



Introduction to Containers and Kubernetes



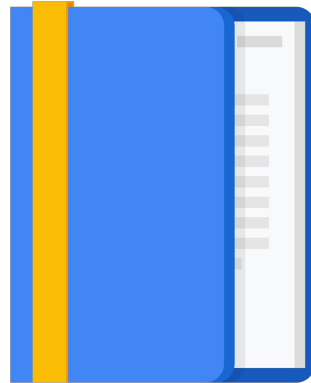
Welcome to the 'Introduction to Containers and Kubernetes' module. In this module you will learn what Containers are, what their benefits are for application deployment, how containers are configured and built, what functions container management solutions like Kubernetes provide, and what the advantages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

Learn how to ...

Create a container using Cloud Build.

Store a container in Container Registry.

Compare and contrast Kubernetes and Google Kubernetes Engine (GKE) features.



In this module, you'll learn how to create a container using Cloud Build, store a container in Container Registry, and compare and contrast Kubernetes and Google Kubernetes Engine features.

Agenda

Introduction to Containers

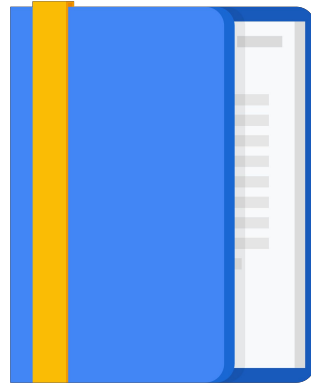
Lab: Working with Cloud Build

Introduction to Kubernetes

Introduction to Google Kubernetes Engine

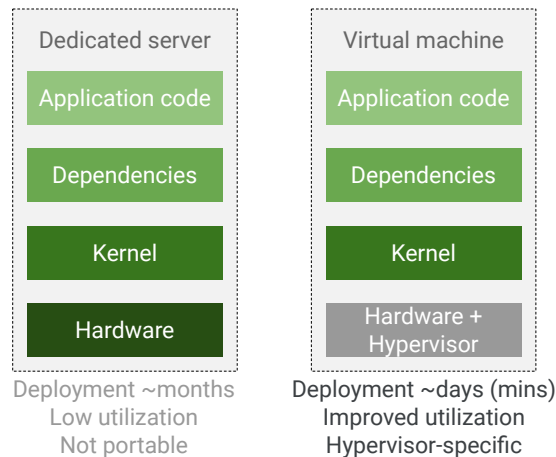
Computing Options

Summary



Let's start by introducing Containers. In this lesson you'll learn about the key features of containers and the advantages of using containers for application deployment compared to alternatives such as deploying apps directly to virtual machines. You'll learn about Google's Cloud Build, and see how to use it to build and manage your application images.

Hypervisors create and manage virtual machines



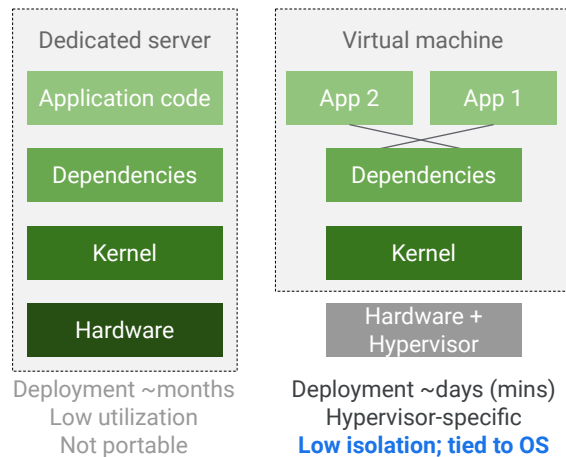
Not very long ago, the default way to deploy an application was on its own physical computer. To set one up, you'd find physical space, power, cooling, and network connectivity for it, and then install an operating system, any software dependencies, and finally the application. If you need more processing 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

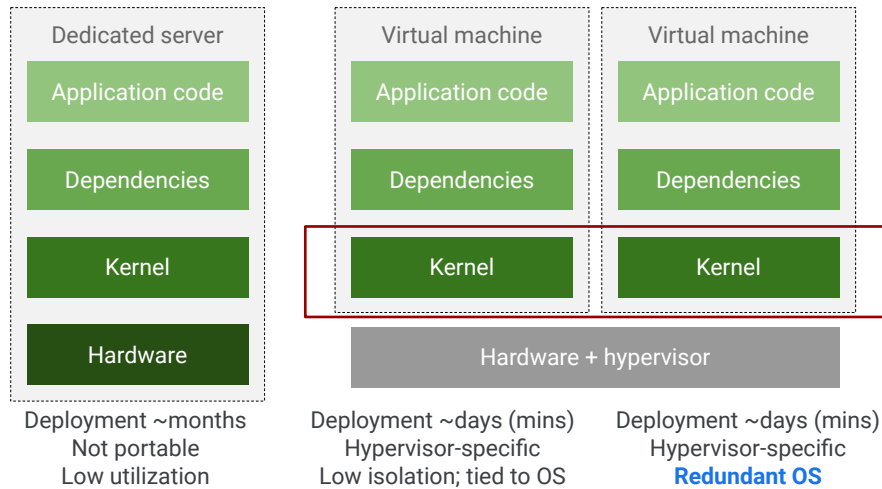


Running multiple applications within a single VM creates another problem: applications that share dependencies are not isolated from each other. The resource requirements of one application can starve other applications of the resources they need. Also, a dependency upgrade for one application might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

