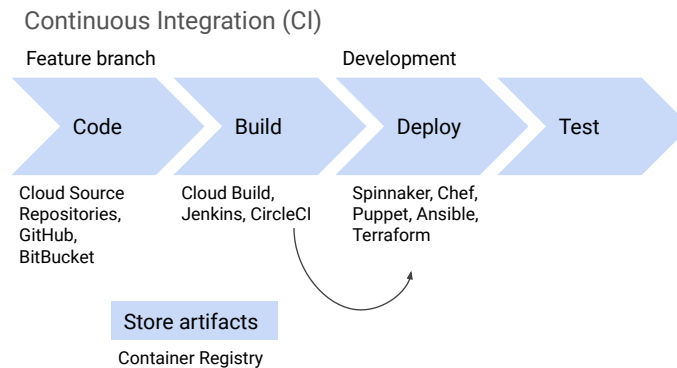# Google Cloud

Deploying Applications

Mike Dunker
Technical Curriculum Developer,
Google Cloud

To run reliable services, you must have reliable release processes. When you have many engineers building components of an application, it's crucial that they are able to run unit tests, integration tests, and other tests quickly.

With most user-facing software, teams usually want to release software often with new features and bug fixes. To enable high release velocity, build, test and release processes must be automated as much as possible. In the module Deploying Applications, you'll learn the components of a continuous integration and delivery pipeline. You'll also learn how to build container images for your application by using Cloud Build and push the images to Container Registry.

Implement continuous integration and delivery for reliable releases

Continuous Integration (CI)

Feature branch   Development

Code   Build   Deploy   Test

Cloud Source   Cloud Build,   Spinnaker, Chef,
Repositories,   Jenkins, CircleCI Puppet, Ansible,
GitHub,   Terraform
BitBucket

Store artifacts

Container Registry

Continuous integration is a developer workflow in which developers frequently pull from the main branch and commit their changes into a feature branch in a source code repository such as Cloud Source Repositories or GitHub. These commit triggers are built in a build system such as Jenkins or CircleCI.

The build process creates a new application image by using Cloud Build and stores it in an artifact repository such as container registry. A deployment system such as Spinnaker deploys the artifacts in your Cloud environment.

You can use Terraform to stand up resources for the managed services that your application needs. For example, you can use Terraform to create Cloud Storage buckets and Pub/Sub topics. After your application is deployed in your development environment, you can automatically run tests to verify your code. If all the tests pass, you can push changes from the feature branch to the main branch.

Implement continuous integration and delivery for reliable releases
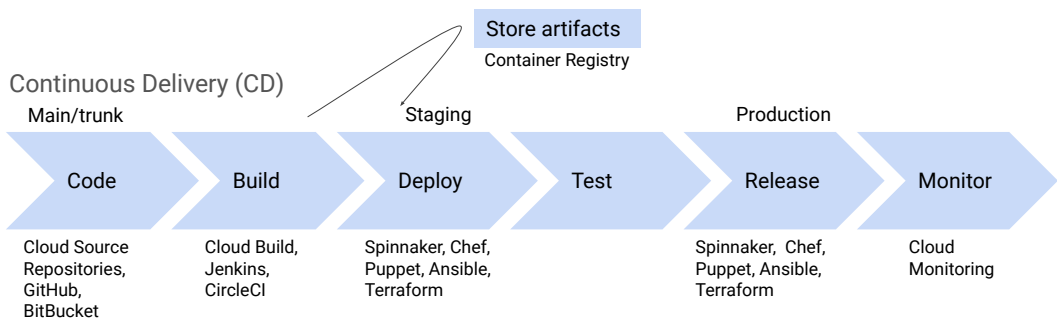
Continuous delivery is a workflow that is triggered when changes are pushed to the main branch in the repository. The build system builds the code and creates application images. The deployment system deploys the application images to the staging environment and runs integration tests, performance tests, and more. If all tests pass, the build is tagged as a release candidate build. You can manually approve a release candidate build. This approval can trigger deployment to production environments as a canary or blue-green release.
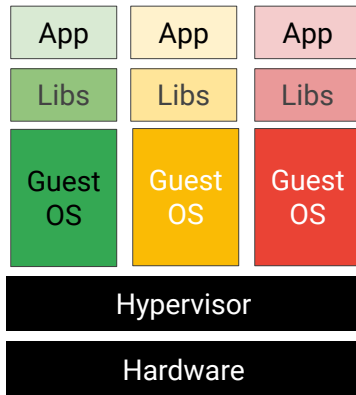
You can monitor the performance of your application in the production environment by using monitoring services such as Cloud Monitoring. If the new deployment functions optimally, you can switch over your entire traffic to this new release. But if you discover problems, you can also quickly roll back to the last stable release.

