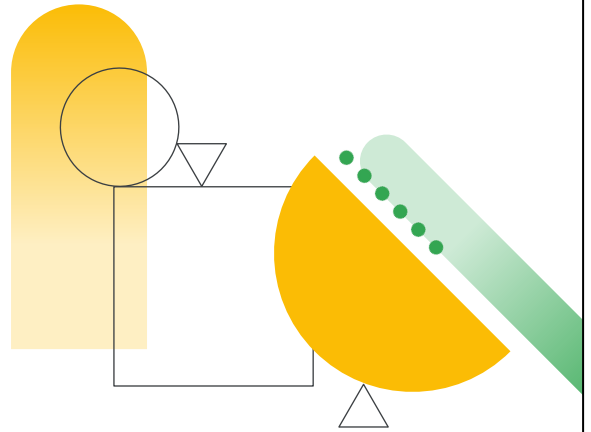


DevOps Automation



This module introduces DevOps automation, a key factor in achieving consistency, reliability, and speed of deployment.

Learning objectives

- 01 Automate service deployment using CI/CD pipelines.
- 02 Leverage Cloud Source Repositories for source and version control.
- 03 Automate builds with Cloud Build and build triggers.
- 04 Manage container images with Container Registry.
- 05 Investigate infrastructure with code using Terraform.



Specifically, we will talk about services that support continuous integration and continuous delivery practices, part of a DevOps way of working.

With DevOps and microservices, automated pipelines for integrating, delivering, and potentially deploying code are required. These pipelines ideally run on on-demand provisioned resources. This module introduces the Google Cloud tools for developing code and creating automated delivery pipelines that are provisioned on demand.

We will talk about using Cloud Source Repositories for source and version control, Cloud Build, including build triggers, for automating builds, and managing containers with Container Registry.

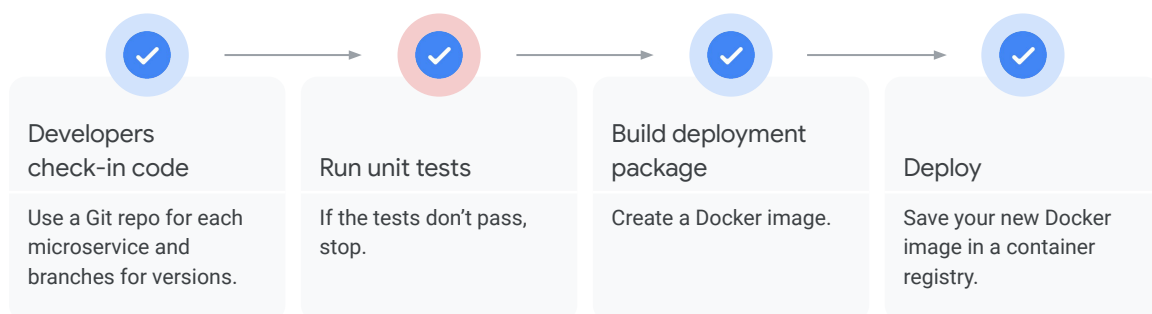
We'll finish by reviewing infrastructure as code tools like Terraform.



Continuous Integration Pipelines

Let's begin by talking about continuous integration pipelines.

Continuous integration pipelines automate building applications



Continuous integration pipelines automate building applications. This graphic shows a very simplistic view of a pipeline, which would be customized to meet your requirements.

The process starts with checking code into a repository where all the unit tests are run. On successful passing of the tests, a deployment package is built as a Docker image. This image is then saved in a container registry from where it can be deployed.

Each microservice should have its own repository.

Typical extra steps include linting of code/quality analysis by tools such as SonarQube, integration tests, generating test reports, and image scanning.

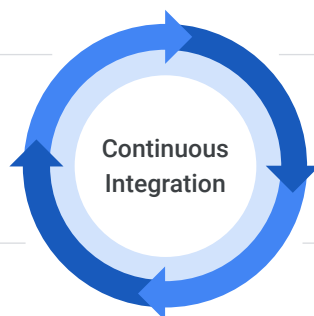
Google provides the components required for a continuous integration pipeline

Cloud Source Repositories

Developers push to a central repository when they want a build to occur.

Container Registry

Store your Docker images or deployment packages in a central location for deployment.



Cloud Build

Build system executes the steps required to make a deployment package or Docker image.

Build triggers

Watches for changes in the Git repo and starts the build.

Google Cloud

Google Cloud provides the components required to build a continuous integration pipeline. Let's go through each of those.