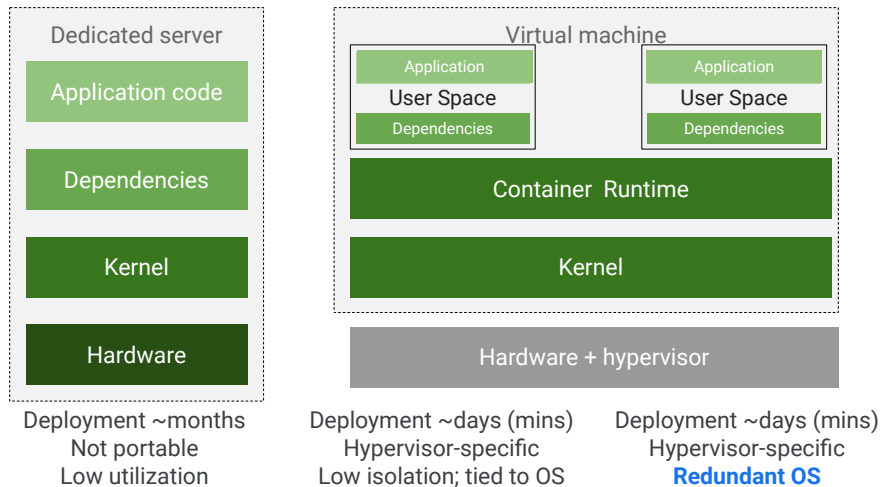


The VM-centric way to solve this problem is to run a dedicated virtual machine for each application.

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



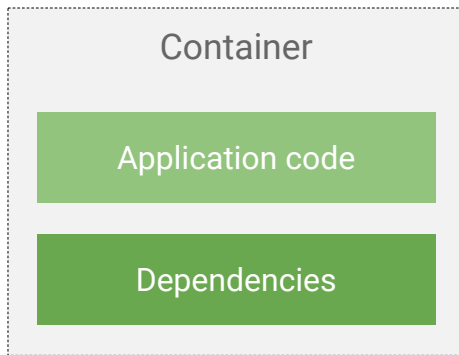
A more efficient way to resolve the dependency problem is to implement abstraction at the level of the application and its dependencies. You don't have to virtualize the entire machine, or even the entire operating system, but just the user space. The user space is all of the code that resides above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

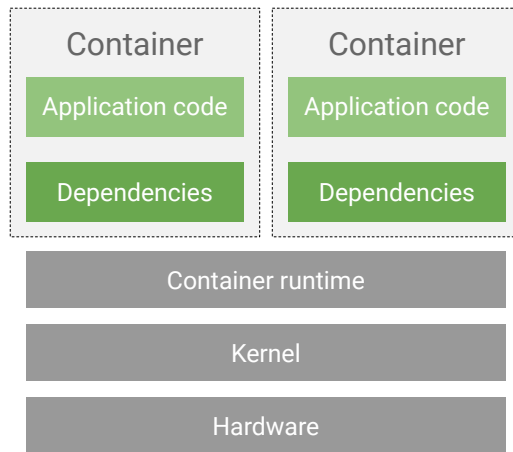
Containers are lightweight, standalone, resource-efficient, portable, executable packages



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.

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.

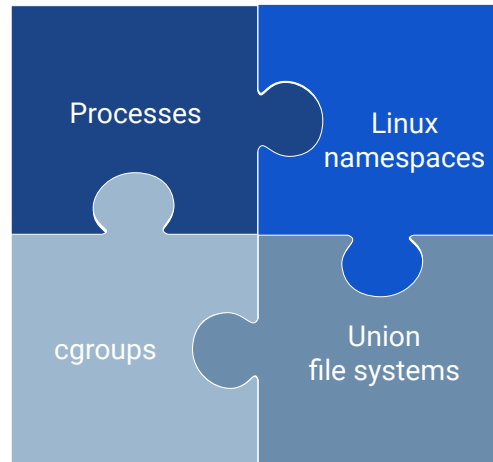
Finally, containers make it easier to build applications that use the microservices design pattern: that is, loosely coupled, fine-grained components. This modular design pattern allows the operating system to scale and upgrade components of an application without affecting the application as a whole.

