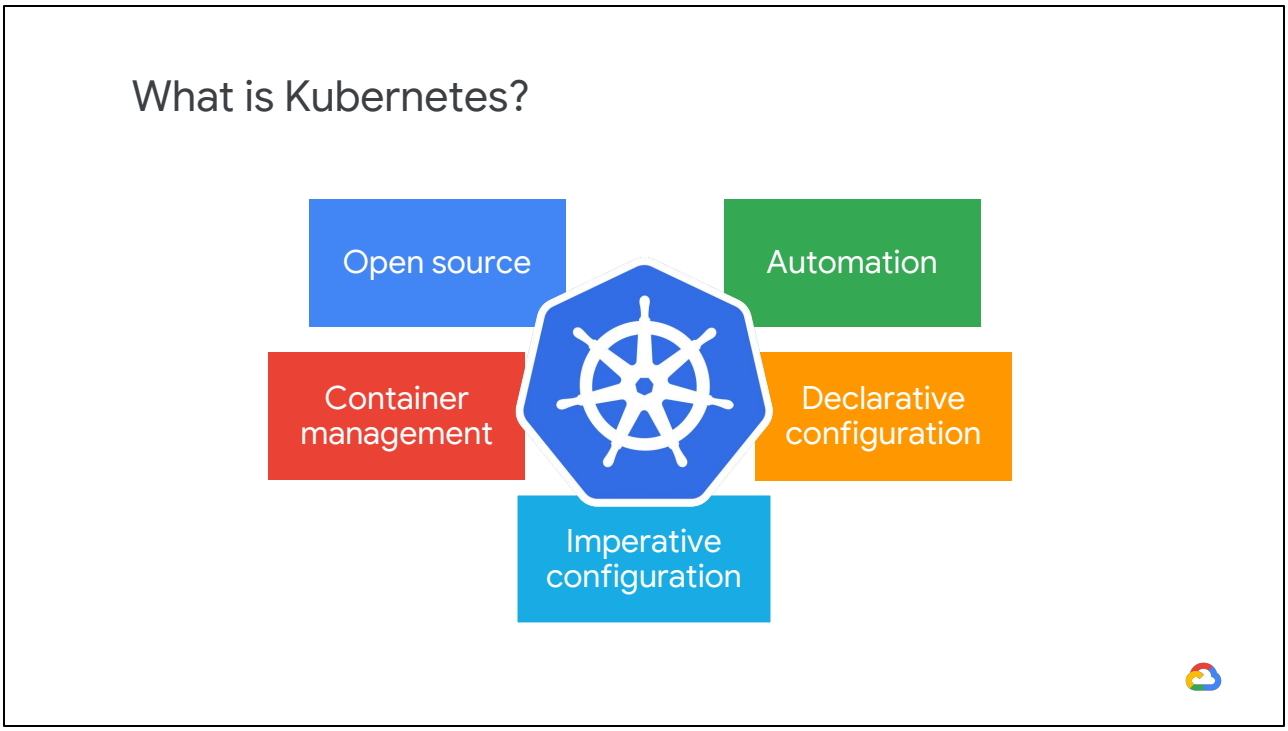
One downside of building containers with the docker command is that you must trust the computer that you do your builds on. Google provides a managed service for building containers that's integrated with Cloud IAM. This service is called Cloud Build, and we'll use it in this course.



Your organization has really embraced the idea of containers. Because containers are so lean, your coworkers are creating them in numbers far exceeding the counts of virtual machines you used to have. And the applications running in them need to communicate over the network, but you don't have a network fabric that lets containers find each other. You need help.

How can you manage your container infrastructure better?

Kubernetes is an open-source platform that helps you orchestrate and manage your container infrastructure on-premises or in the cloud.



So what is Kubernetes? It's a container-centric management environment. Google originated it and donated it to the open-source community. Now it's a project of the vendor-neutral Cloud Native Computing Foundation.