The continuous deployment workflow varies slightly in that there is no manual approval process. The deployment system automatically deploys release candidates to the production environment.
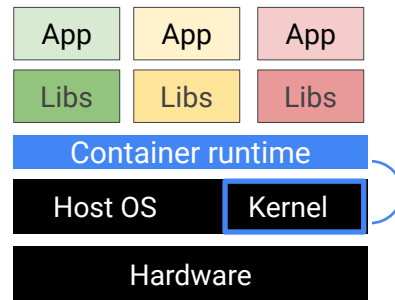
In the rest of this presentation, you'll consider two key aspects of your Cloud native application: the container image for your application and the Google Cloud resources required by your application.

Containers: an efficient way to isolate code and manage workloads

Hypervisor-based virtualization   *Container-based virtualization*

App	App	App
Libs	Libs	Libs
Guest OS	Guest OS	Guest OS
Hypervisor		
Hardware		

App	App	App
Libs	Libs	Libs
Container runtime		
Host OS	Kernel	
Hardware		

Let's do a quick refresher on Containers.

Container-based virtualization is an alternative to hardware virtualization, as used in traditional virtual machines. Virtual machines are isolated from one another in part by having each virtual machine have its own instance of the operating system. But operating systems can be slow to boot and can be resource-heavy. Containers respond to these problems by using modern operating systems' built-in capabilities to isolate environments from one another.
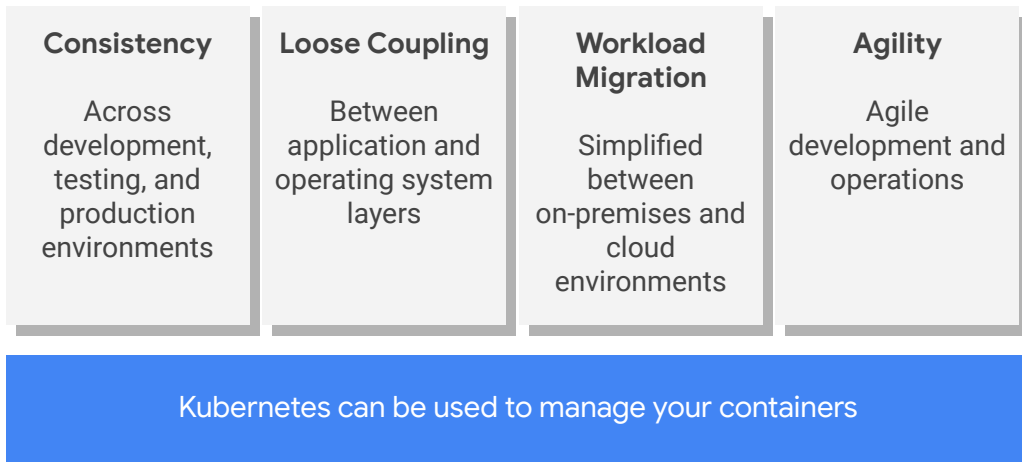
A process is a running program. In Linux and Windows, the memory address spaces of running processes have long been isolated from one another. Popular implementations of software containers build on this isolation. They take advantage of additional operating-system features that give processes the capability to have their own namespaces and that give a supervisor process the ability to limit other processes' access to resources.

Containers start much faster than virtual machines and use fewer resources, because each container does not have its own instance of the operating system. Instead, developers configure each container with a minimal set of software libraries to do the job. A lightweight container runtime does the plumbing jobs needed to allow that container to launch and run, calling into the kernel as necessary. The container runtime also determines the image format.

Google Kubernetes Engine uses the Docker container runtime, and Docker

containers are what we'll focus on in this course.

Why use containers?