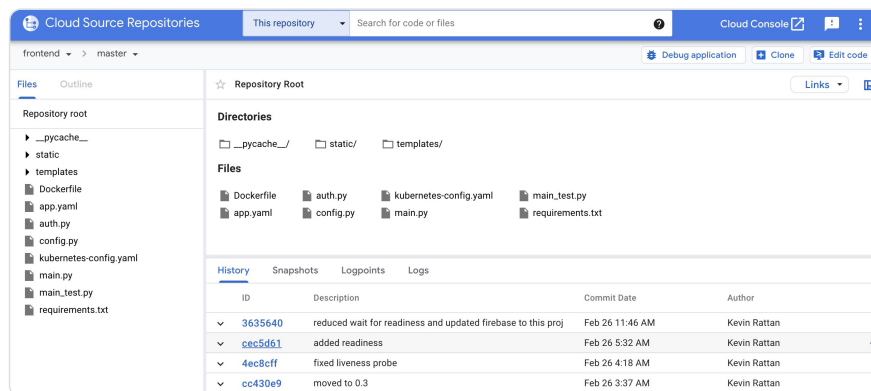
- The **Cloud Source Repositories** service provides private Git repositories hosted on Google Cloud. These repositories let you develop and deploy an app or service in a space that provides collaboration and version control for your code. Cloud Source Repositories is integrated with Google Cloud, so it provides a seamless developer experience.
- **Cloud Build** executes your builds on Google Cloud infrastructure. It can import source code from Cloud Storage, Cloud Source Repositories, GitHub, or Bitbucket, execute a build to your specifications, and produce artifacts such as Docker containers or Java archives. Cloud Build executes your build as a series of build steps, where each build step is run in a Docker container. A build step can do anything that can be done from a container, irrespective of the environment. There are standard steps, or you can define your own steps.
- A **Cloud Build trigger** automatically starts a build whenever you make any changes to your source code. You can configure the trigger to build your code on any changes to the source repository or only changes that match certain criteria.
- **Container Registry** is a single place for your team to manage Docker images or deployment packages, perform vulnerability analysis, and decide who can

- access what with fine-grained access control.

Let's go through each of these services in more detail.

Cloud Source Repositories provides managed Git repositories

Control access to your repos using IAM within your Google Cloud projects.



Google Cloud

Cloud Source Repositories provides managed Git repositories.

You can use IAM to add team members to your project and to grant them permissions to create, view, and update repositories.

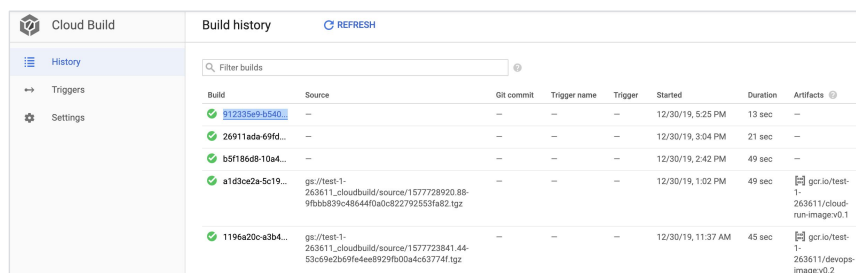
Repositories can be configured to publish messages to a specified Pub/Sub topic. Messages can be published when a user creates or deletes a repository or pushes a commit.

Some other features of Cloud Source Repositories include the ability to debug in production using Cloud Debugger, audit logging to provide insights into what actions were performed where and when, and direct deployment to App Engine. It is also possible to connect an existing GitHub or Bitbucket repository to Cloud Source Repositories. Connected repositories are synchronized with Cloud Source Repositories automatically.

Cloud Build lets you build software quickly across all languages

- Google-hosted Docker build service
 - Alternative to using Docker build command
- Use the CLI to submit a build


```
gcloud builds submit --tag gcr.io/your-project-id/image-name
```



Build	Source	Git commit	Trigger name	Trigger	Started	Duration	Artifacts
912335e9-6540...	---	---	---	---	12/30/19, 5:25 PM	13 sec	---
26911ada-69fd...	---	---	---	---	12/30/19, 3:04 PM	21 sec	---
b5f186d8-10a4...	---	---	---	---	12/30/19, 2:42 PM	49 sec	---
a1d30e2a-5c19...	gs://test-1-263611_cloudbuild/source/1577728920.88-9fbb839c486440a0c822792553fa82.tgz	---	---	---	12/30/19, 1:02 PM	49 sec	gcr.io/test-1-263611/cloud-run-image:v0.1
1196a20c-a3b4...	gs://test-1-263611_cloudbuild/source/1577723841.44-53c69e2b59fe4ee8929fb00a4c63774f.tgz	---	---	---	12/30/19, 11:37 AM	45 sec	gcr.io/test-1-263611/devops-image:v0.2

Google Cloud

Developers gain complete control over defining workflows for building, testing, and deploying across multiple environments including VMs, serverless, and Kubernetes.