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.

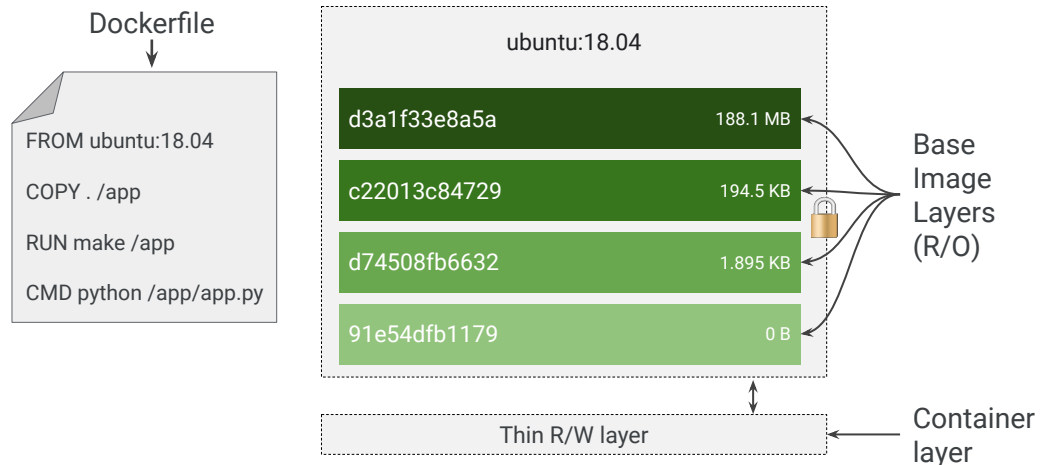
One foundation is the Linux process. Each Linux process has its own virtual memory address space, separate from all others, and Linux processes are rapidly created and destroyed.

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.

Containers are structured in layers



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 least likely to change, through to the layers most likely to change.

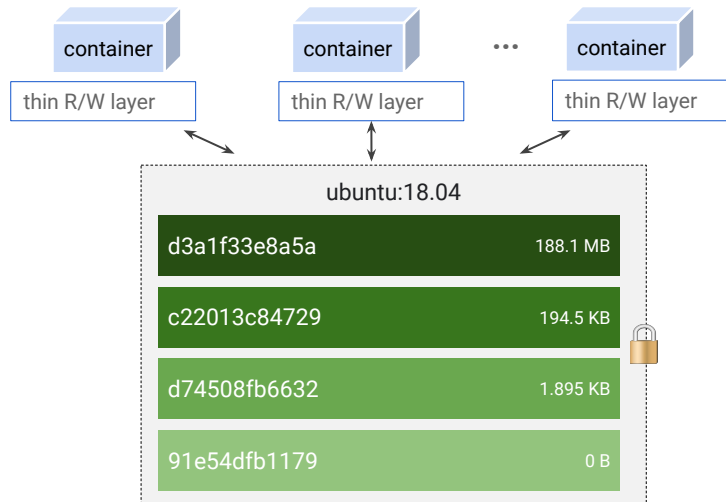
By the way, I promised that I would explain how this Dockerfile example is

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.

Containers promote smaller shared images



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 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?



Store and download container images from Artifact Registry

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.

Artifact Registry is the single place to store container images as well as language and OS packages. You can use Artifact Registry with other Google Cloud services namely IAM for access control, KMS for customer managed encryption keys, Cloud Build for CI/CD and scan for container vulnerabilities with Container Analysis.

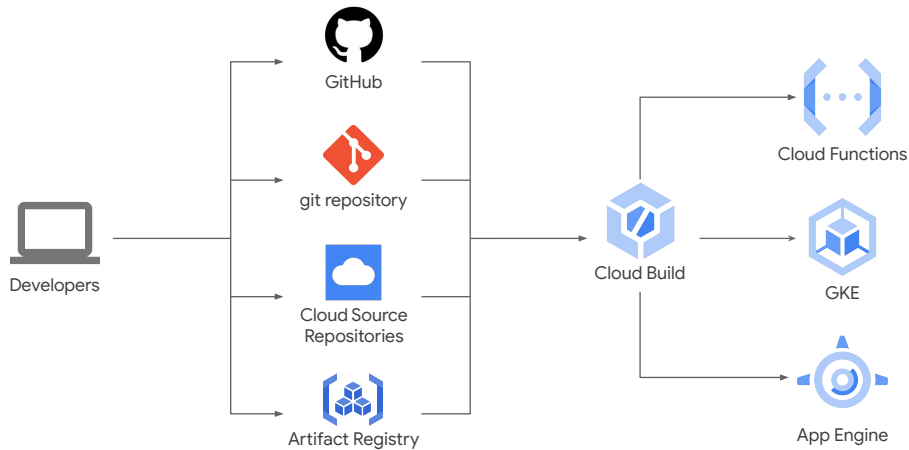
You can also find container images in other public repositories: Docker Hub Registry, GitLab, and others.