Consistency	Loose Coupling	Workload Migration	Agility
Across development, testing, and production environments	Between application and operating system layers	Simplified between on-premises and cloud environments	Agile development and operations

Kubernetes can be used to manage your containers

So what does a container provide that a virtual machine does not?
- First, simple deployment: By packaging your application as a singularly addressable, registry-stored, one-command-line deployable component, a container radically simplifies the deployment of your app no matter where you're deploying it.
- Second, rapid availability: By abstracting just the OS instead of the whole physical computer, this package can "boot" in ~1/20th of a second, compared to a minute or so for a modern virtual machine.
- Third, the ability to create microservices: Containers allow developers and operators to further subdivide compute resources. If a micro VM instance seems like overkill for your app, or if scaling an entire VM at a time seems like a big step function, containers will make a big, positive impact in your systems.
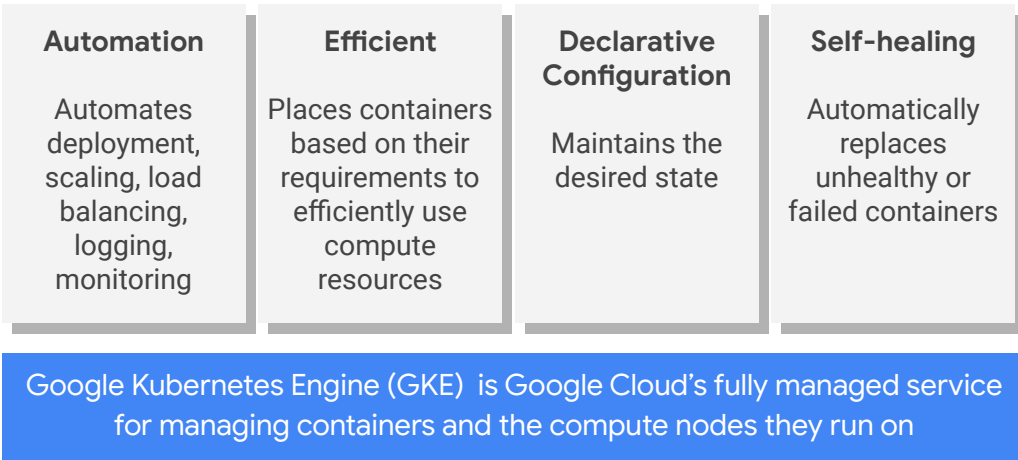
What are the implications of some of these advantages?

- A developer has in their laptop plenty of compute power to run multiple containers, making for easier and faster development. Although it is certainly possible to run several virtual machines on a laptop, it's far from fast, easy, or lightweight.
- Similarly, release administration is easier: pushing a new version of a container is a single command. Testing is also cheaper: in a public cloud

- where a VM is billed for a minimum of 10 minutes of compute time, a single test might only cost you a small amount. But if you're running thousands of programmatically driven tests per day, this starts to add up. With a container, you could do thousands of simple tests at the same cost, amounting to large savings for your production applications.
- Another implication is the composability of application systems using this model, especially with applications using open-source software. Although it might be a daunting systems administration task for a developer to install and configure MySQL, memcached, MongoDB, Hadoop, RabbitMQ, node.js, nginx, and so on together on a single box to provide a platform for their application, it is much easier and a vastly lower risk to start a few containers housing these applications with some very compact scripting. This model eliminates a significant amount of error-prone, specialized, boilerplate work.

We use Kubernetes to manage our containers.
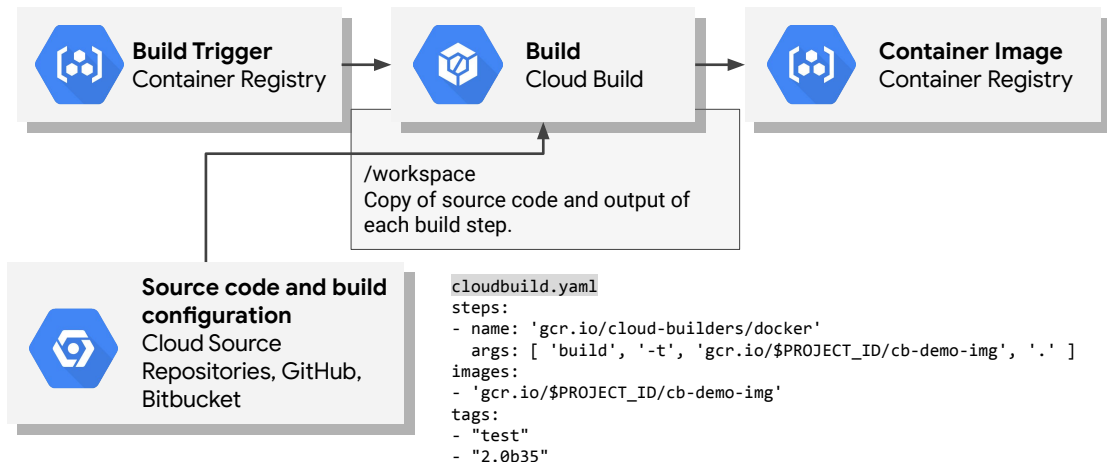
Why use Kubernetes?