There is no more need to provision or maintain build environments: all is handled by Cloud Build. You write a build config to provide instructions to Cloud Build on what tasks to perform. These are defined as a series of steps. Each step is executed by a cloud builder. Cloud builders are containers with common languages and tools installed in them.

Builders can be configured to fetch dependencies, run unit tests, static analyses and integration tests, and create artifacts with build tools such as docker, gradle, maven, bazel, and gulp.

Cloud Build executes the build steps you define. Executing build steps is similar to executing commands in a script.

You can either use the build steps provided by Cloud Build and the Cloud Build community or write your own custom build steps.

A GitHub link to a list of available cloud builders can be found in the course resources for this module.

[\[https://github.com/GoogleCloudPlatform/cloud-builders\]](https://github.com/GoogleCloudPlatform/cloud-builders)

Build triggers watch a repository and build a container whenever code is pushed

Supports Maven, custom builds, and Docker

The image displays three sequential steps of the 'Create trigger' process in Google Cloud Build:

- Step 1: Select source** - The user chooses a repository hosting option: ☒ Cloud Source Repository, ☐ GitHub, or ☐ Bitbucket. Buttons: Continue, Cancel.
- Step 2: Select repository** - The source is 'Cloud Source Repository'. The user filters repositories and selects 'default' (Cloud Source Repository). Buttons: Continue, Cancel.
- Step 3: Trigger settings** - The trigger is named 'Build My Docker Container'. The trigger type is 'Branch'. The branch regex is '*/'. The build configuration is 'Dockerfile' (Specify the path within the Git repo). The Dockerfile directory is '/'. Buttons: Continue, Cancel.

Google Cloud

Build triggers watch a repository and build a container whenever code is pushed. Google's build triggers support Maven, custom builds, and Docker.

A Cloud Build trigger automatically starts a build whenever a change is made to source code. It can be set to start a build on commits to a particular branch or on commits that contain a particular tag. You can specify a regular expression to match the branch or tag value. A GitHub link to the syntax for the regular expression is available in this module's course resources.

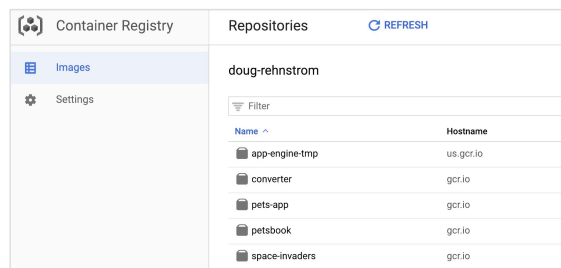
The build configuration can be specified either in a Dockerfile or a Cloud Build file. The configuration required is shown on this slide.

First, a source is selected. This can be Cloud Source Repositories, Github, or Bitbucket. In the next stage, a source repository is selected, followed by the trigger settings. The trigger settings include information like the branch or tag to use for trigger, and the build configuration, for example the Dockerfile or Cloud Build file.

[Syntax: <https://github.com/google/re2/wiki/Syntax>]

Container Registry is a Google Cloud-hosted Docker repository

- Images built using Cloud Build are automatically saved in Container Registry.
 - Tag images with the prefix **gcr.io/your-project-id/image-name**
- Can use Docker push and pull commands with Container Registry.
 - `docker push gcr.io/your-project-id/image-name`
 - `docker pull gcr.io/your-project-id/image-name`

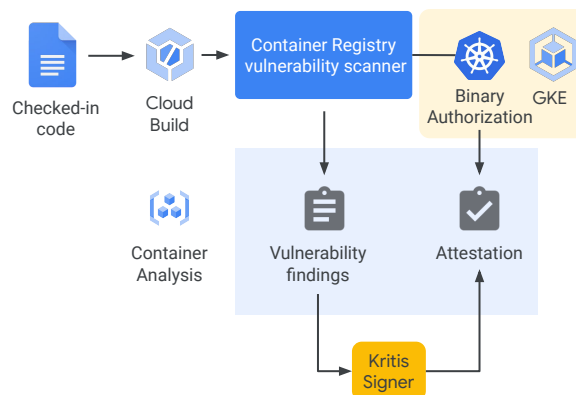


Container Registry provides a secure, private Docker image repository on Google Cloud. Image layer files and tags are stored in Cloud Storage. Cloud Storage IAM allows configuration of who can access, view, or download images. Container Registry provides native Docker support. This means you can push and pull Docker images to your private registry using the standard Docker command line interface. Multiple registries can be defined.