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 IAM. This service is called Cloud Build, and we'll use it in this course.

Retrieve the source code for your builds from a variety of storage locations



Cloud Build can retrieve the source code for your builds from many code repositories, including Cloud Source Repositories, or git-compatible repositories like GitHub and Bitbucket.

To generate a build with Cloud Build, you define a series of steps. For example, you can configure build steps to fetch dependencies, compile source code, run integration tests, or use tools such as Docker, Gradle, and Maven. Each build step in Cloud Build runs in a Docker container.

Then Cloud Build can deliver your newly built images to various execution environments: not only GKE, but also App Engine and Cloud Functions.

Agenda

Introduction to Containers

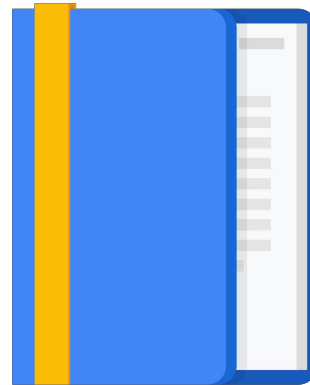
[Lab: Working with Cloud Build](#)

Introduction to Kubernetes

Introduction to Google Kubernetes
Engine

Computing Options

Summary



Lab Intro

Working with Cloud Build



In this lab, you'll build a Docker container image from provided code and a Dockerfile using Cloud Build. You'll then upload the container to Container Registry. The tasks that you will perform include using Cloud Build to build and push containers and using Container Registry to store and deploy containers.

Agenda

Introduction to Containers

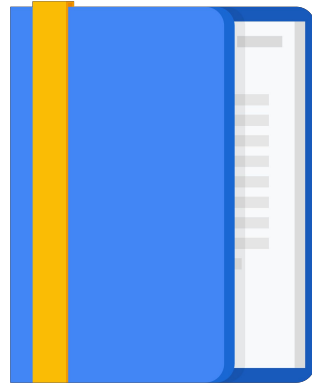
Lab: Working with Cloud Build

[Introduction to Kubernetes](#)

Introduction to Google Kubernetes
Engine

Computing Options

Summary



Now let's introduce a popular container management and orchestration solution called Kubernetes.

Managing your container infrastructure

You've embraced containers, but managing them at scale is a challenge

What can you do to better manage your container infrastructure?

Kubernetes!

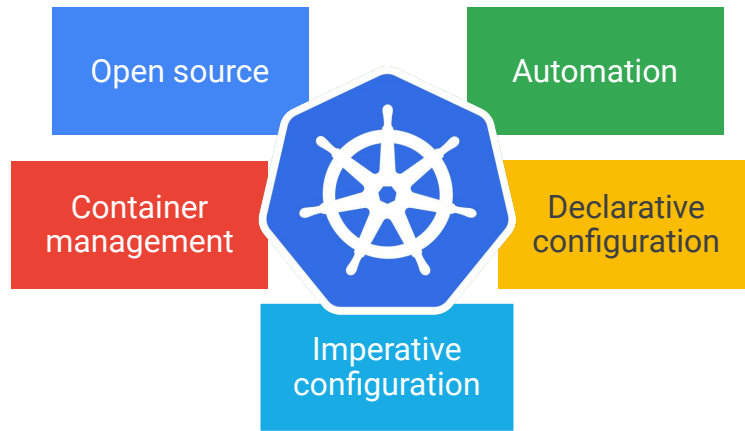


Your organization has really embraced the idea of containers. Because containers are so lean, your coworkers are creating them in numbers far exceeding the counts of virtual machines you used to have. And the applications running in them need to communicate over the network, but you don't have a network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

What is Kubernetes?



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.

Kubernetes features

- Supports both stateful and stateless applications
- Autoscaling
- Resource limits
- Extensibility
- Portability



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 Google Cloud and others. This allows Kubernetes to be deployed anywhere. You can move Kubernetes workloads freely without vendor lock-in.

Agenda

Introduction to Containers

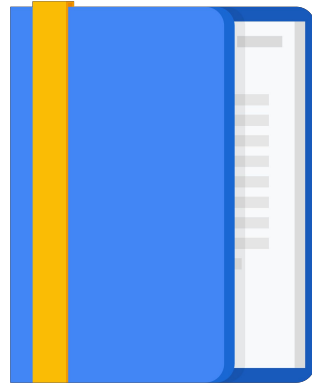
Lab: Working with Cloud Build

Introduction to Kubernetes

Introduction to Google Kubernetes
Engine

Computing Options

Summary



Google Cloud's managed services offering for Kubernetes is called Google Kubernetes Engine, or GKE. Why do people choose it?