Automation	Efficient	Declarative Configuration	Self-healing
Automates deployment, scaling, load balancing, logging, monitoring	Places containers based on their requirements to efficiently use compute resources	Maintains the desired state	Automatically replaces unhealthy or failed containers

Google Kubernetes Engine (GKE) is Google Cloud's fully managed service for managing containers and the compute nodes they run on

Kubernetes is an open-source tool which manages container orchestration. Benefits include:

- Automation: Kubernetes automates the deployment, scaling, load balancing, logging, monitoring, and other management features of containerized applications.
- Efficiency: By specifying CPU and memory requirements of containers, Kubernetes can efficiently pack containers on nodes to efficiently use compute nodes.
- Declarative configuration: You can configure the desired state you want to achieve, rather than issuing a series of commands to achieve that desired state. Kubernetes will make your system conform to that desired state. This greatly reduces the operational complexity of large systems.
- Self-healing: Kubernetes can automatically replace containers that have failed or are failing a health check. This leads to many fewer middle-of-the-night pages for operations staff.

Google Kubernetes Engine, or GKE, is Google Cloud's fully managed solution for running containers. GKE scales, repairs, and manages your cluster of compute nodes that runs your Kubernetes-managed containers.

Use Cloud Build and Container Registry to create application images


```
cloudbuild.yaml
steps:
- name: 'gcr.io/cloud-builders/docker'
  args: [ 'build', '-t', 'gcr.io/$PROJECT_ID/cb-demo-img', '.' ]
images:
- 'gcr.io/$PROJECT_ID/cb-demo-img'
tags:
- "test"
- "2.0b35"
```

The container image for your application is a complete package that contains the application binary and all the software that's required for the application to run. When you deploy the same container image on your development, test and production environments, you can be sure that your application will perform exactly the same way in each of these environments. Cloud Build is a fully managed service that enables you to set up build pipelines to create a Docker container image for your application and push the image to a Google Cloud container registry. You don't need to download all build tools and container images to a build machine or manage your own build infrastructure.

By using Container Registry and Cloud Build, you can create build pipelines that are automatically triggered when you commit code to a repository. In Container Registry, you can create a build trigger that is executed based on a trigger type. A trigger type specifies whether build should be triggered based on commits to a particular branch in a repository or commits that contain a particular tag.

You must also create a build configuration file that specifies the steps in the build pipeline. Steps are analogous to commands or scripts that you execute to build your application. Each build step is a Docker container that's invoked by Cloud Build when the build is executed. The step name identifies the container to invoke for the build step. The images attribute contains the name of the container image to be created by this build configuration. Cloud Build enables you to specify different types of source repositories, tag container images to enable searches, and create build steps that perform operations such as downloading and processing data, without even creating

a container image. The build configuration can be specified in a YAML or JSON format.

Cloud Build mounts your source code into the "/workspace" directory of the Docker container associated with a build step. The artifacts produced by each build step are persisted in the "/workspace" folder and can be used by the following build step. Cloud Build automatically pushes the built container image to Container Registry. In Container Registry, you can view the status and history of builds. Cloud Build also publishes build status notifications to Pub/Sub. You can subscribe to these notifications to take action based on build status or other attributes.

For more information, refer to the documentation links in the Course Resources.
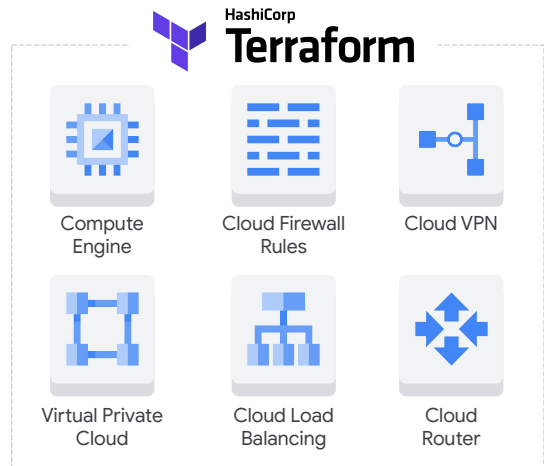
[Cloud Build Documentation: https://cloud.google.com/cloud-build/docs/]
[Cloud Build API documentation:
https://cloud.google.com/cloud-build/docs/api/reference/rest/]
[Container Registry: https://cloud.google.com/container-registry/docs/]

Use Terraform to provision your Google Cloud resources

- Repeatable deployment process
- Declarative language
- Focus on the application
- Parallel deployment
- Template-driven

HashiCorp
Terraform

Compute Engine

Cloud Firewall Rules

Cloud VPN

Virtual Private Cloud

Cloud Load Balancing

Cloud Router

Terraform is an open source tool that lets you provision Google Cloud resources.