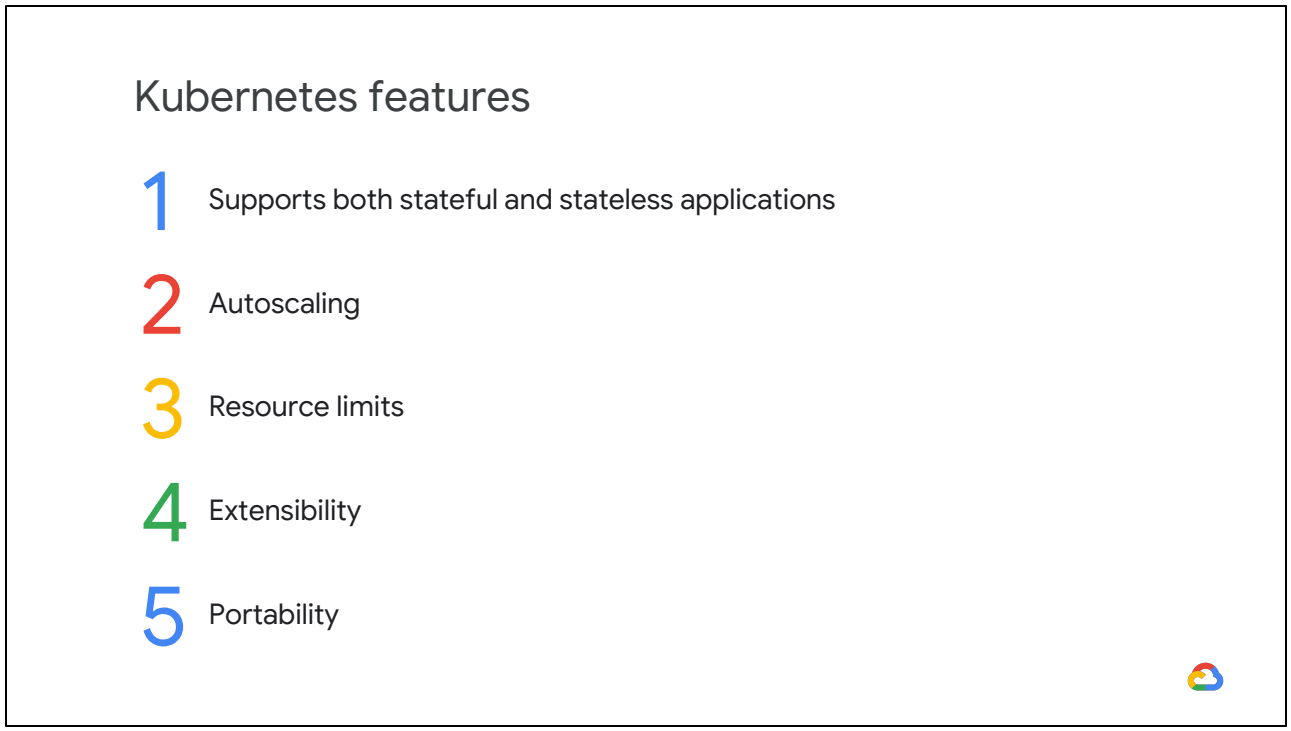
It automates the deployment, scaling, load balancing, logging, monitoring, and other management features of containerized applications. These are the features that are characteristic of typical platform-as-a-service solutions.

Kubernetes also facilitates the features of infrastructure-as-a-service, such as allowing a wide range of user preferences and configuration flexibility.

Kubernetes supports declarative configurations. When you administer your infrastructure declaratively, you describe the

desired state you want to achieve, instead of issuing a series of commands to achieve that desired state. Kubernetes's job is to make the deployed system conform to your desired state and to keep it there in spite of failures. Declarative configuration saves you work. Because the system's desired state is always documented, it also reduces the risk of error.

Kubernetes also allows imperative configuration, in which you issue commands to change the system's state. But administering Kubernetes at scale imperatively would be a big missed opportunity. One of the primary strengths of Kubernetes is its ability to automatically keep a system in a state you declare. Experienced Kubernetes administrators use imperative configuration only for quick temporary fixes and as a tool in building a declarative configuration.



Now that you know what Kubernetes is, let's talk about some of its features.

Kubernetes supports different workload types. It supports stateless applications, such as Nginx or Apache web servers, and stateful applications where user and session data can be stored persistently. It also supports batch jobs and daemon tasks.

Kubernetes can automatically scale in and out containerized applications based on resource utilization.

You can specify resource request levels and resource limits for your workloads, and Kubernetes will obey them. These resource controls let Kubernetes improve overall workload performance within a cluster.

Developers extend Kubernetes through a rich ecosystem of plugins and addons. For example, there is a lot of creativity going on currently with Kubernetes Custom Resource Definitions, which bring the Kubernetes declarative management model to an amazing variety of other things that need to be managed. The primary focus of this specialization, though, is architecting with Kubernetes, because it's provided as a service by Google Cloud, so extending Kubernetes is not in our scope.