Managing Kubernetes within Google Cloud

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

Is there a managed service for Kubernetes within Google Cloud?

Yes! Google Kubernetes Engine

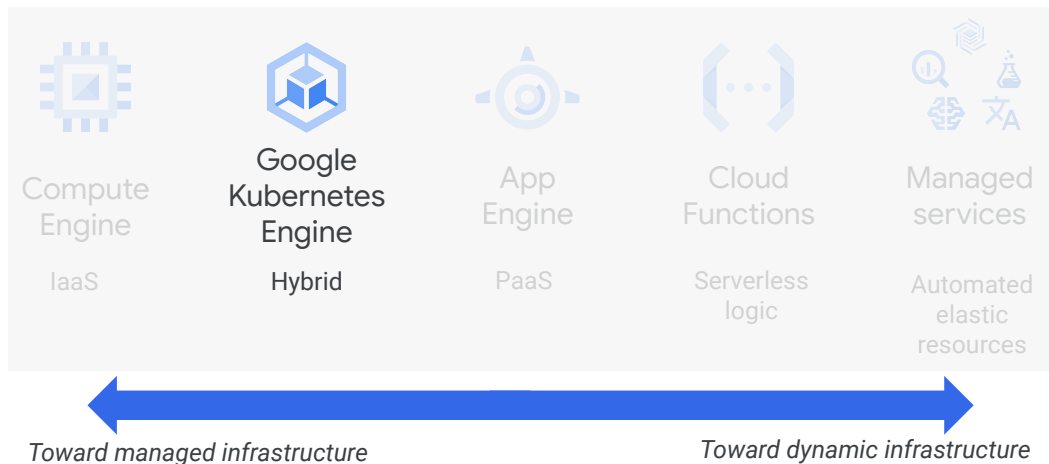


What if you have begun using Kubernetes in your environment, but the infrastructure has become a burden to maintain?

Is there anything within Google Cloud that can help you?

Absolutely yes. Google Cloud offers a managed Kubernetes solution called Google Kubernetes Engine.

GKE lets you deploy workloads easily



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 Google Cloud.

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

GKE has many features

| | | |
|--------------------------------|-----------------------------------|-----------------------|
| Fully managed | Container-optimized OS | Autopilot |
| Auto repair | Auto upgrade | Cluster scaling |
| Identity and access management | Integrated logging and monitoring | Integrated networking |
| Seamless integration | Google Cloud console | |



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.

GKE Autopilot is a mode of operation in GKE in which Google manages your cluster configuration, including your nodes, scaling, security, and other preconfigured settings

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 Artifact 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.

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

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

And finally the Google Cloud 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 Google Cloud console is a dashboard for your GKE clusters and workloads that you don't have to manage, and it's more powerful than the Kubernetes dashboard.

Agenda

Introduction to Containers

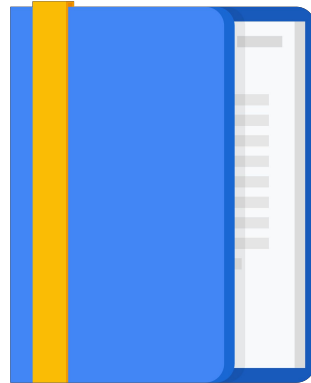
Lab: Working with Cloud Build

Introduction to Kubernetes

Introduction to Google Kubernetes
Engine

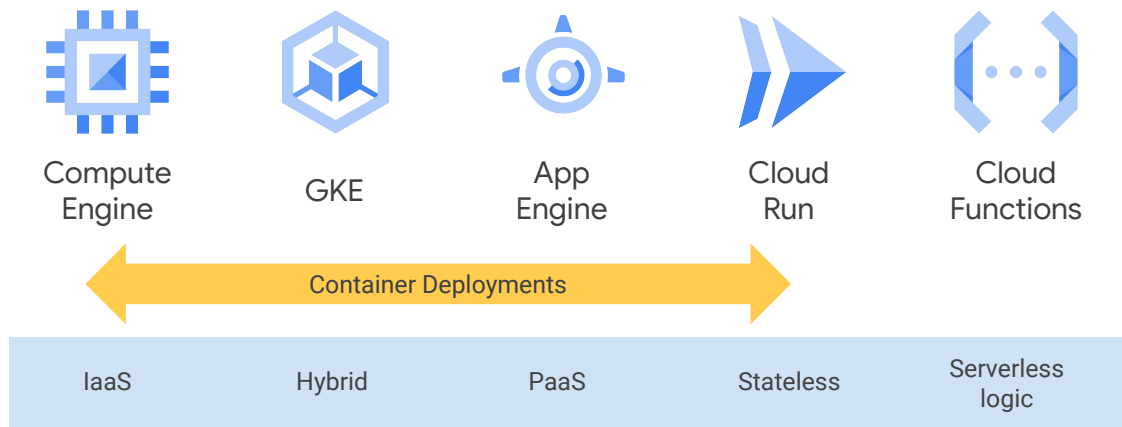
Computing Options

Summary



In this last lesson, you'll learn about the available computing options. In a previous module, I briefly introduced your choices for running compute workloads in Google Cloud. Now that we know more about how containers work, we can compare these choices in more detail.