Terraform lets you provision Google Cloud resources—such as virtual machines, containers, storage, and networking—with declarative configuration files. You just specify all the resources needed for your application in a declarative format and deploy your configuration. HashiCorp Configuration Language, or HCL, allows for concise descriptions of resources using blocks, arguments, and expressions.

This deployment can be repeated over and over with consistent results, and you can delete an entire deployment with one command or click. The benefit of a declarative approach is that it allows you to specify what the configuration should be and let the system figure out the steps to take.

Instead of deploying each resource separately, you specify the set of resources that compose the application or service, which allows you to focus on the application. Unlike Cloud Shell, Terraform deploys resources in parallel.

Terraform uses the underlying APIs of each Google Cloud service to deploy your resources. This enables you to deploy almost everything we have seen so far, from instances, instance templates, and groups, to VPC networks, firewall rules, VPN tunnels, Cloud Routers, and load balancers.

For a full list of supported resource types, refer to the Using Terraform with Google Cloud documentation link in the Course Resources.

Terraform language

- Terraform language is the interface to declare resources.

- Resources are infrastructure objects.

- The configuration file guides the management of the resource.

```
resource "google_compute_network" "default" {
  name = "${var.network_name}"
  auto_create_subnetworks = false
}

<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}
```

The Terraform language is the user interface to declare resources. Resources are infrastructure objects such as Compute Engine virtual machines, storage buckets, containers, or networks. A Terraform configuration is a complete document in the Terraform language that tells Terraform how to manage a given collection of infrastructure. A configuration can consist of multiple files and directories.

The syntax of the Terraform language includes:

- Blocks that represent objects and can have zero or more labels. A block has a body that enables you to declare arguments and nested blocks.
- Arguments are used to assign a value to a name.
- An expression represents a value that can be assigned to an identifier.

Terraform can be used on multiple public and private clouds

- Considered a first-class tool in Google Cloud
- Already installed in Cloud Shell

```
provider "google" {
   region      = "us-central1"
}

resource "google_compute_instance" {
   name        = "instance name"
   machine_type= "n1-standard-1"
   zone        = "us-central1-f"

   disk {
      image = "image to build instance"
   }
}

output "instance_ip" {
   value = "${google_compute.ip_address}"
}
```

Terraform can be used on multiple public and private clouds. Terraform is already installed in Cloud Shell.

The example Terraform configuration file shown starts with a provider block that indicates that Google Cloud is the provider. The region for the deployment is specified inside the provider block.

The resource block specifies a Google Cloud Compute Engine instance, or virtual machine. The details of the instance to be created are specified inside the resource block.

The output block specifies an output variable for the Terraform module. In this case, a value will be assigned to the output variable "instance_ip."

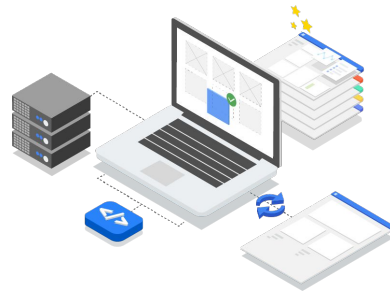# Use the Cloud Foundation Toolkit to optimize your Terraform deployment

- Reference templates, which reflect Google Cloud best practices
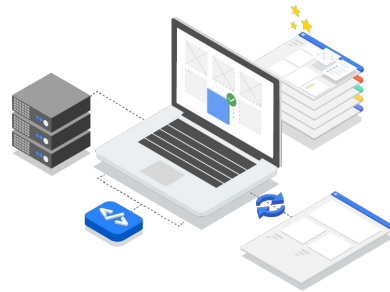
The Cloud Foundation Toolkit provides a series of reference templates for Config Connector and Terraform which reflect Google Cloud best practices.

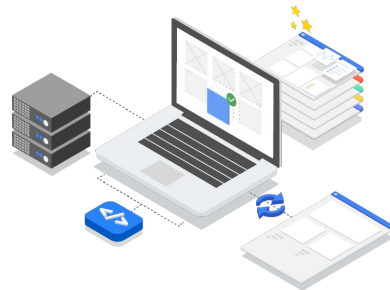# Use the Cloud Foundation Toolkit to optimize your Terraform deployment

- Reference templates, which reflect Google Cloud best practices
- Repeatable and enterprise-ready

These off-the-shelf templates can be used to quickly build a repeatable, enterprise-ready foundation in Google Cloud. Using the open-source templates, you can automate repeatable tasks, and provision entire environments in a consistent fashion. Teams can collaborate on the infrastructure by participating in code reviews and suggesting source code changes.