Because it's open-source, Kubernetes also supports workload portability across on-premises or multiple cloud service providers such as GCP and others. This allows Kubernetes to be deployed anywhere. You can move Kubernetes workloads freely without vendor lock-in.

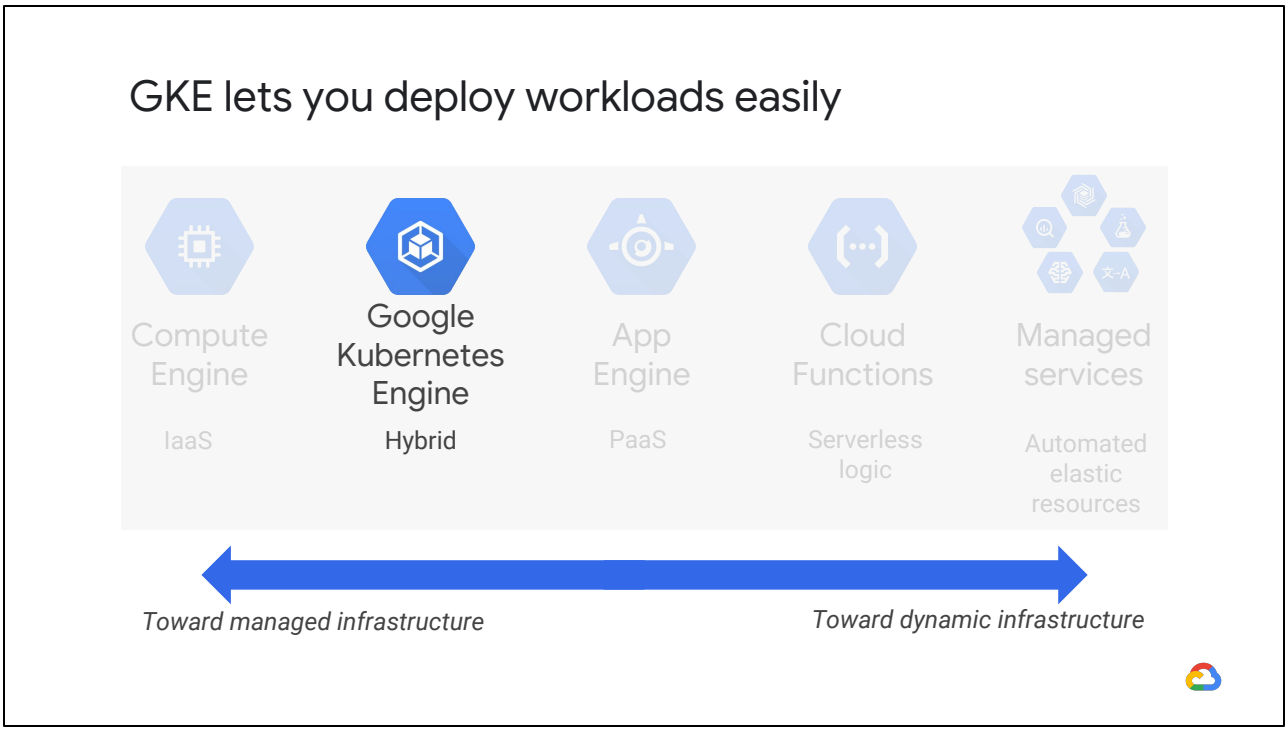
Managing Kubernetes within GCP

Kubernetes is powerful, but it's a full-time job managing the infrastructure

Is there a managed service for Kubernetes within GCP?