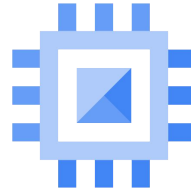
Comparing Google Cloud computing solutions



The services are Compute Engine, GKE, App Engine, Cloud Run, and Cloud Functions. At the end of this lesson, you will understand why people choose each.

Compute Engine

- Fully customizable virtual machines.
- Persistent disks and optional local SSDs.
- Global load balancing and autoscaling.
- Per-second billing.



Compute Engine offers virtual machines running on Google Cloud. You can select predefined VM configurations. At the time this course was developed, these virtual machines could be as large as 224 vCPUs and more than 12 TB of memory. You can also create customized configurations to precisely match your performance and cost requirements.

Virtual machines need block storage. Compute Engine offers you two main choices: Persistent disks and local SSDs. Persistent disks offer network storage that can scale up to 257 TB, and you can easily take snapshots of these disks for backup and mobility. You can also choose local SSDs, which enable very high input/output operations per second.

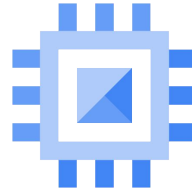
You can place your Compute Engine workloads behind global load balancers that support autoscaling. Compute Engine offers a feature called *managed instance groups*; with these, you can define resources that are automatically deployed to meet demand.

Google Cloud enables fine-grained control of costs of Compute Engine resources by providing per-second billing. This granularity helps reduce your costs when deploying compute resources for short periods of time, such as batch processing jobs. Compute Engine offers preemptible virtual machines, which provide significantly cheaper

pricing for your workloads that can be interrupted safely.

Compute Engine use cases

- Complete control over the OS and virtual hardware.
- Well suited for lift-and-shift migrations to the cloud.
- Most flexible compute solution, often used when a managed solution is too restrictive.



So why do people choose Compute Engine?

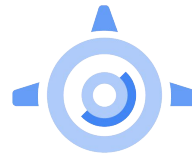
With Compute Engine, you have complete control over your infrastructure. You can customize operating systems, and even run applications that rely on a mix of operating systems.

You can easily lift and shift your on-premises workloads into Google Cloud without rewriting your applications or making any changes.

Compute Engine is the best option when other computing options don't support your application or requirements.

App Engine

- Provides a fully managed, code-first platform.
- Streamlines application deployment and scalability.
- Provides support for popular programming languages and application runtimes.
- Supports integrated monitoring, logging, and diagnostics.
- Simplifies version control, canary testing, and rollbacks.



App Engine has a completely different orientation from Compute Engine. App Engine is a fully managed application platform. Using App Engine means zero server management and zero configuration deployments.

So if you're a developer, you can focus on building applications and not worry about the deployment part.

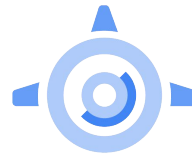
If you're a developer, you can simply upload your code, and App Engine will deploy the required infrastructure. App Engine supports popular languages like Java, Node.js., Python, PHP, C#, .NET, Ruby, and Go. You can also run container workloads.

Cloud Monitoring, Cloud Logging, and diagnostics such as Cloud Profiler and Error Reporting are also tightly integrated with App Engine. Google Cloud's operation suite integrates with tools such as the Cloud SDK, Cloud Source Repositories, IntelliJ, Visual Studio, and Powershell.

App Engine also supports version control and traffic splitting.

App Engine use cases

- Websites
- Mobile app and gaming backends
- RESTful APIs



App Engine is a good choice if you simply need to focus on writing code and do not need or want to worry about building a highly reliable and scalable infrastructure. You can focus on building applications instead of deploying and managing the environment.

Use cases of App Engine include web sites, mobile app and gaming backends, and as a way to present a RESTful API to the internet. What's a RESTful API? In short, it's an application program interface that resembles the way a web browser interacts with a web server. RESTful APIs are easy for developers to work with and extend, and App Engine makes them easy to operate.

Google Kubernetes Engine

- Fully managed Kubernetes platform.
- Supports cluster scaling, persistent disks, automated upgrades, and auto node repairs.
- Built-in integration with Google Cloud services.
- Portability across multiple environments
 - Hybrid computing
 - Multi-cloud computing



Next, the main topic of this course: Google Kubernetes Engine. We learned that Kubernetes is an orchestration system for applications in containers. It automates deployment, scaling, load balancing, logging and monitoring, and other management features. Google Kubernetes Engine extends Kubernetes management on Google Cloud by adding features and integrating with other Google Cloud services automatically.