# Use the Cloud Foundation Toolkit to optimize your Terraform deployment

- Reference templates, which reflect Google Cloud best practices
- Repeatable and enterprise-ready
- Designed to meet compliance and security needs

The Cloud Foundation Toolkit is designed specifically to meet the compliance and security needs of enterprises. By creating a foundational environment using these templates, you can be confident that best practices are implemented out of the box — including key security and governance controls.

Use the Cloud Foundation Toolkit to optimize your Terraform deployment

- Reference templates, which reflect Google Cloud best practices
- Repeatable and enterprise-ready
- Designed to meet compliance and security needs
- Provides a consistent set of tools and patterns
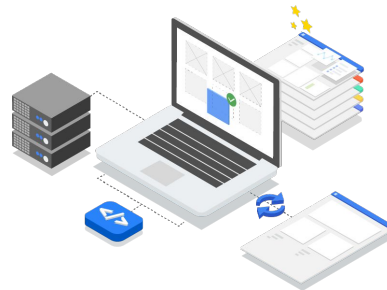
With the Cloud Foundation Toolkit, you don't need to spend time developing your own templates and patterns for Google Cloud. Instead, you can build on open-source templates and focus only on the customizations which are unique to your company and workloads. Developers can move faster and migrations are less time-consuming.

Each template from the Cloud Foundation Toolkit can be used independently. You can choose which patterns make sense for your organization, and add new ones as your environment evolves. The open-source templates can easily be forked and modified to suit your organization's needs. By adopting the toolkit, you can be confident that different teams are deploying their applications and environments using a consistent set of tools and patterns. This reduces the potential for misconfigurations and inconsistencies, while allowing easier collaboration across different teams.

For more information,refer to the Cloud Foundation Toolkit documentation link in the Course Resources.

Lab

Deploying the Application into Google Kubernetes Engine

Duration: 45 minutes

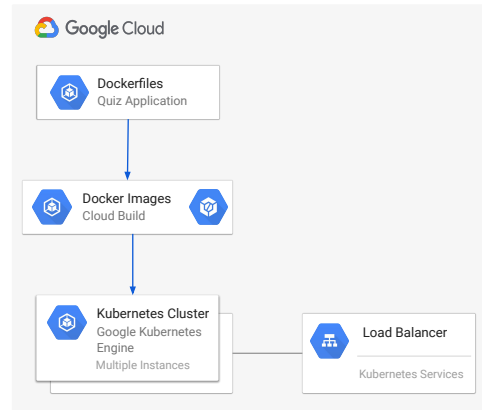In this lab, you will deploy an application into Google Kubernetes Engine.

Lab objectives

Create Dockerfiles to package up the Quiz application frontend and backend code for deployment

Harness Cloud Build to produce Docker images

Provision a GKE cluster to host the Quiz application

Employ Kubernetes deployments to provision replicated pods into GKE

Leverage a Kubernetes service to provision a load balancer for the Quiz frontend

Google Cloud

Google Cloud

Dockerfiles
Quiz Application

Docker Images
Cloud Build

Kubernetes Cluster
Google Kubernetes Engine
Multiple Instances

Load Balancer
Kubernetes Services

In the lab, you will create Docker files to describe how your application's container image should be built, you'll use Cloud Build to build Docker images and store the images in Container Registry, you'll launch a Kubernetes cluster in GKE and deploy your application's container image in the cluster. Finally, you'll expose the front-end service by using a load balancer.

Google Cloud

Summary

To summarize, implement continuous integration and delivery pipelines to enable repeatable and reliable deployments. Treat your infrastructure as code. This approach will enable you to create and store versions of your infrastructure. In case of unexpected problems, you can roll back to a previous version of the infrastructure and application image that works. Using Cloud Build, you can build Docker images for your application from source code located in Cloud Storage, Cloud Source Repositories, GitHub, or Bitbucket. Cloud Build automatically pushes these images to Container Registry. Your application's binaries as well as dependencies will all be packaged and released as one unit. Your application can then function consistently in any environment. You can use Terraform to stand up your Google Cloud infrastructure including Compute Engine VMs, GKE, and more.