Container Analysis is a service that can also perform vulnerability scanning of images against a constantly updated database of known vulnerabilities. Container Analysis can be integrated with binary authorization to prevent the deployment of images with known security issues. Container Analysis must be explicitly enabled for a project.

Binary authorization allows you to enforce deploying only trusted containers into GKE

- Enable binary authorization on GKE cluster.
- Add a policy that requires signed images.
- When an image is built by Cloud Build an “attestor” verifies that it was from a trusted repository (Source Repositories, for example).
- Container Registry includes a vulnerability scanner that scans containers.



Google Cloud

Now binary authorization allows you to enforce deployment of only trusted containers into GKE. Binary authorization is a Google Cloud service and is based on the Kritis specification.

For this to work, you must enable binary authorization on your GKE cluster where your deployment will be made. A policy is required to sign the images. When an image is built by Cloud Build, an attestor verifies that it was from a trusted repository; for example, Source Repositories. Container Registry includes a vulnerability scanner that scans containers. A typical workflow is shown in the diagram.

Checkin of code triggers a Cloud Build. As part of the build, Container Registry will perform a vulnerability scan when a new image is uploaded. The scanner publishes messages to Pub/Sub. The Kritis Signer listens to Pub/Sub notifications from a container registry vulnerability scanner and makes an attestation if the image scanning passed the vulnerability scan. Google Cloud binary authorization service then enforces the policy requiring attestations by the Kritis signer before a container image can be deployed.

More details can be found at:

<https://cloud.google.com/binary-authorization/docs/cloud-build>

<https://cloud.google.com/binary-authorization/docs/vulnerability-scanning>



Infrastructure as Code

Let's now move on to consider infrastructure as code.

Moving to the cloud requires a mindset change

On-Premises

- Buy machines.
- Keep machines running for years.
- Prefer fewer big machines.
- Machines are capital expenditures.

Cloud

- Rent machines.
- Turn machines off as soon as possible.
- Prefer lots of small machines.
- Machines are monthly expenses.

Moving to the cloud requires a mindset change. The on-demand, pay-per-use model of cloud computing is a different model to traditional on-premises infrastructure provisioning. A typical on-premises model would be to buy machines and keep these running continuously. The compute infrastructure is typically built from fewer, larger machines. From an accounting view, the machines are capital expenditure that depreciates over time.

When using the cloud, resources are rented instead of purchased, and as a result we want to turn the machines off as soon as they are not required to save on costs. The approach is to typically have lots of smaller machines—scale out instead of scale up—and to expect and engineer for failure. From an accounting view, the machines are a monthly operating expense.

In the cloud, all infrastructure needs to be disposable

- Don't fix broken machines.
- Don't install patches.
- Don't upgrade machines.
- If you need to fix a machine, delete it and re-create a new one.
- To make infrastructure disposable, automate everything with code:
 - Can automate using scripts.
 - Can use declarative tools to define infrastructure.

In other words, in the cloud, all infrastructure needs to be disposable. The key to this is infrastructure as code (IaC), which allows for the provisioning, configuration, and deployment activities to be automated.

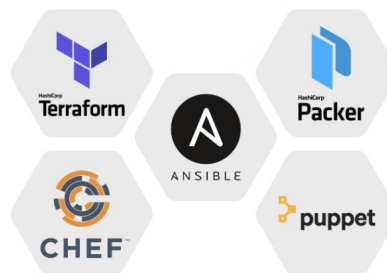
Having the process automated minimizes risks, eliminates manual mistakes, and supports repeatable deployments and scale and speed. Deploying one or one hundred machines is the same effort. The automation can be achieved using scripts or declarative tools such as Terraform which we will discuss later.

It is really important that no time is spent trying to fix broken machines or installing patches or upgrades. These will lead to problems recreating the environments at a later date. If a machine requires maintenance, remove it and create a new one instead.

Costs can be reduced by provisioning ephemeral environments, such as test environments that replicate the production environment.