GKE supports cluster scaling, persistent disks, automated upgrades with the latest version of Kubernetes, and auto repair for unhealthy nodes.

It has built-in integration with Cloud Build, Container Registry, Cloud Monitoring, and Cloud Logging.

Existing workloads running within on-premises clusters can be easily moved onto Google Cloud. There's also no vendor lock-in.

GKE use cases

- Containerized applications.
- Cloud-native distributed systems.
- Hybrid applications.



Overall, GKE is very well-suited for containerized applications, cloud-native distributed systems, and hybrid applications.

Kubernetes and GKE are discussed in-depth throughout this course.

Cloud Run

- Enables stateless containers.
- Abstracts away infrastructure management.
- Automatically scales up and down.
- Open API and runtime environment.



Cloud Run is a managed compute platform that enables you to run stateless containers via web requests or Pub/Sub events. Cloud Run is serverless: it abstracts away all infrastructure management so you can focus on developing applications. It is built on Knative, an open-source, Kubernetes-based platform. It builds, deploys, and manages modern serverless workloads. Cloud Run gives you the choice of running your containers either fully-managed or in your own GKE cluster.

Cloud Run automatically scales up and down from zero depending on traffic almost instantaneously, so you never have to worry about scale configuration. Cloud Run also charges you only for the resources you use (calculated down to the nearest 100 milliseconds), so you will never have to pay for your over-provisioned resources.

With Cloud Run you can deploy your stateless containers with a consistent developer experience to a fully managed environment or to your own GKE cluster. This common experience is enabled by Knative, an open API and runtime environment built on Kubernetes that gives you freedom to move your workloads across different environments and platforms: fully managed on Google Cloud, on GKE, or anywhere Knative runs.

Cloud Run use cases

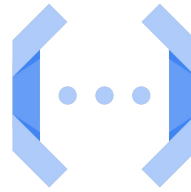
- Deploy stateless containers that listen for requests or events.
- Build applications in any language using any frameworks and tools.



Cloud Run enables you to deploy stateless containers that listen for requests or events delivered via HTTP. With Cloud Run, you can build your applications in any language using whatever frameworks and tools you want and deploy them in seconds without having to manage the server infrastructure.

Cloud Functions

- Event-driven, serverless compute service.
- Automatic scaling with highly available and fault-tolerant design.
- Charges apply only when your code runs.
- Triggered based on events in Google Cloud services, HTTP endpoints, and Firebase.



Cloud Functions is an event-driven, serverless compute service for simple, single-purpose functions that are attached to events. In Cloud Functions, you simply upload your code written in JavaScript, Python, or Go; Google Cloud will automatically deploy appropriate computing capacity to run that code.

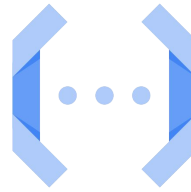
These servers are automatically scaled and are deployed with a highly available and fault-tolerant design.

You're only charged for the time your code runs. For each function, invocation memory and CPU use is measured in 100 millisecond increments, rounded up to the nearest increment. Cloud Functions also provides a perpetual free tier, so many Cloud Function use cases can be free of charge.

With Cloud Functions, your code is triggered within a few milliseconds based on events: for example, a file being uploaded to Google Cloud Storage, or a message being received with Pub/Sub. Cloud Functions can also be triggered based on HTTP endpoints you define and events in the Firebase mobile-application backend.

Cloud Functions use cases

- Supporting microservice architecture.
- Serverless application backends.
 - Mobile and IoT backends.
 - Integrate with third-party services and APIs.
- Intelligent applications.
 - Virtual assistant and chat bots.
 - Video and image analysis.