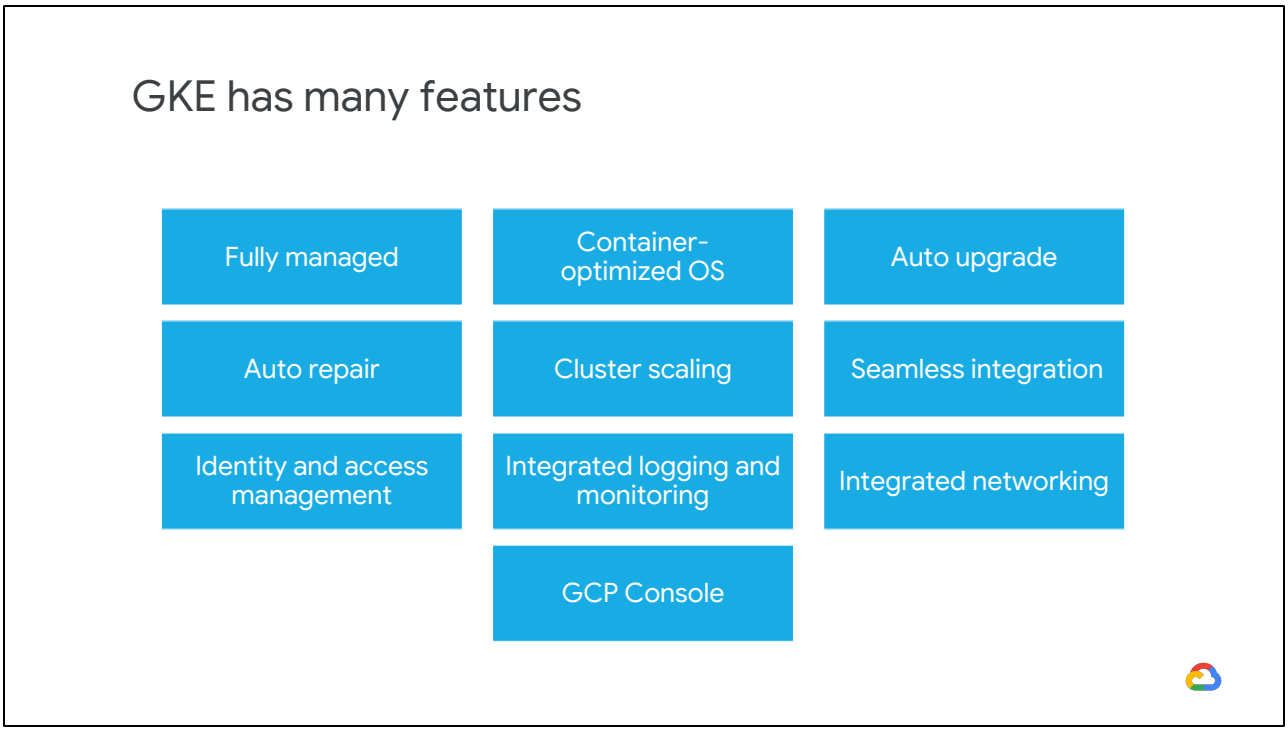
Yes! Google Kubernetes Engine

What if you have begun using Kubernetes in your environment, but the infrastructure has become a burden to maintain?
Is there anything within Google Cloud Platform that can help you?
Absolutely yes. GCP offers a managed Kubernetes solution called Google Kubernetes Engine.



Google Kubernetes Engine is a managed Kubernetes service on Google infrastructure. GKE helps you to deploy, manage, and scale Kubernetes environments for your containerized applications on GCP.

More specifically, GKE is a component of the GCP compute offerings. It makes it easy to bring your Kubernetes workloads into the cloud.



GKE is fully managed, which means you don't have to provision the underlying resources.

GKE uses a container-optimized operating system to run your workloads. These operating systems are maintained by Google and are optimized to scale quickly with a minimal resource footprint. The container-optimized OS is discussed later in this course.

When you use GKE, you start by directing the service to instantiate a Kubernetes system for you. This system is called a cluster. GKE's auto-upgrade feature can be enabled to ensure that your clusters are always automatically upgraded with the latest stable version of Kubernetes.

The virtual machines that host your containers in a GKE cluster are called nodes. If you enable GKE's auto-repair feature, the service

will repair unhealthy nodes for you. It'll make periodic health checks on each node of the cluster. If a node is determined to be unhealthy and require repair, GKE will drain the node (in other words, cause its workloads to gracefully exit) and recreate the node.

Just as Kubernetes supports scaling workloads, GKE supports scaling the cluster itself.

GKE seamlessly integrates with Google's Cloud Build and Container Registry. **This allows you to automate deployment using private container images that you have securely stored in Container Registry.**

GKE also integrates with Google's Identity and Access Management, which allows you to control access through the use of accounts and role permissions.

Stackdriver is Google Cloud's system for monitoring and management for services, containers, applications, and infrastructure. GKE integrates with Stackdriver Monitoring to help you understand your applications' performance.

GKE is integrated with Google Virtual Private Clouds and makes use of GCP's networking features.

And finally the GCP Console provides insights into GKE clusters and their resources and allows you to view, inspect and delete resources in those clusters. You might be aware that open-source Kubernetes contains a dashboard, but it takes a lot of work to set it up securely. But the GCP Console is a dashboard for your GKE clusters and workloads that you don't have to manage, and it's