Infrastructure as code (IaC) allows for the quick provisioning and removing of infrastructures

- Build an infrastructure when needed.
- Destroy the infrastructure when not in use.
- Create identical infrastructures for dev, test, and prod.
- Can be part of a CI/CD pipeline.
- Templates are the building blocks for disaster recovery procedures.
- Manage resource dependencies and complexity.
- Google Cloud supports many IaC tools.



Google Cloud

Terraform is one of the tools used for Infrastructure as Code or IaC. Before we dive into understanding Terraform, let's look at what infrastructure as code does. In essence, infrastructure as code allows for the quick provisioning and removing of infrastructures.

The on-demand provisioning of a deployment is extremely powerful. This can be integrated into a continuous integration pipeline that smoothes the path to continuous deployment.

Automated infrastructure provisioning means that the infrastructure can be provisioned on demand, and the deployment complexity is managed in code. This provides the flexibility to change infrastructure as requirements change. And all the changes are in one place. Infrastructure for environments such as development and test can now easily replicate production and can be deleted immediately when not in use. All because of infrastructure as code.

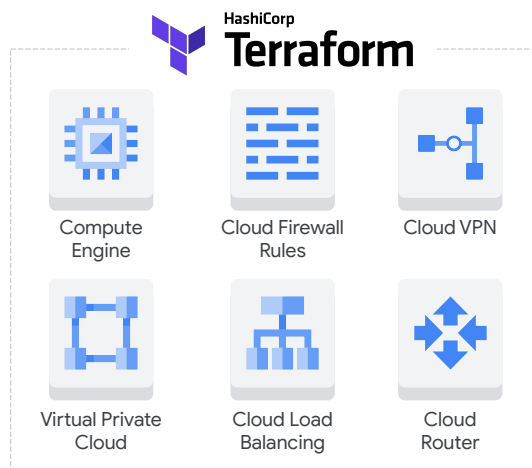
Several tools can be used for IaC. Google Cloud supports Terraform, where deployments are described in a file known as a configuration. This details all the resources that should be provisioned. Configurations can be modularized using templates, which allows the abstraction of resources into reusable components across deployments.

In addition to Terraform, Google Cloud also provides support for other IaC tools, including:

- Chef
- Puppet
- Ansible
- Packer

Terraform is an infrastructure automation tool

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



Google Cloud

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 (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 will deploy 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, see the documentation at [Using Terraform with Google Cloud](#).

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.instance_ip_address}"  
}
```

Google Cloud

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."

Lab Intro

Building a DevOps Pipeline



Google Cloud

In this first lab, you will build a DevOps pipeline using Cloud Source Repositories, Cloud Build, and Container Registry. Specifically, you will first create a Git repository. You will then write a simple Python application and add it to your repository. After that you will test your web application in Cloud Shell and then define a Docker build.

Once you define the build, you will use Cloud Build to create a Docker image and store the image in Container Registry. Then, you will see how to automate builds using triggers. Once you have the trigger, you will test it by making a change to your program and pushing that change to your Git repo.

Lab Review

Building a DevOps Pipeline



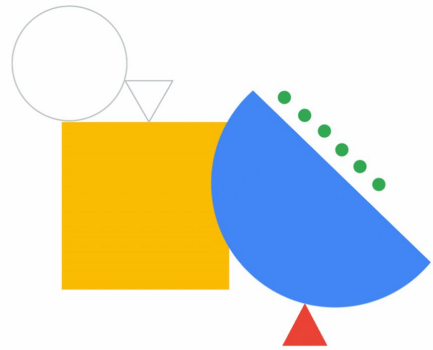
Google Cloud

In this lab, you learned how to use Google Cloud tools to create a simple and automated continuous integration pipeline. You used Cloud Source Repositories to create a Git repository and then used Cloud Build and triggers to automate the creation of your Docker images when code was checked into the repo.

When Cloud Build created your Docker images, it stored them in Container Registry. You saw how to access those images and test them in a Compute Engine VM.

You can stay for a lab walkthrough, but remember that Google Cloud's user interface can change, so your environment might look slightly different.

Review: DevOps Automation



In this module, you learned about services that you can use to help automate the deployment of your cloud resources and services. You used Cloud Source Repositories, Cloud Build, triggers, and Container Registry to create continuous integration pipelines. A continuous integration pipeline automates the creation of deployment packages like Docker images in response to changes in your source code.

You also saw how to automate the creation of infrastructure using the infrastructure as code tool, Terraform.