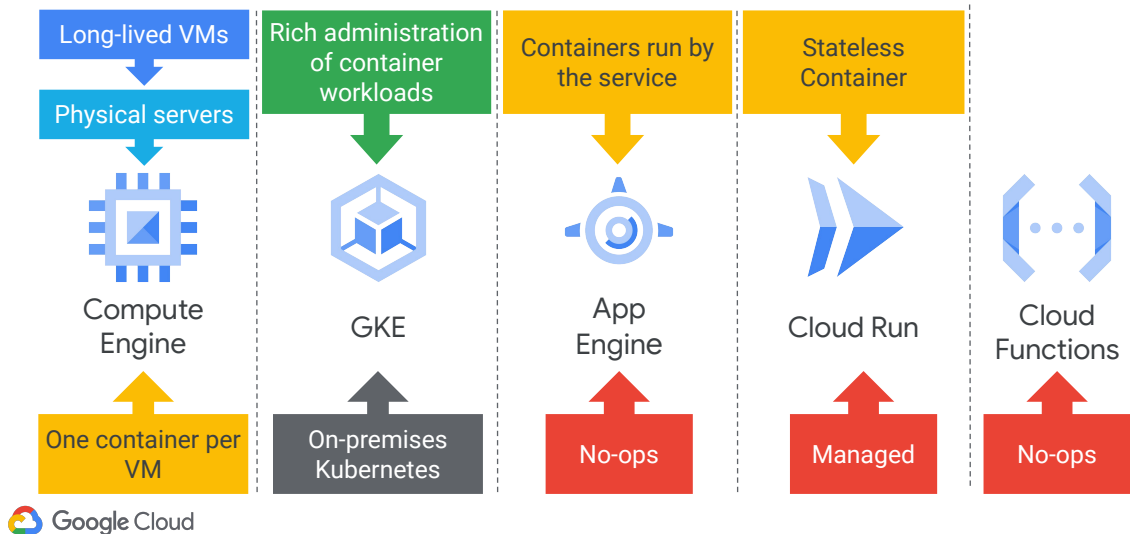


Cloud Functions can be used as part of a microservices application architecture.

You can also build simple, serverless mobile or IoT backends or integrate with third-party services and APIs. Files uploaded to your Cloud Storage bucket can be processed in real time. Similarly, the data can be extracted, transformed, and loaded for querying and analysis.

Google Cloud customers often use Cloud Functions as part of intelligent applications such as virtual assistants, video or image analysis, and sentiment analysis.

Which compute service should you adopt?



So which compute service should you adopt? A lot depends on where you're coming from.

If you're running applications on physical server hardware, it will be the path of least resistance to move into Compute Engine.

What if you're running applications in long-lived virtual machines, in which each VM is managed and maintained? In this case, you'll also find moving into Compute Engine is the quickest Google Cloud service for getting your applications into the cloud.

What if you don't want to think about operations at all? App Engine and Cloud Functions are good choices. You can learn more about the differences between App Engine and Cloud Functions in the courses under "Developing Applications with Google Cloud."

I hope that this course so far has helped you understand why software containers are so beneficial. Containerization is the most efficient and portable way to package an application. The popularity of containerization is growing very quickly. In fact, both Compute Engine and App Engine can launch containers for you. Compute Engine will accept a container image from you and launch a virtual machine instance containing it; you can use Compute Engine technologies to scale and manage the resulting VM.

And App Engine Flexible Environment will accept a container image from you and run it in the same no-ops environment that App Engine delivers for code.

But what if you want more control over your containerized workloads than App Engine offers, and denser packing than Compute Engine offers? That increasingly popular use case is what GKE is designed to address. The Kubernetes paradigm of container orchestration is incredibly powerful and vendor-neutral, and a broad and vibrant community has developed around it. Using Kubernetes as a managed service from Google Cloud saves you work and lets you benefit from all the other Google Cloud resources too. You can also choose Cloud Run to run stateless containers on a managed compute platform.

And of course, if you're already running Kubernetes in your on-premises data centers, moving to GKE is a great choice, because you'll be able to bring along both your workloads and your management approach.

Agenda

Introduction to Containers

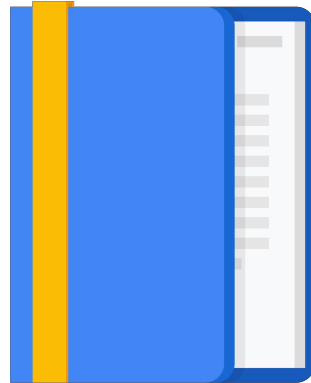
Lab: Working with Cloud Build

Introduction to Kubernetes

Introduction to Google Kubernetes
Engine

Computing Options

Summary



Summary

Create a container using Cloud Build.

Google Cloud compute solutions:

- Compute Engine
- App Engine
- Google Kubernetes Engine
- Cloud Run
- Cloud Functions

Store a container in Container Registry.

Compare and contrast Kubernetes and Google Kubernetes Engine (GKE) features.



That concludes 'Introduction to Containers and Kubernetes'.

In this module, you learned how to create a container using Cloud Build to build and manage your application images. You reviewed the five compute solutions offered by Google Cloud starting from Compute Engine for lift-and-shift workloads, App Engine, GKE, Cloud Run, and Cloud Functions and weighed the benefits of each with common use cases.

In your labs, you created docker-formatted container images and saw how you can privately store them in Container Registry. You also learned how to securely manage your builds with the native integration with Cloud IAM.

Lastly, you learned that GKE supports automatic Kubernetes versions updates, automatic repair for unhealthy nodes, and even scaling of the cluster itself for you.

All of this allows you to focus your time on dreaming up and writing your next great application. This means less time worrying about infrastructure, managing and maintaining deployment environments, securely building and sharing containers, and having to solve for massive scale when your amazing idea hits that next big user milestone.

In the next module, you'll learn about the architecture of Kubernetes.

